Deflated Shifted Block Krylov Subspace Methods for Hermitian Positive Definite Matrices

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

am Fachbereich Mathematik und Naturwissenschaften der
Bergischen Universität Wuppertal vorgelegte und genehmigte

Dissertation

von

Dipl.-Math. Sebastian Birk

Gutachter:	Prof. Dr. Andreas Frommer
Gutachter:	Prof. Dr. Bruno Lang

Dissertation eingereicht am:	2015-01-09
Tag der Disputation:	2015-04-28

Abstract

This thesis[1] considers the task of computing solutions of families of large sparse linear systems that differ by a shift with the identity matrix and have several different right-hand sides at the same time. We explore the applicability of existing Krylov subspace methods for solving shifted systems and methods for solving systems with multiple right-hand sides. Moreover, we develop methods that, based on deflated block Lanczos-type processes, exploit both features—shifts and multiple right-hand sides—at once and tackle well-known problems that multiple right-hand sides can bring along. We present numerical evidence that our methods can be superior as compared to applying other iterative methods, in typical situations.

Zusammenfassung

Diese Arbeit beschäftigt sich mit dem Lösen großer dünnbesetzter linearer Gleichungssysteme, deren Systemmatrizen um Vielfache der Einheitsmatrix verschoben sind und die gleichzeitig mehrere rechte Seiten umfassen. Wir untersuchen, wie gut sich eine Reihe von bestehenden Krylov-Unterraumverfahren zum Lösen dieser Gleichungssysteme eignet. Weiterhin entwickeln wir Verfahren, die beide Eigenschaften—sowohl das Verschieben um Vielfache der Einheitsmatrix als auch mehrere rechte Seiten—ausnutzen können und die auf deflationierten Block-Lanczos-Prozessen beruhen, die mit bekannten Problemen beim Lösen von Gleichungssystemen mit mehreren rechten Seiten umgehen können. Wir führen numerische Tests durch, die zeigen, dass unsere Verfahren in einigen Anwendungsfällen bestehenden Verfahren überlegen sind.

[1]This work was supported by Deutsche Forschungsgemeinschaft through the Collaborative Research Centre SFB-TRR 55 "Hadron Physics from Lattice QCD"

Für Bjarne, Finja und Doreen.

Contents

1 Introduction

This thesis focuses on the solution of families of shifted linear systems with multiple right-hand sides

$$(A + \sigma_i I)x_{i,j} = b_j, \quad i = 1, \dots, s, \quad j = 1, \dots, m \tag{1.1}$$

where $(A + \sigma_i I) \in \mathbb{C}^{n \times n}$ is a large sparse Hermitian positive definite matrix for every σ_i. This class of systems arises naturally for instance in lattice QCD. There, the action of a matrix on multiple vectors needs to be computed and the matrix itself stems from a matrix function evaluated for a large sparse matrix A. In other words, we are interested in computing $f(A)b_j$ for $b = 1, \dots, m$. The matrix function $f(A)$ can be approximated via a partial fraction expansion which leads to multiple shifted systems. Another application in which families of shifted systems like (1.1) arise is image processing. Working on more than one image in parallel or separating the colour channels even of a single image results in multiple right-hand sides. Here, the shifts stem from a regularisation and, based on some criterion, one is interested in finding optimal shifts and the corresponding solutions $x_{i,j}$.

Krylov subspace methods are a class of methods that provide means to solve large sparse systems by implicitly or explicitly transforming these systems into smaller dense systems that can be tackled with well-known techniques for dense matrices. For solving $Ax = b$ and for (1.1) in the special cases of shifted systems where $m = 1$ and systems with multiple right-hand sides where $s = 1$ a variety of Krylov subspace methods exists. These can be applied to the systems in (1.1) one by one—either one shift after the other, one right-hand side after the other, or both.

This work explores which of the existing Krylov subspace methods are well-suited for solving (1.1). Further, novel Krylov subspace methods are developed that are targeted especially at multiple shifts and multiple right-hand sides and by this solve (1.1) for all shifts and right-hand sides at the same time. Unlike most of the literature we will not only count the number of matrix-vector multiplications performed by the methods as a measure for comparing different methods since this can be quite deceptive especially in the case of block methods. Instead, we implemented all presented methods and compare actual runtime measurements of the algorithms. Since time measurements alone bring along other issues we additionally analysed the algorithms using a detailed cost model to have a theoretical background for our findings.

We demonstrate numerically that our novel Krylov subspace methods for solving families of linear systems as in (1.1) do not only reduce the number of matrix-vector multiplications significantly but also yield lower computational time than a variety of existing Krylov subspace methods in certain situations. Moreover, we present results that our cost model can be used as support for deciding whether one Krylov subspace method should be preferred over another.

1.1 Outline

This thesis is organised as follows.

The remainder of Chapter 1 is dedicated to introduce the notation and basic definitions used throughout this thesis. Especially, when dealing with iterative methods that involve more than just vectors—for example vectors grouped together in matrices, slices of matrices, and mixtures thereof—a distinct and consistent notation is vital for a good understanding. For this, Section 1.2 introduces the notation used in this work and gives examples where needed. Some definitions that are used at multiple places are collected in Section 1.3.

In order to have this thesis self-contained the chapters 2 and 3 are dedicated to introduce some basics. Chapter 2 explains the concept of iterative methods for solving linear systems in general. Krylov subspace methods as a particular incarnation of iterative methods are described in Section 2.2. There, we state some well known Krylov subspace methods which later algorithms are based upon. In Chapter 3 functions of matrices are defined. Proficient readers might skip these two chapters since they mostly contain basic concepts. However, we have to include their contents to some extent since in later chapters we expand on some of the methods and concepts introduced there and need to have some details as a back-reference.

In Chapter 4 we introduce two categories of applications in which solving (1.1) is needed. The first in Section 4.1 stems from lattice QCD and the second in Section 4.2 from inverse problems in image deconvolution. We define the two terms there and describe in detail how systems like (1.1) arise in these particular applications. Later, we will use these applications for our numerical tests.

In Chapters 5, 6, and 7 we introduce different approaches that can be used to solve systems like (1.1). First, we focus on shifted methods in Chapter 5 which can be applied to each right-hand side separately. In Section 5.1 we describe shifted CG followed by the formulation of restarted shifted CG in Section 5.2. Second, in Chapter 6 we present methods for solving families of linear systems with multiple right-hand sides and most prominently block methods. Section 6.1 introduces seed methods like the Single Seed Method and Seed-CG. With incremental eigCG a sophisticated representative of deflation methods is described in Section 6.2. A whole range of block methods—all related to BlockCG—will be featured in Section 6.3. Lastly, Chapter 7 is devoted to developing methods aiming exactly at (1.1) by handling shifts and multiple right-hand sides at the same time. In Section 7.1 we display Lanczos-type processes that we use to base block Krylov subspace methods on. Two deflated block Krylov subspace methods that go by the names DSBlockCG and BFDSCG are developed in the Sections 7.2 and 7.3, respectively. Finally, another method called SBCGrQ for solving (1.1) is described in Section 7.4.

Since one of our goals is to compare different approaches for solving (1.1) we decided to gather all of our numerical tests and results in one place. This is done in Chapter 8. We describe our implementations and test settings in Section 8.1 and give numerical results for the tests involving all the methods presented in previous chapters in the Sections 8.2, 8.3, and 8.4.

Finally, Chapter 9 is used to draw conclusions and discuss potential future work.

Parts of Chapter 2, 3 and 4 are based on or taken in part from the authors Diploma thesis [Bir08]. Some parts of Chapter 7 related to the DSBlockCG algorithm are meanwhile published in [BF14].

1.2 Notation and Abbreviations

In the following tables the abbreviations, symbols, and notations used throughout this thesis are explained.

Table 1.1: Abbreviations

rhs(s)	right-hand side(s)
mvm(s)	matrix-vector multiplication(s)
mvp(s)	matrix-vector product(s)[1]
mbvm(s)	matrix-block-vector multiplication(s)
hpd	Hermitian positive definite
nnz	number of non-zeros
ev	eigenvector
ew	eigenvalue (see [TB97])
smallest ev	eigenvector belonging to the eigenvalue of smallest magnitude

Table 1.2: Miscellaneous notation

$\mathbb{K}$	Either of the fields of real or complex numbers, i.e. $\mathbb{K} \in \{\mathbb{C}, \mathbb{R}\}$.
$\mathbb{R}^+$	Set of positive real numbers $\mathbb{R}^+ := \{x \in \mathbb{R} : x > 0\}$.
Π_k	Set of polynomials of degree $\leq k$.
$\overline{\Pi}_k$	Set of normalised polynomials of degree $\leq k$, i.e. $p \in \Pi_k$ and $p(0) = 1$.
$\mathcal{R}_{l/m}$	Set of rational functions; numerator degree l and denominator degree m.
δ_{ij}	Kronecker delta $\delta_{ij} := \begin{cases} 1, & \text{if } i = j \text{ and} \\ 0, & \text{if } i \neq j. \end{cases}$
$A \otimes B$	Kronecker product of $A = (a_{i,j}) \in \mathbb{C}^{m\times n}$ and $B \in \mathbb{C}^{p\times q}$ defined as $$A \otimes B = \begin{bmatrix} a_{1,1}B & \cdots & a_{1,n}B \\ \vdots & & \vdots \\ a_{m,1}B & \cdots & a_{m,n}B \end{bmatrix}.$$
$W^\perp$	The orthogonal complement of W, W being a subspace of $\mathbb{K}^n$.
$=$	In algorithms $=$ will be used as an assignment operator.
$\varepsilon_{\text{mach}}$	Machine precision.

[1]We use mvm and mvp synonymously.

Table 1.3: Notation related to matrices, vectors, and indices

$A = [a_{i,j}]$	The matrix A consists of the elements $a_{i,j}$.
$A = [a_1 \mid \dots \mid a_n]$	The matrix A consists of the vectors $a_1, \dots, a_n$ as columns.
$\boldsymbol{A}$	Matrices that are composed of elements that bear a matrix-structure themselves, e.g. a block diagonal matrix would be written as $\boldsymbol{A} = \operatorname{diag}(A_1, A_2)$ where A_1 and A_2 are matrices.
I	Identity matrix with $(I)_{ij} = \delta_{ij}$. If its dimension is not clear by context we use an index $I_n \in \mathbb{C}^{n \times n}$.
$SU(3)$	Special unitary group $\{U \in \mathbb{C}^{3\times 3} : U^H U = I_3, \det(U) = 1\}$.
A^{-H}	Short-hand notation $A^{-H} = (A^{-1})^H = (A^H)^{-1}$.
$A_{2,:}$	We use a colon to refer to slices of matrices, i.e. complete columns or rows, here: the second row of matrix A.
$\operatorname{vec}(A)$	The matrix $A = [a_1 \mid \dots \mid a_n]$ reshaped as a vector, i.e. $\operatorname{vec}(A) = [a_1^T \mid \dots \mid a_n^T]^T$.
$x^{(i)}$	Upper indices on vectors, matrices or scalars in round brackets are iteration indices.
e_i	The i-th unit vector with dimension given by context.
p_i	For polynomials lower indices indicate iteration indices to prevent confusion with the derivative.
$x_{i,j}$	Lower indices on vectors, matrices and scalars indicate entities belonging to a right-hand side, a shift or elements of a matrix or vector depending on context.
$\langle a, b \rangle$	Inner product on $\mathbb{K}^n$ defined as $\langle a, b \rangle := b^H a$.
$\lVert x \rVert_A$	A-norm of the vector x, $\lVert x \rVert_A := \sqrt{\langle Ax, x \rangle}$.
$\lVert A \rVert$	Operator norm of A, $\lVert A \rVert := \sup_{\lVert x \rVert = 1} \lVert Ax \rVert$.
$\kappa(A)$	Condition number of A, $\kappa(A) := \lVert A \rVert \lVert A^{-1} \rVert$. If $\kappa(A)$ is large, A is called ill-conditioned.
$\operatorname{spec}(A)$	Spectrum of matrix A.
$\rho(A)$	Spectral radius of matrix A.
$\operatorname{span}\{\dots\}$	Space of the linear combinations of the vectors or matrices inside the curly brackets.
$\operatorname{colspan}\{\dots\}$	Space of the linear combinations of the columns of the matrix/matrices inside the curly brackets.

1.3 Basic Definitions

Landau notation

As a measure for the approximation quality of a function as well as the time complexity and memory requirements of algorithms we use the *Landau notation* which we define next.

Definition 1.1. *Let $R \subseteq \mathbb{R}$, $a \in \mathbb{R}$ with $\forall \varepsilon : \exists r \in R : |r - a| < \varepsilon$, and $f, g : R \to \mathbb{R}$. We say $f(x) = \mathcal{O}(g(x))$ as $x \to a$, if and only if*

$$\exists c, \varepsilon : \forall x : |x - a| < \varepsilon \Rightarrow |f(x)| \leq c|g(x)|$$

and for $f, g : R \to \mathbb{R}^+$ we say $f(x) = \mathcal{O}(g(x))$, if and only if

$$\exists c, x_0 : \forall x \geq x_0 : f(x) \leq cg(x).$$

As a short-hand notation we often use $f = \mathcal{O}(g)$.

Block vectors

Solving systems like (1.1) for a matrix $A \in \mathbb{C}^{n \times n}$ in an iterative manner involves vectors $v \in \mathbb{C}^n$. Some methods presented in Chapter 6 and Chapter 7, however, group $m \ll n$ vectors as columns of a matrix $V \in \mathbb{C}^{n \times m}$ and perform operations with these. We call such a tall and skinny compound matrix V a *block-vector*.

Cost model for algorithm analysis

In the chapters 5, 6 and 7 we will analyse the computational cost of the algorithms that are presented there. For this we will use a cost model that we introduce here.

The methods presented perform different kinds of operations for solving (1.1), for instance, matrix-vector multiplications (mvms) with the matrix A or vector additions. A common way to compare algorithms is to count the number of mvsm. In some cases, however, this can be quite a deceptive measure, since the cost of other operations might not be negligible. Thus, we will compare the algorithms not only using the number of mvms but also by measuring their runtime. One of our goals will be to find some theoretical background to explain the measured runtime differences of the methods.

We want to base our comparisons on counting the different kinds of operations and get a number—depending on some parameters—that is characteristic for the particular algorithm. The following two definitions introduce the foundation of our cost model.

Definition 1.2. *Let $x, y \in \mathbb{C}^n$ and $\alpha \in \mathbb{C}$. We will call the following types of operations* vector operations *(of size n):*

$$\begin{aligned} y &\leftarrow \alpha x + y && \textit{(axpy)} \\ y &\leftarrow \alpha y && \textit{(scaling)} \\ y &\leftarrow x && \textit{(assigning)} \\ y &\leftrightarrow x && \textit{(swapping)} \\ \alpha &\leftarrow y^H x && \textit{(dot-product)} \end{aligned}$$

We treat all the vector operations appearing in Definition 1.2 as having the same computational cost of n floating point operations. This is reasonable to assume—especially on modern hardware that supports fused multiply-add floating point operations. When we discuss counting operations in methods that involve a matrix $A \in \mathbb{C}^{n\times n}$ we will implicitly only count vector operations of size n.

Definition 1.3. *Let $A \in \mathbb{K}^{n\times n}, v \in \mathbb{K}^n$, and $V \in \mathbb{K}^{n\times m}$. We define c_A to be the cost of computing the matrix-vector product Av in terms of vector operations. Further, $c_A^\square(m)$ is the cost of computing the matrix-block-vector product AV in terms of vector operations divided by m.*

This means that computing Av has the computational cost of nc_A floating point operations.

For block-vectors we assume in some cases that the cost of matrix-block-vector products is about mnc_A floating point operations, although we might anticipate lower costs than that. Lower costs, i.e. $c_A^\square(m) < c_A$ and thus $mnc_A^\square(m) < mnc_A$, might be achieved since operations on block-vectors can result in more cache-friendly memory access patterns and vectorisation depending on the used hardware. We elaborate on this in more detail in the discussion for those methods where block-vectors are relevant. Likewise, for block-vector operations we will discuss the estimated computational cost in the appropriate chapters.

The introduction of c_A and $c_A^\square$ allows us to sum the computational costs of vector operations and matrix-vector products.

Definition 1.4. *With o_{name} we denote the cost of all vector operations and matrix-vector products in units of vector operations to solve a linear system using the particular method "name". We call o_{name} the number of vector(-equivalent) operations or the number of operations for short.*

When discussing the presented algorithms in later chapters we will state the number of operations depending on some parameters according to some assumptions on the algorithm. We will assume that n is much larger than any other parameter for the algorithms. Hence, every operation that depends only on such parameters much smaller than n will be neglected. In a few places one might need to break with this rule to obtain a more suitable cost model, e.g. when $\mathcal{O}(m^3)$ operations are involved. We will point this out and discuss the implications where it is needed. Lastly, we only count the operations in the main loops of the algorithms and assume that the number of iteration steps is large enough so that we can neglect the costs for initialisations.

Cost model example: Householder QR

As an example for the just introduced cost model and for later use we apply our cost model to Algorithm 1.1. This algorithm computes the *Householder QR decomposition* for tall skinny matrices and is a high-level description of [GL96, Algorithm 5.2.1] and [GL96, Section 5.1.6].

Algorithm 1.1: HOUSEHOLDER QR

```
Input  : $A \in \mathbb{C}^{n\times m}$   matrix with $m < n$, modified during the algorithm
Output : QR decomposition $QR = A$ with unitary $Q \in \mathbb{C}^{n\times m}$ and upper triangular $R \in \mathbb{C}^{m\times m}$

  // compute factored form of the decomposition [GL96, Algorithm 5.2.1]
1 for $k = 1, 2, \dots, m$ do
2     compute Householder vector $v$ and scalar $\beta_k$ from $A_{k:n,k}$   // [GL96, Algorithm 5.1.1]
3     apply Householder reflection $I - \beta_k vv^H$ to $A_{k:n,k:m}$
4     if $k < n$ then store $v_{2:m-k+1}$ in $A_{k+1:m,k}$
  // build the matrix $Q$ [GL96, Section 5.1.6]
5 $Q = (I_n)_{:,1:m}$
6 for $k = m, m-1, \dots, 1$ do                 // [GL96, Section 5.1.6]
7     get Householder vector $v$ from $A_{k+1:n,k}$
8     apply Householder reflection $I - \beta_k vv^H$ to $Q_{k:n,k:m}$
  // obtain $R$
9 $R$ can be obtained as the upper triangular part of $A$
```

The exact definition of the vector v and the scalar β_k in the lines 2 and 7 is not relevant for our intended analysis and we refer to [GL96, Algorithm 5.1.1] for details. The only thing we need to know here about [GL96, Algorithm 5.1.1] is that it involves two vector operations—an inner product and a scaling of the resulting vector v. The following proposition states the number of vector operations in Algorithm 1.1. Note that we assume $m \ll n$ such that we can assume vector operations of length n to cost the same as vector operations of length $n-m$.

Proposition 1.5. *Let $A \in \mathbb{C}^{n\times m}$. The total number of vector operations for computing the Householder QR decomposition $QR = A$ with Algorithm 1.1 is*

$$o_{\text{hhqr}} = 2m^2 + 4m. \tag{1.2}$$

Proof. The cost for the loop in line 1 is

$$\sum_{k=1}^{m}(2 + 2(m-k) + 1) = m^2 + 2m$$

which consists of

- 2 vector operations for line 2,
- computing $v^H x$ and an axpy afterwards for $m-k$ vectors in line 3 and
- storing v in A in line 4.

Afterwards, the loop in line 6 computes the matrix Q in a backwards loop as explained in [GL96, Section 5.1.6] and contributes

$$\sum_{i=k}^{m}(1 + 2k) = m^2 + 2m$$

vector operations. This is achieved by exploiting that $I - \beta_k vv^H$ in step k has a block structure $\left[\begin{smallmatrix} I_{k-1,k-1} & 0 \\ 0 & \tilde{Q} \end{smallmatrix}\right]$ where $\tilde{Q}$ is a full $n-k+1 \times m-k+1$ matrix. Therefore, the loop consists of

- building the vector v,
- k dot-products and k axpy.

Overall we end up with (1.2). □

In the subsections 7.1.4 and 7.3.5 we use the *column-pivoting Householder QR decomposition* from [GL96, Algorithm 5.4.1] which yields a factorisation $QR = AP$. We do not want to go into too much detail by stating the algorithm here and only describe the additions it makes to Algorithm 1.1 and summarise its number of vector operations using our cost model. The idea of the column-pivoting Householder QR decomposition is to permute the remaining columns of A so that the column of largest norm is the next one to compute the Householder vector from. The naïve approach would require to compute the norms of all remaining columns in every step of the loop in line 1. But in [GL96, Section 5.4.1] an update method for the norms is presented which lowers the additional costs of the column-pivoting. So the two additions that are relevant to the number of vector operations are a computation of all column-norms of A before the loop in line 1 and swapping two columns of A in every step of the loop. This yields the following proposition.

Proposition 1.6. *Let $A \in \mathbb{C}^{n\times m}$. The total number of vector operations for computing the column pivoting Householder QR decomposition with [GL96, Algorithm 5.4.1] is*

$$o_{\mathrm{hhcpqr}} = 2m^2 + 6m.$$

2 Iterative Methods

This section deals with the problem of solving linear systems of the kind

$$Ax = b, \quad A \in \mathbb{C}^{n\times n}, \quad x, b \in \mathbb{C}^n, \tag{2.1}$$

where A is non-singular. We will introduce the concept of iterative methods briefly. In Section 2.1 we describe basic iterative methods relying on a matrix splitting and resulting in so-called splitting methods. Krylov subspace methods will be covered in Section 2.2 in more detail.

For small n a wide range of methods for solving (2.1) exists. Most of them boil down to computing a matrix decomposition of A into easily invertible factors. Methods of this kind are called *direct methods*. Computing the LU-decomposition of A for example allows to solve $Ax = LUx = b$ in two steps, namely $Ly = b$ and $Ux = y$. Since L is lower triangular and U is upper triangular, solving these two systems is straightforward. More details and methods can be found in [GL96], [Saa03] and [Gre97].

In case of considerably large n on the other hand two problems arise. First, computing dense matrix decompositions has a computational complexity of order $\mathcal{O}(n^3)$ except for special cases. This prevents the naïve matrix decomposition approach from being feasible in this case. Second, in many applications the arising matrices have the special property of being sparse. This means that most entries are plain zeros and the number of non-zeros (nnz) is small in relation to all n^2 entries. A decomposition of such a sparse matrix is not guaranteed to be sparse. On the contrary, special care is required to prevent losing sparseness completely or create too many non-zeros, which would result in a huge increase in storage requirements and computational time. Sparse direct solvers include methods involving a sparse LU-decomposition, hence eluding some of the drawbacks. In these solvers some effort has to be put into computing a reordering resulting in a favourable sparsity pattern of the factors L and U. An overview of methods can be found in [Duf98] and software capable of solving large sparse systems involves UMFPACK [DD95], MUMPS [Ame$^+$99] and SuperLU [Dem$^+$99]. As an alternative *iterative methods* come into play.

In general, iterative methods build a sequence $x^{(0)}, x^{(1)}, \dots, x^{(k)} \in \mathbb{C}^n$ of vectors approximating the solution $x^\star = A^{-1}b$ of equation (2.1). The starting vector $x^{(0)}$ is an initial guess that can be chosen arbitrarily, be given by context or simply set to zero. Two values that are of interest in the iteration processes are the *error* $e^{(k)}$ and the *residual* $r^{(k)}$ defined by

$$\begin{aligned} r^{(k)} &:= b - Ax^{(k)} \\ e^{(k)} &:= x^\star - x^{(k)} \end{aligned}$$

for which

$$r^{(k)} = Ae^{(k)} \tag{2.2}$$

holds.

The aim of iterative methods is to reduce the error as the iteration proceeds. In some methods the error converges to zero whilst in other methods at least in theory with exact arithmetic at some iteration step the error could be exactly zero. But because of round off errors even for the latter methods finding the *exact solution* $x^\star$ is next to impossible or at least impractical in practical computations. Note that round off errors are not unique to iterative methods and direct methods are also affected, albeit in different ways. In many applications, however, a certain accuracy is sufficient, and reducing the error further in iterative methods is a waste of effort. Therefore, at some point of the iteration we want to stop with an approximate solution vector $x^{(k)}$ approximating $x^\star$ well enough for our application. The error $e^{(k)}$, however, is unknown.

One cheap way to remedy the unknown error $e^{(k)}$ is to monitor the residuals $r^{(k)}$. The relation (2.2) then gives a hint that $\|e^{(k)}\| \leq \|A^{-1}\|\|r^{(k)}\|$. Thus, for reasonably well conditioned matrices A the error cannot be too far off and the residuals can be used to stop the iteration. Typically, the iteration is stopped as soon as

$$\omega_1 := \frac{\|b - Ax^{(k)}\|}{\|b\|} < \delta \tag{2.3}$$

for some given δ, and we call $\widehat{x} = x^{(k)}$ the *approximate solution* of (2.1). In some circumstances the criterion

$$\omega_2 := \frac{\|b - Ax^{(k)}\|}{\|A\|\|x^{(k)}\| + \|b\|} < \delta$$

can be a better choice, since in [ADR92] and [ST05] it is pointed out that the criterion of equation (2.3) can be interpreted as assuming all backward error only resides in the right-hand sides. The second criterion takes perturbations in A into account, which in (2.3) could lead to rejecting good approximations $x^{(k)}$ that have a large residual $r^{(k)}$. We note that $\omega_2 \leq \omega_1$.

Alternatively, there are error estimation methods that can compute bounds for the error [GM97; ST05]. These methods need some additional computation, which means that whilst having the benefit of better stopping criteria the algorithm gets more complicated and takes more time than without error estimation. The latter can partly be remedied by either computing error estimates only in a subset of the iteration steps or to settle for a lower accuracy on the error bounds if the estimation method allows for this kind of tuning [Fro+13b].

2.1 Splitting Methods

An important class of methods are the so-called *splitting methods*. Methods of this kind rely on the matrix splitting

$$A = M - N$$

and define the iterative scheme as

$$x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b. \tag{2.4}$$

In order for such a method to be feasible, the systems $My = z$ have to be easy to solve. For defining different kinds of splittings it is useful to write the matrix A as $A = L + D + U$. Here, D is the diagonal of A, L is the lower triangular part of A and U the upper triangular part. The most common methods based on this splitting scheme are displayed in Table 2.1.

Table 2.1: Common splitting methods

splitting	name of the resulting method
$M = D$	Jacobi iteration
$M = D + L$	Gauss-Seidel iteration
$M = \frac{1}{\omega}D + L$	Successive Over-Relaxation (SOR)
$M = I$	Richardson iteration

Now that we have the iterative scheme (2.4) the next question arises naturally: Will an iteration based on (2.4) find the solution $x^\star$ eventually? The following theorem answers this question concerning the convergence of splitting methods.

Theorem 2.1. [GL96, Theorem 10.1.1] *Let $b \in \mathbb{C}^n$ and $A = M - N \in \mathbb{C}^{n\times n}$ non-singular. If M is non-singular and $\rho\left(M^{-1}N\right)$ satisfies the inequality $\rho\left(M^{-1}N\right) < 1$, then the iterates of* (2.4) *converge to $x^\star = A^{-1}b$ for any starting vector $x^{(0)}$.*

Proof. The error $e^{(k+1)} = x^{(k+1)} - x^\star$ in step $k+1$ satisfies

$$\begin{aligned} Me^{(k+1)} = Ne^{(k)} &\Leftrightarrow e^{(k+1)} = M^{-1}Ne^{(k)} \\ &\Leftrightarrow e^{(k+1)} = (M^{-1}N)^{k+1}e^{(0)}. \end{aligned}$$

Then

$$\begin{aligned} \lim_{k\to\infty} x^{(k)} = x^\star \text{ for every vector } x^{(0)} &\Leftrightarrow \lim_{k\to\infty} (M^{-1}N)^k = 0 \\ &\Leftrightarrow \rho\left(M^{-1}N\right) < 1. \end{aligned}$$

Herein, the equivalence $\lim_{k\to\infty} A^k = 0 \Leftrightarrow \rho(A) < 1$ holds for any matrix $A \in \mathbb{C}^{n\times n}$ and can be proven in the following way.

$\Rightarrow$ Suppose $\lim_{k\to\infty}(A)^k = 0$. For an arbitrary eigenvalue λ of A with an associated eigenvector v we have $\lambda^k v = A^k v \to 0$ for $k \to 0$, thus $|\lambda| < 1$.

$\Leftarrow$ Now suppose $\rho(A) < 1$. Since for any $\varepsilon > 0$ a matrix norm $\|\cdot\|$ with the property

$$\rho(A) \leq \|A\| \leq \rho(A) + \varepsilon$$

exists [Gre97, Theorem 1.3.3], there is a matrix norm $\|\cdot\|$ with $\|A\| < 1$. Now we have $\|A^k\| \leq \|A\|^k \to 0$ for $k \to 0$ yielding $\lim_{k\to\infty} A^k = 0$. $\square$

More detailed information on iterative methods and splitting methods in particular can be found in [GL96, §10.1.2ff.].

2.2 Krylov Subspace Methods

As seen in the beginning of this chapter, iterative methods are distinctive in the way they select the next iterate x_k. Krylov subspace methods in particular pick the next vector out of a special subspace. The next definition and theorems provide the needed terminology.

Definition 2.2. *Let $A \in \mathbb{C}^{n\times n}$ and $r \in \mathbb{C}^n$. The* Krylov subspace *of dimension k generated by the matrix A and vector r is defined as*

$$\begin{aligned} \mathcal{K}_k(A, r) :=\ & \operatorname{span}\left\{r, Ar, A^2r, \dots, A^{k-1}r\right\} \\ =\ & \left\{x \in \mathbb{C}^n \,:\, x = p_{k-1}(A)r, p_{k-1} \in \Pi_{k-1}\right\}. \end{aligned} \tag{2.5}$$

An immediate consequence of this definition is that the sequence of Krylov subspaces $\mathcal{K}_1(A, r), \dots, \mathcal{K}_k(A, r)$ is nested and cannot grow arbitrarily. The next lemma substantiates this [Saa03].

Lemma 2.3. *Let $A \in \mathbb{C}^{n\times n}$ and $r \in \mathbb{C}^n$. There exists $g \le n$ such that*

1. $\dim(\mathcal{K}_m(A,r)) = m$ *for* $m = 1, 2, \dots, g$
2. $\mathcal{K}_g(A,r) = \mathcal{K}_{g+1}(A,r) = \dots = \mathcal{K}_k(A,r)$ *for every* $k \ge g$

The number $g := g(A,r)$ is called the grade *of A with respect to r.*

Proof. Definition 2.2 directly implies $\mathcal{K}_m(A,r) \subseteq \mathcal{K}_{m+1}(A,r)$ and

$$\dim(\mathcal{K}_{m+1}(A,r)) \begin{cases} \le \dim(\mathcal{K}_m(A,r)) + 1, \\ \ge \dim(\mathcal{K}_m(A,r)), \\ \le n. \end{cases}$$

Thus, there is a smallest $g \le n$ with the property

$$\mathcal{K}_g(A,r) = \mathcal{K}_{g+1}(A,r).$$

The immediate result is that $A^g r$ is already contained in $\mathcal{K}_g(A,r)$. Therefore, scalars $\alpha_i \in \mathbb{C}$ exist with

$$0 = A^g r - \sum_{i=0}^{g-1} \alpha_i A^i r = p(A)r$$

where $p(t) = t^g - \sum_{i=0}^{g-1} \alpha_i t^i$. For arbitrary $k \ge g$, let $t^k = p_1(t)p(t) + p_2(t)$ and $\deg(p_2(t)) < g$, then

$$A^k r = p_1(A) \underbrace{p(A)r}_{=0} + \underbrace{p_2(A)r}_{\in \mathcal{K}_g(A,r)} .$$

Hence, $\mathcal{K}_k(A,r) = \mathcal{K}_{k-1}(A,r)$ which proves the second property. □

The following theorem provides the theoretical background why using Krylov subspaces for solving the system (2.1) is justified. It states that in exact computation the solution of (2.1) is contained in a Krylov subspace of dimension less than n. Afterwards we can define what we will refer to as a Krylov subspace method.

Theorem 2.4. *Let $A \in \mathbb{C}^{n\times n}$, $x^{(0)} \in \mathbb{C}^n$, $r = b - Ax^{(0)}$ and g be as in Lemma 2.3. If A is non-singular then*

$$x^\star \in x^{(0)} + \mathcal{K}_g(A,r) \text{ and} \tag{2.6}$$

$$x^\star \notin x^{(0)} + \mathcal{K}_{g-1}(A,r), \tag{2.7}$$

where $x^\star = A^{-1}b$ denotes the solution to $Ax = b$.

Proof. Let $p(t) = \sum_{i=0}^{g} \alpha_i t^i$ be a polynomial of smallest degree for which $p(A)r = 0$ holds, i.e.

$$0 = \sum_{i=0}^{g} \alpha_i A^i r. \tag{2.8}$$

Its existence is implied by the proof of Lemma 2.3. First we show that $\alpha_0 \neq 0$. Since A is non-singular, assuming $\alpha_0 = 0$ in (2.8) implies

$$0 = q(A)r = \sum_{i=0}^{g-1} \alpha_{i+1} A^i r$$

which contradicts the minimality of g. Now we can use equation (2.8) to obtain

$$A^{-1}r = -\frac{1}{\alpha_0}\left(\sum_{i=1}^{g} \alpha_i A^{i-1} r\right) \in \mathcal{K}_g(A, r).$$

This results in

$$\begin{aligned} x^\star &= x^{(0)} + (x^\star - x^{(0)}) \\ &= x^{(0)} + A^{-1}r \\ &\in x^{(0)} + \mathcal{K}_g(A, r) \end{aligned}$$

and proves property (2.6) as well as (2.7) considering the minimality of g. □

Definition 2.5. *A* Krylov subspace method *for solving $Ax = b$ is an iterative method that generates iterates $x^{(k)}$ which fulfil*

$$x^{(k)} \in x^{(0)} + \mathcal{K}_k(A, r^{(0)}),$$

where $r^{(0)} = b - Ax^{(0)}$; and $x^{(0)}$ is the starting vector.

Therefore, every Krylov subspace method builds up polynomials $p_{k-1}(t) \in \Pi_{k-1}$ and $q_k(t) = 1 - tp(t) \in \overline{\Pi}_k$ as the iteration proceeds. These polynomials satisfy

$$\begin{aligned} x^{(k)} &= x^{(0)} + p_{k-1}(A) r^{(0)} \quad \text{and} \\ r^{(k)} &= b - Ax^{(k)} \\ &= b - Ax^{(0)} - Ap_{k-1}(A) r^{(0)} \\ &= (I - Ap_{k-1}(A)) r^{(0)} \\ &= q_k(A) r^{(0)}. \end{aligned}$$

Furthermore, every sequence $q_k \in \overline{\Pi}_k, \deg(q_k) = k$, defines a Krylov subspace method as does each $p_{k-1} \in \Pi_{k-1}, \deg(p_{k-1}) = k-1$, according to $q_k(t) = 1 - tp_{k-1}(t)$. Note that the polynomials p_{k-1} and q_k depend on the matrix A and the initial residual $r^{(0)} = b - Ax^{(0)}$. From the vast number of possible Krylov subspace methods some inhere desirable properties. For example, some methods minimise the error or residual in some norm or have short recurrences. Before we describe a few Krylov subspace methods in detail we introduce three procedures to generate bases for Krylov subspaces.

2.2.1 The Arnoldi Process

It is possible, albeit not advisable, to base Krylov subspace methods on the basis $\{r, Ar, A^2r, \dots, A^{k-1}r\}$. This basis has some disadvantages. For instance do the vectors $A^k r$ for $k \to \infty$ tend to point in the direction of the eigenvector belonging to the largest eigenvalue in modulus. Moreover, depending on the operator A the vectors in this basis vary in norm significantly. Consequently, while spanning the space (2.5) an alternative set of basis vectors is computed alongside a matrix describing the change of basis.

The *Arnoldi process* presented in Algorithm 2.1 builds an orthonormal basis for the Krylov subspace $\mathcal{K}_m(A, v^{(1)})$ by applying the *modified Gram-Schmidt process* [GL96].

Algorithm 2.1: Arnoldi Process

```
Input  :  $A \in \mathbb{C}^{n\times n}$     system matrix
          $v^{(1)} \in \mathbb{C}^n$     starting vector
          $m$                number of basis vectors to build
Output:   $\{v^{(1)}, v^{(2)}, \dots, v^{(m)}\}$   orthonormal basis of $\mathcal{K}_m(A, v^{(1)})$

1 $\beta = \|v^{(1)}\|_2$
2 $v^{(1)} = v^{(1)}/\beta$
3 for $k = 1, 2, \dots, m$ do
4     $v = Av^{(k)}$
5     for $i = 1, 2, \dots, k$ do                     // modified Gram-Schmidt
6         $h_{i,k} = \langle v, v^{(i)} \rangle$
7         $v = v - h_{i,k} v^{(i)}$
8     $h_{k+1,k} = \|v\|_2$
9     $v^{(k+1)} = v/h_{k+1,k}$
```

The vectors $v^{(1)}, \dots, v^{(k)}$ generated by Algorithm 2.1 form an n-by-k matrix $V^{(k)} = [v^{(1)}| \dots |v^{(k)}]$ with orthonormal columns and the scalars h_{ij} form a

k-by-k upper Hessenberg matrix

$$H^{(k)} = \begin{bmatrix} h_{1,1} & h_{1,2} & \cdots & h_{1,k} \\ h_{2,1} & h_{2,2} & \ddots & h_{2,k} \\ & \ddots & \ddots & \vdots \\ & & h_{k,k-1} & h_{k,k} \end{bmatrix}.$$

The resulting matrices of the Arnoldi process satisfy the *Arnoldi relation*

$$AV^{(k)} = V^{(k)}H^{(k)} + h_{k+1,k}v^{(k+1)}e_k^H = V^{(k+1)}H^{(k+1,k)}. \tag{2.9}$$

The vector e_k is the k-th unit k-vector and $H^{(k+1,k)}$ denotes the matrix whose top k rows are those of the matrix $H^{(k)}$ and the last row consists of zeros except for the last entry, which is $h_{k+1,k}$, i.e.

$$H^{(k+1,k)} = \begin{bmatrix} h_{1,1} & h_{1,2} & \cdots & h_{1,k} \\ h_{2,1} & h_{2,2} & \ddots & h_{2,k} \\ & \ddots & \ddots & \vdots \\ & & h_{k,k-1} & h_{k,k} \\ & & & h_{k+1,k} \end{bmatrix}.$$

A discussion of a potential breakdown that happens when the vector $v^{(k)}$ vanishes at a step $k < m \leq n$ can be found in [Saa03, Chapter 6].

We will need the following definition on several occasions.

Definition 2.6. *Let $V^{(k)}$ and $H^{(k)}$ be the matrices generated by the Arnoldi process for the matrix A after k steps. Further, let γ be an eigenvalue of $H^{(k)}$ and y an eigenvector belonging to γ. Then γ is called a* Ritz-value*, $V^{(k)}y$ is called a* Ritz-vector *and we call $(\gamma, V^{(k)}y)$ a* Ritz-pair *of A w.r.t. $V^{(k)}$.*

2.2.2 The Lanczos Process

The following theorem reveals that if the matrix A is Hermitian then this property carries over to $H^{(k)}$. Making use of this property allows for a simplification of the Arnoldi process of Algorithm 2.1.

Theorem 2.7. [Saa03, Theorem 6.2] *Assume that the Arnoldi process of Algorithm 2.1 is applied to a Hermitian matrix A. Then the coefficients $h_{i,j}$ generated by the algorithm are such that*

$$\begin{aligned} h_{i,j} &= 0, && \text{for } |i-j| > 1, \\ h_{j,j+1} &= h_{j+1,j}, && \text{for } j = 1, 2, \dots, n. \end{aligned} \tag{2.10}$$

Proof. From equation (2.9)

$$\begin{aligned} (V^{(k)})^H A V^{(k)} &= (V^{(k)})^H V^{(k+1)} H^{(k+1,k)} \\ &= \begin{bmatrix} I_k & \begin{matrix} 0 \\ \vdots \\ 0 \end{matrix} \end{bmatrix} H^{(k+1,k)} \\ &= H^{(k)} \end{aligned}$$

can be obtained. Thus, if A is Hermitian so is $H^{(k)}$. Combining $H^{(k)} = (H^{(k)})^H$ with the fact that $H^{(k)}$ is upper Hessenberg thus implies that $H^{(k)}$ is tridiagonal and therefore (2.10) holds. $\square$

The Arnoldi process for Hermitian A goes under the name *Lanczos process* and is given in Algorithm 2.2.

Algorithm 2.2: Lanczos Process

```
Input  :  $A \in \mathbb{C}^{n\times n}$, hpd    system matrix
          $v^{(1)} \in \mathbb{C}^n$           starting vector
          $m$                   number of basis vectors to build

Output:   $\{v^{(1)}, v^{(2)}, \dots, v^{(m)}\}$   orthonormal basis of $\mathcal{K}_m(A, v^{(1)})$

1 $\beta^{(0)} = \|v^{(1)}\|_2$
2 $v^{(1)} = v^{(1)}/\beta^{(0)}$
3 $v^{(0)} = 0$
4 for $k = 1, 2, \dots, m$ do
5     $v = Av^{(k)} - \beta^{(k-1)}v^{(k-1)}$
6     $\alpha^{(k)} = \langle v, v^{(k)} \rangle$
7     $v = v - \alpha^{(k)}v^{(k)}$
8     $\beta^{(k)} = \|v\|_2$
9     $v^{(k+1)} = v/\beta^{(k)}$
```

Algorithm 2.2 does not need all previous vectors $v^{(1)}, \dots, v^{(k-1)}$ to compute $v^{(k)}$. This is a substantial improvement over the Arnoldi process—both in computational time and in storage requirements. The matrices of the Lanczos process in Algorithm 2.2 fulfil the *Lanczos relation*

$$AV^{(k)} = V^{(k)}T^{(k)} + \beta^{(k)}v^{(k+1)}(e_k)^H = V^{(k+1)}T^{(k+1,k)} \tag{2.11}$$

where

$$T^{(k)} = \begin{bmatrix} \alpha^{(1)} & \beta^{(1)} & & & \\ \beta^{(1)} & \alpha^{(2)} & \beta^{(2)} & & \\ & \beta^{(2)} & \ddots & \ddots & \\ & & \ddots & \ddots & \beta^{(k-1)} \\ & & & \beta^{(k-1)} & \alpha^{(k)} \end{bmatrix} \text{ and } T^{(k+1,k)} = \begin{bmatrix} T^{(k)} \\ 0\cdots 0\ \beta^{(k)} \end{bmatrix}.$$

As for the Arnoldi process we refer to [Saa03] for a discussion of potential breakdowns.

2.2.3 The Two-Sided Lanczos Process

It is possible to transport the idea of having short recurrences like in the Lanczos process back to the non-Hermitian case. This comes at the expense of building two bi-orthogonal bases instead of one orthogonal basis, though. The resulting process using bi-orthogonalisation is called the *two-sided Lanczos process* and is displayed in Algorithm 2.3.

Algorithm 2.3: Two-Sided Lanczos Process

Input :	$A \in \mathbb{C}^{n\times n}$, hpd	system matrix
	$v^{(1)} \in \mathbb{C}^n, w^{(1)} \in \mathbb{C}^n$	starting vectors
	m	number of basis vectors to build
Output:	$v^{(i)}, w^{(i)}$ for $1 \le i \le m$	bi-orthogonal bases of the Krylov subspaces $\mathcal{K}_m(A, v^{(1)})$ and $\mathcal{K}_m(A^H, w^{(1)})$

```
1  $v^{(0)} = w^{(0)} = 0$, $\beta^{(0)} = \gamma^{(0)} = 0$
2  $v^{(1)} = v^{(1)} / \|v^{(1)}\|$, $w^{(1)} = w^{(1)} / \|w^{(1)}\|$
3  for $k = 1, 2, \dots, m$ do
4      $v = Av^{(k)}$
5      $w = A^H w^{(k)}$
6      $\alpha^{(k)} = \langle v, w^{(k)} \rangle$
7      $v = v - \alpha^{(k)} v^{(k)} - \beta^{(k-1)} v^{(k-1)}$
8      $w = w - \bar{\alpha}^{(k)} w^{(k)} - \gamma^{(k-1)} w^{(k-1)}$
9      $\gamma^{(k)} = \|v\|_2$
10     $v^{(k+1)} = v / \gamma^{(k)}$
11     $\beta^{(k)} = \langle v^{(k+1)}, w \rangle$
12     $w^{(k+1)} = w / \bar{\beta}^{(k)}$
```

In Algorithm 2.3 we chose to normalise the vector $v^{(i)}$ and to scale $w^{(i)}$, since we need to guarantee

$$\langle v^{(i)}, w^{(i)} \rangle = 1. \tag{2.12}$$

Other scalings of $v^{(i)}$ and $w^{(i)}$ in the algorithm are possible as long as (2.12) holds. The vectors and scalars computed by the two-sided Lanczos process satisfy a similar relation as in the Hermitian case. Written in matrix form the relations

$$\begin{aligned} AV^{(k)} &= V^{(k)}T^{(k)} + \gamma^{(k)}v^{(k+1)}(e_k)^H = V^{(k+1)}T^{(k+1,k)} \\ A^H W^{(k)} &= W^{(k)}(T^{(k)})^H + \bar{\beta}^{(k)}w^{(k+1)}(e_k)^H = W^{(k+1)}\hat{T}^{(k+1,k)} \end{aligned} \tag{2.13}$$

hold with

$$\begin{aligned}
V^{(k)} &= [v^{(1)}|\dots|v^{(k)}], W^{(k)} = [w^{(1)}|\dots|w^{(k)}],\\
T^{(k)} &= \begin{bmatrix} \alpha^{(1)} & \beta^{(1)} & & & \\ \gamma^{(1)} & \alpha^{(2)} & \beta^{(2)} & & \\ & \gamma^{(2)} & \ddots & \ddots & \\ & & \ddots & \ddots & \beta^{(k-1)} \\ & & & \gamma^{(k-1)} & \alpha^{(k)} \end{bmatrix},\\
T^{(k+1,k)} &= \begin{bmatrix} T^{(k)} \\ 0 \cdots 0\ \gamma^{(k)} \end{bmatrix} \text{ and } \hat{T}^{(k+1,k)} = \begin{bmatrix} (T^{(k)})^H \\ 0 \cdots 0\ \bar{\beta}^{(k)} \end{bmatrix}.
\end{aligned}$$

A discussion of potential breakdowns can be found in [Saa03, Chapter 7].

2.2.4 Conjugate Gradients

The Arnoldi and Lanczos processes pave the way for introducing several Krylov subspace methods. The generated iterates of Krylov subspace methods are vectors from the affine subspace $x^{(0)} + \mathcal{K}_k(A, r^{(0)})$. Using the basis vectors $V^{(k)} = [v^{(1)}|\dots|v^{(k)}]$ created by the Arnoldi or Lanczos process we can write the iterates as

$$x^{(k)} = x^{(0)} + V^{(k)}y^{(k)} \tag{2.14}$$

with $y^{(k)} \in \mathbb{C}^k$. This implicitly builds the polynomials mentioned above and therefore the choice of $y^{(k)}$ defines the Krylov subspace method. In this and the following two subsections, three Krylov subspace methods—namely Conjugate Gradients, GMRES and QMR—in their basic version are introduced and some of their properties are discussed. More details and other Krylov subspace method can be found in [Gre97] and [Saa03]. The remainder of this chapter is mostly based on these two books.

Let $b \in \mathbb{C}^n$ and $A \in \mathbb{C}^{n\times n}$ Hermitian positive definite. The method of *Conjugate Gradients* (CG) proposed in [HS52] can solve $Ax = b$ having the handy property of short recurrences. The CG method can be derived as a Krylov subspace method by imposing the *Ritz-Galerkin condition*

$$r^{(k)} \perp \mathcal{K}_k(A, r^{(0)}), \tag{2.15}$$

i.e. $(V^{(k)})^H A e^{(k)} = (V^{(k)})^H r^{(k)} = 0$, which implies a minimisation of the A-norm of the error $e^{(k)} = e^{(0)} - V^{(k)}y^{(k)}$ (cf. [SW93, Theorem 14.2.4]). Therefore, $y^{(k)}$ satisfies

$$(V^{(k)})^H(r^{(0)} - AV^{(k)}y^{(k)}) = \left\|r^{(0)}\right\|_2 e_1 - T^{(k)}y^{(k)} = 0,$$

where e_1 is the first unit k-vector and $T^{(k)} \in \mathbb{R}^{k\times k}$ is defined in equation (2.11). The task now is to find a $y^{(k)}$ that solves this equation in an efficient way. We will present most of the technical details of the derivation of the CG method, because some of the methods in the chapters 5, 6 and 7 are based on CG and expand the algorithm in some way.

Since $T^{(k)} = (V^{(k)})^H A V^{(k)}$, the matrix $T^{(k)}$ is positive definite and the *root-free Cholesky decomposition* of $T^{(k)}$ exists. Then $T^{(k)}$ can be written as a product

$$T^{(k)} = L^{(k)} D^{(k)} (L^{(k)})^H,$$

where $L^{(k)}$ is lower bidiagonal and $D^{(k)} = \operatorname{diag}\left(d^{(1)}, \dots, d^{(k)}\right)$. In order to compute iterates $x^{(k)}$ without having to save the whole matrix $V^{(k)}$, an auxiliary matrix

$$P^{(k)} := V^{(k)} (L^{(k)})^{-H}$$

is needed. The *search direction* vectors building up $P^{(k)} = [p^{(0)} | p^{(1)} | \dots | p^{(k-1)}]$ are A-orthogonal and can be updated by

$$p^{(k)} = v^{(k)} - \bar{\tilde{\nu}}^{(k-1)} p^{(k-1)}$$

where $\tilde{\nu}^{(k-1)} = (L^{(k)})_{k,k-1}$ denotes the entry of $L^{(k)}$ at the position $(k, k-1)$. Now, the iterate $x^{(k)}$ can be represented as an update of $x^{(k-1)}$ using

$$\begin{aligned} x^{(k)} &= x^{(0)} + V^{(k)} (T^{(k)})^{-1} \left\| r^{(0)} \right\|_2 e_1 \\ &= x^{(0)} + P^{(k)} (D^{(k)})^{-1} (L^{(k)})^{-1} \left\| r^{(0)} \right\|_2 e_1 \\ &= x^{(0)} + P^{(k-1)} (D^{(k-1)})^{-1} (L^{(k-1)})^{-1} \left\| r^{(0)} \right\|_2 e_1 + \tilde{\mu}^{(k-1)} p^{(k-1)} \\ &= x^{(k-1)} + \tilde{\mu}^{(k-1)} p^{(k-1)}, \end{aligned}$$

where $\tilde{\mu}^{(k-1)} = (d^{(k)})^{-1} \left\| r^{(0)} \right\|_2 ((L^{(k)})^{-1})_{k,1}$. Having the update formulation for $x^{(k)}$, the according one for $r^{(k)}$ is easily obtained as

$$\begin{aligned} r^{(k)} &= b - A x^{(k)} \\ &= b - A x^{(k-1)} - \tilde{\mu}^{(k-1)} A p^{(k-1)} \\ &= r^{(k-1)} - \tilde{\mu}^{(k-1)} A p^{(k-1)}. \end{aligned}$$

We still need to eliminate the explicit dependence on the Lanczos vector $v^{(k)}$ in the recurrence for $p^{(k)}$. Since we know from (2.15) that $v^{(k)}$ and $r^{(k)}$ are collinear we can use the recurrence for $r^{(k)}$ instead of computing $v^{(k)}$. This finally leads to the widely known coupled two-term recurrence version of the Conjugate Gradient algorithm displayed in Algorithm 2.4. The scalars $\mu^{(k-1)}$ and $\nu^{(k-1)}$ in the algorithm can be computed without the explicit dependency on the Cholesky decomposition (cf. [Dem97, Chapter 6.6.3]). A comparison with three-term formulations can be found in [GS00]. There, the authors show

Algorithm 2.4: Conjugate Gradients

```
Input  :  $A \in \mathbb{C}^{n\times n}$, hpd          system matrix
          $b \in \mathbb{C}^n$, $x^{(0)} \in \mathbb{C}^n$   right-hand side and initial guess

Output:   approximate solution $x^{(k)}$ to $Ax = b$

1 $r^{(0)} = b - Ax^{(0)}$
2 $p^{(0)} = r^{(0)}$
3 for $k = 1, 2, \ldots$ until convergence do
4     $z^{(k-1)} = Ap^{(k-1)}$
5     $\mu^{(k-1)} = \frac{\langle r^{(k-1)}, r^{(k-1)}\rangle}{\langle p^{(k-1)}, z^{(k-1)}\rangle}$
6     $x^{(k)} = x^{(k-1)} + \mu^{(k-1)}p^{(k-1)}$
7     $r^{(k)} = r^{(k-1)} - \mu^{(k-1)}z^{(k-1)}$
8     $\nu^{(k-1)} = \frac{\langle r^{(k)}, r^{(k)}\rangle}{\langle r^{(k-1)}, r^{(k-1)}\rangle}$
9     $p^{(k)} = r^{(k)} + \nu^{(k-1)}p^{(k-1)}$
```

that coupled two-term recurrence formulations of a specific form show superior numerical stability.

In later chapters we need to compare the computational cost of Algorithm 2.4 to other methods. For this we need the following result.

Proposition 2.8. *Let k_s be the number of iteration steps of Algorithm 2.4 to solve $Ax = b$. Then the number of operations of Algorithm 2.4 according to the cost model from Definition 1.4 is*

$$o_{\mathrm{cg}} = k_s(c_A + 5). \tag{2.16}$$

Proof. Besides the matrix-vector product in line 4 we have three axpy operations for computing $x^{(k)}, r^{(k)}$ and $p^{(k)}$, one inner product $\left\langle p^{(k-1)}, z^{(k-1)}\right\rangle$ and one inner product $\left\langle r^{(k)}, r^{(k)}\right\rangle$. The latter can be computed once and be reused in two more locations. This sums up to $c_A + 5$ vector operations. □

We will finish the discussion of the CG algorithm with an analysis of its convergence behaviour. The A-norm of the error $e^{(k)}$ in step $k > 0$ of the CG algorithm is minimised over the whole space [Gre97, Theorem 2.3.2]

$$e^{(0)} + \operatorname{span}\left\{Ae^{(0)}, A^2e^{(0)}, \ldots, A^ke^{(0)}\right\} = e^{(0)} + \operatorname{span}\left\{p^{(0)}, \ldots, p^{(k-1)}\right\}, \tag{2.17}$$

therefore the representation

$$e^{(k)} = \tilde{p}_k(A)e^{(0)}, \quad \tilde{p}_k \in \overline{\Pi}_k,$$

holds for $\tilde{p}_k$ defined by

$$\left\|e^{(k)}\right\|_A = \left\|\tilde{p}_k(A)e^{(0)}\right\|_A = \min_{p_k \in \overline{\Pi}_k} \left\|p_k(A)e^{(0)}\right\|_A. \tag{2.18}$$

Writing the positive definite matrix A as $A = U\Lambda U^H$, where U is a unitary matrix and $\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$, $A^{1/2}$ can be defined by $A^{1/2} = U\Lambda^{1/2}U^H$, see Theorem 3.5. Then $\|v\|_A = \left\|A^{1/2}v\right\|_2$ and equation (2.18) implies

$$\begin{aligned}
\left\|e^{(k)}\right\|_A &= \min_{p^k \in \overline{\Pi}_k} \left\|A^{1/2}p_k(A)e^{(0)}\right\|_2 \\
&= \min_{p^k \in \overline{\Pi}_k} \left\|Up_k(\Lambda)U^H A^{1/2}e^{(0)}\right\|_2 \\
&\le \min_{p^k \in \overline{\Pi}_k} \left\|Up_k(\Lambda)U^H\right\|_2 \left\|e^{(0)}\right\|_A \\
&= \min_{p^k \in \overline{\Pi}_k} \left\|p_k(\Lambda)\right\|_2 \left\|e^{(0)}\right\|_A.
\end{aligned} \tag{2.19}$$

Since $p_k(\Lambda) = \operatorname{diag}(p_k(\lambda_1), \dots, p_k(\lambda_n))$, relation (2.19) can be written as

$$\frac{\left\|e^{(k)}\right\|_A}{\left\|e^{(0)}\right\|_A} \le \min_{p_k \in \overline{\Pi}_k} \max_{1 \le i \le n} |p_k(\lambda_i)| \tag{2.20}$$

which leads to the next theorem. But first we need to define the Chebyshev polynomials which we will need to retrieve a more convenient bound for (2.20).

Definition 2.9. *The* Chebyshev polynomials $T_k(z)$ *on the interval* $[-1, 1]$ *are defined recursively by*

$$\begin{aligned}
T_0(z) &= 1, \\
T_1(z) &= z, \\
T_{k+1}(z) &= 2zT_k(z) - T_{k-1}(z), \quad k = 1, 2, \dots
\end{aligned}$$

Theorem 2.10. [Gre97, Theorem 3.1.1] *Let $e^{(k)}$ be the error at step k of the CG algorithm applied to the Hermitian positive definite linear system $Ax = b$. Then*

$$\frac{\left\|e^{(k)}\right\|_A}{\left\|e^{(0)}\right\|_A} \le \frac{2}{\gamma^k + \gamma^{-k}} \le 2\gamma^k, \tag{2.21}$$

where

$$\gamma = \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}\right)$$

and $\kappa = \lambda_{max}/\lambda_{min}$ is the condition number of A.

Proof. The transformation

$$z \to \frac{2z - \lambda_{max} - \lambda_{min}}{\lambda_{max} - \lambda_{min}}$$

maps the interval $[\lambda_{min}, \lambda_{max}]$ onto $[-1, 1]$. The shifted and scaled Chebyshev polynomial for the interval $[\lambda_{min}, \lambda_{max}]$ is given by

$$\tilde{p}_k(z) = \frac{T_k\left(\frac{2z-\lambda_{max}-\lambda_{min}}{\lambda_{max}-\lambda_{min}}\right)}{T_k\left(\frac{-\lambda_{max}-\lambda_{min}}{\lambda_{max}-\lambda_{min}}\right)}, \tag{2.22}$$

which is defined since $0 \neq T_k\left(\frac{-\lambda_{max}-\lambda_{min}}{\lambda_{max}-\lambda_{min}}\right)$, and it fulfils $\tilde{p}_k(0) = 1$. The Chebyshev polynomial $\tilde{p}_k$ can now be used to bound

$$\min_{p_k \in \overline{\Pi}_k} \max_{1 \le i \le n} |p_k(\lambda_i)| \le \max_{t \in [\lambda_{min}, \lambda_{max}]} |\tilde{p}_k(t)| .$$

Since $T_k(z)$ can be represented as (cf. [SW93])

$$T_k(z) = \begin{cases} \cos(k \cos^{-1}(z)) & z \in [-1, 1] \\ \cosh(k \cosh^{-1}(z)) & z \notin [-1, 1], \end{cases}$$

the numerator in (2.22) is bounded by 1 for $z \in [\lambda_{min}, \lambda_{max}]$. If $z < -1$, which is true for the parameter $\frac{-\lambda_{max}-\lambda_{min}}{\lambda_{max}-\lambda_{min}}$ in the denominator of (2.22), then z can be written in the form $z = -\cosh(\ln(y)) = -\frac{1}{2}(y + y^{-1})$. This yields the relation $T_k(z) = \frac{1}{2}(y^k + y^{-k})$. Now y in $z = -\frac{1}{2}(y + y^{-1})$ can be computed for $z = \frac{-\lambda_{max}-\lambda_{min}}{\lambda_{max}-\lambda_{min}}$ in the following way

$$\begin{aligned} &\frac{1}{2}(y + y^{-1}) = \frac{\kappa + 1}{\kappa - 1} = \frac{\lambda_{max} + \lambda_{min}}{\lambda_{max} - \lambda_{min}} \\ \Leftrightarrow\ &\frac{1}{2}y^2 - \frac{\kappa + 1}{\kappa - 1}y + \frac{1}{2} = 0. \end{aligned}$$

The latter equation has solutions

$$\begin{aligned} y_{1,2} &= \frac{\kappa + 1}{\kappa - 1} \pm \sqrt{\frac{(\kappa + 1)^2 - (\kappa - 1)^2}{(\kappa - 1)^2}} \\ &= \frac{\kappa + 1}{\kappa - 1} \pm \frac{2\sqrt{\kappa}}{\kappa - 1} \end{aligned}$$

that can be transformed to

$$y_1 = -\frac{\sqrt{\kappa} + 1}{\sqrt{\kappa} - 1} \quad \text{and} \quad y_2 = -\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1}.$$

Both solutions imply

$$\frac{1}{2}(y^k + y^{-k}) = \frac{1}{2}\left(\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k + \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k\right)$$

which proves the first inequality in (2.21). The second inequality follows immediately from

$$\left(\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k + \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k\right)^{-1} \le \left(\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^k\right)^{-1}. \qquad \square$$

2.2.5 GMRES

The CG method relied on A being Hermitian positive definite. In the case of arbitrary non-singular A different methods have to be applied. One such method is the *Generalized Minimal Residual method* (GMRES) proposed in [SS86]. It is based on the Arnoldi process of Algorithm 2.1. According to (2.14) the iterates can be written as

$$x^{(k)} = x^{(0)} + V^{(k)}y^{(k)}.$$

The vectors $y^{(k)}$ in GMRES are computed to minimise the 2-norm of the residual $r^{(k)}$ over the space

$$r^{(0)} + A\mathcal{K}_k(A, r^{(0)}).$$

Using equation (2.9) and $\beta = \left\|r^{(0)}\right\|_2$ we can represent the residual as

$$\begin{aligned} r^{(k)} = b - Ax^{(k)} &= b - A(x^{(0)} + V^{(k)}y^{(k)}) \\ &= r^{(0)} - AV^{(k)}y^{(k)} \\ &= \beta v^{(1)} - V^{(k+1)}H^{(k+1,k)}y^{(k)} \\ &= V^{(k+1)}(\beta e_1 - H^{(k+1,k)}y^{(k)}). \end{aligned}$$

Since $V^{(k+1)}$ has orthonormal columns, the residual norm is

$$\left\|r^{(k)}\right\|_2 = \left\|\beta e_1 - H^{(k+1,k)}y^{(k)}\right\|_2.$$

For minimising $\left\|r^{(k)}\right\|_2$ the GMRES method therefore solves the least squares problem

$$y^{(k)} = \underset{y}{\operatorname{argmin}} \left\|\beta e_1 - H^{(k+1,k)}y\right\|_2. \tag{2.23}$$

This is done by computing a *QR decomposition* of $H^{(k+1,k)}$. The upper Hessenberg structure of $H^{(k+1,k)}$ can be exploited for computing this decomposition efficiently. Using *Givens rotations* the non-zero entries below the diagonal in $H^{(k+1,k)}$ can be eliminated one by one. With a sequence $G^{(1)}, \dots, G^{(k)}$ of Givens rotations we obtain an upper triangular $(k+1 \times k)$-matrix $R^{(k)}$ as

$$R^{(k)} = G^{(k)} G^{(k-1)} \cdots G^{(1)} H^{(k+1,k)}.$$

Herein, $G^{(i)}$ denotes the $(k+1 \times k+1)$-matrix defining the Givens rotation that eliminates the $(i+1,i)$-entry in $G^{(i-1)} G^{(i-2)} \cdots G^{(1)} H^{(k+1,k)}$, i.e.

$$G^{(i)} := \begin{bmatrix} I & & & \\ & c^{(i)} & s^{(i)} & \\ & -\bar{s}^{(i)} & c^{(i)} & \\ & & & I \end{bmatrix} \begin{array}{l} \\ \leftarrow \quad i\text{-th row} \\ \leftarrow \ (i{+}1)\text{-st row} \\ \\ \end{array}$$

with

$$c^{(i)} = \begin{cases} 0 & , \text{if } \eta = 0 \\ \frac{|\eta|}{\sqrt{|\eta|^2 + |\theta|^2}} & , \text{else} \end{cases}, \qquad s^{(i)} = \begin{cases} 1 & , \text{if } \eta = 0 \\ \frac{\eta}{|\eta|} \cdot \frac{\bar{\theta}}{\sqrt{|\eta|^2 + |\theta|^2}} & , \text{else} \end{cases}$$

and

$$\begin{aligned} \eta &= e_i^T \left[G^{(i-1)} G^{(i-2)} \cdots G^{(1)} H^{(k+1,k)} \right] e_i, \\ \theta &= e_{i+1}^T \left[G^{(i-1)} G^{(i-2)} \cdots G^{(1)} H^{(k+1,k)} \right] e_i. \end{aligned}$$

Therefore, with

$$Q^{(k)} := (G^{(1)})^H (G^{(2)})^H \cdots (G^{(k)})^H$$

we get

$$Q^{(k)} R^{(k)} = H^{(k+1,k)}.$$

Now (2.23) can be solved using

$$(\beta Q^{(k)} e_1)_{1:k} = (R^{(k)} y^{(k)})_{1:k}$$

which yields $y^{(k)}$ and consequently the next iterate $x^{(k)}$. Moreover, only the $(k+1)$-entry in the vector $\beta Q^{(k)} e_1 - R^{(k)} y^{(k)}$ is non-zero and represents the norm of the residual since

$$\begin{aligned} \|r^{(k)}\| &= \|\beta Q^{(k)} e_1 - R^{(k)} y^{(k)}\| \\ &= |(s^{(k)})_{k+1}|, \end{aligned}$$

where $s^{(k)} := \beta Q^{(k)} e_1$. Finally, we obtain the GMRES method as Algorithm 2.5.

Since the whole matrix $V^{(k)}$ has to be stored for building $x^{(k)}$, the memory footprint and computational cost for orthogonalisation might grow too large

Algorithm 2.5: GMRES

```
Input  :  $A \in \mathbb{C}^{n\times n}$                  system matrix
          $b \in \mathbb{C}^n$, $x^{(0)} \in \mathbb{C}^n$   right-hand side and initial guess
Output:   approximate solution $x^{(k)}$ to $Ax = b$

1  $r^{(0)} = b - Ax^{(0)}$
2  $\beta^{(0)} = \|r^{(0)}\|_2$
3  $v^{(1)} = r^{(0)}/\beta^{(0)}$
4  $s^{(0)} = \beta^{(0)} e_1$
5  for $k = 1, 2, \dots$ until convergence do
6      compute $v^{(k+1)}$ and $H^{(k+1,k)}$                  // Arnoldi process (Alg. 2.1)
7      apply $G^{(k-1)} \cdots G^{(1)}$ to the last column of $H^{(k+1,k)}$ yielding $\widetilde{R}^{(k)}$
8      compute the Givens rotation $G^{(k)}$ using $\widetilde{R}^{(k)}$
9      apply $G^{(k)}$ to the result $\widetilde{R}^{(k)}$ of line 7 yielding $R^{(k)}$
10     apply $G^{(k)}$ to the $k$-th and $(k+1)$-st entry of $s^{(k-1)}$ yielding $s^{(k)}$
11     $\beta^{(k)} = |(s^{(k)})_{k+1}|$
12 solve $(s^{(k)})_{1:k} = (R^{(k)} y^{(k)})_{1:k}$ for $y^{(k)}$
13 set $x^{(k)} = x^{(0)} + V^{(k)} y^{(k)}$
```

for a high number of iteration steps. To prevent these problems, GMRES can be restarted after m steps, which is denoted as GMRES(m). The idea then is to solve the system $Ad = r^{(m)} = b - Ax^{(m)}$ and update the solution $x^{(m)}$ to $x^{(m)} + d$ afterwards. In [Saa03] the GMRES method and its restarted modifications are discussed in detail. Furthermore, a GMRES variant can be based on an incomplete orthogonalisation by truncating the orthogonalisation in the Arnoldi process. The resulting algorithm is called Quasi-GMRES (QGMRES) [Saa03].

Similarly to the Conjugate Gradients method where we obtained a bound on the A-norm of the error, the convergence behaviour of GMRES can be described by a bound on the 2-norm of the residuals which is the $A^H A$-norm of the error since $\langle r, r\rangle = \langle A^H A e, e\rangle$. Assume that A is diagonalisable and $A = V\Lambda V^{-1}$ is an eigendecomposition of A with $\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$. Then with

$$\begin{aligned}\|r^{(k)}\| &= \min_{p_k \in \overline{\Pi}_k} \|V p_k(\Lambda) V^{-1} r^{(0)}\| \\ &\le \|V\| \cdot \|V^{-1}\| \cdot \|r^{(0)}\| \min_{p_k \in \overline{\Pi}_k} \|p_k(\Lambda)\|\end{aligned}$$

the residuals of GMRES satisfy the equation

$$\frac{\|r^{(k)}\|}{\|r^{(0)}\|} \le \|V\| \cdot \|V^{-1}\| \min_{p_k \in \overline{\Pi}_k} \max_{i=1,\dots,n} |p_k(\lambda_i)|.$$

However, as discussed in [Gre97] and shown in [GPS96] any non-increasing convergence curve is possible as a plot of the residual norm against the iteration number—regardless of the eigenvalue distribution.

2.2.6 QMR

As the Generalized Minimal Residual method was based on the Arnoldi process it inherited long recurrences and the need for storing the whole matrix $V^{(k)}$. With the Quasi-Minimal Residual method we now introduce a method proposed in [FN91] for non-Hermitian A that uses the two-sided Lanczos process to achieve short recurrences. Again, we write the iterates as $x^{(k)} = x^{(0)} + V^{(k)} y^{(k)}$. This implies that the residuals fulfil

$$r^{(k)} = V^{(k+1)}(\beta e_1 - T^{(k+1,k)} y^{(k)}).$$

But the two-sided Lanczos process does not build an orthonormal basis $V^{(k)}$. Thus, we cannot minimise the norm of the residual just using $T^{(k+1,k)}$. But we can still go ahead and minimise the right-hand side of the inequality

$$\left\|r^{(k)}\right\| \leq \left\|V^{(k+1)}\right\| \cdot \left\|\beta e_1 - T^{(k+1,k)} y^{(k)}\right\| \tag{2.24}$$

by minimising only the second factor. This is exactly what the *Quasi-Minimal Residual method* (QMR) does. Similarly to the GMRES method, in the QMR method a QR decomposition using Givens rotations is computed to transform $T^{(k+1,k)}$ to upper triangular form with

$$\begin{aligned} T^{(k+1,k)} &= Q^{(k)} R^{(k)} \\ &= (G^{(1)})^H \cdots (G^{(k)})^H R^{(k)}. \end{aligned}$$

Moreover, we obtain the $(k+1) \times k$ upper triangular matrix $R^{(k)}$ to have non-zeros only on the main diagonal and the first and second diagonal above the main diagonal. Solving

$$s^{(k)}_{1:k} = (R^{(k)} y^{(k)})_{1:k}$$

with $s^{(k)} := \beta(Q^{(k)} e_1)$ yields $y^{(k)}$ for minimising (2.24). Unlike in the GMRES method we can formulate updates for the iterates $x^{(k)}$ that do not require storing the whole matrix $V^{(k)}$. This is done by introducing the matrix $P^{(k)} = [p^{(1)}| \dots |p^{(k)}]$ as

$$P^{(k)} := V^{(k)} (R^{(k)}_{1:k,1:k})^{-1}. \tag{2.25}$$

Now we can write (2.14) as

$$\begin{aligned} x^{(k)} &= x^{(0)} + V^{(k)} y^{(k)} \\ &= x^{(0)} + P^{(k)} s^{(k)}_{1:k} \\ &= x^{(k-1)} + s^{(k)}_k p^{(k)}. \end{aligned}$$

From (2.25) we see that $p^{(k)}$ can be computed using only the previous two vectors $p^{(k-1)}$ and $p^{(k-2)}$ as well as $v^{(k)}$, i.e.

$$p^{(k)} = \frac{1}{R^{(k)}_{k,k}} \left(v^{(k)} - R^{(k)}_{k-1,k} p^{(k-1)} - R^{(k)}_{k-2,k} p^{(k-2)} \right).$$

In Algorithm 2.6 we state the resulting QMR algorithm. We did not discuss how to find a proper criterion for stopping the iteration and use the norm of $r^{(k)}$ instead. In actual computation we would not want to compute $r^{(k)} = b - Ax^{(k)}$ in every step. For more information we refer to [Saa03, Section 7.3.2]. There it is discussed that $|s^{(k)}|$ decreases monotonically and can be used as a stopping criterion. Moreover, we chose $w^{(1)} = v^{(1)}$ but any w with $\langle w, v^{(1)} \rangle \neq 0$ would do.

Algorithm 2.6: QMR

```
Input  :  $A \in \mathbb{C}^{n \times n}$                       system matrix
          $b \in \mathbb{C}^n$, $x^{(0)} \in \mathbb{C}^n$   right-hand side and initial guess

Output:   approximate solution $x^{(k)}$ to $Ax = b$

1  $r^{(0)} = b - Ax^{(0)}$
2  $\beta^{(0)} = \|r^{(0)}\|_2$
3  $w^{(1)} = v^{(1)} = r^{(0)}/\beta^{(0)}$
4  $p^{(0)} = p^{(-1)} = 0$
5  $s^{(0)} = \beta^{(0)} e_1$
6  for $k = 1, 2, \ldots$ until convergence do
7      compute $v^{(k+1)}$,$w^{(k+1)}$, $T^{(k+1,k)}$          // two-sided Lanczos process (Alg. 2.3)
8      apply $G^{(k-1)} G^{(k-2)}$ to the last column of $T^{(k+1,k)}$ yielding $\widetilde{R}^{(k)}$
9      compute the Givens rotation $G^{(k)}$ using $\widetilde{R}^{(k)}$
10     apply $G^{(k)}$ to the result $\widetilde{R}^{(k)}$ of line 8 yielding $R^{(k)}$
11     apply $G^{(k)}$ to the $k$-th and $(k+1)$-st entry of $s^{(k-1)}$ yielding $s^{(k)}$
12     $p^{(k)} = \frac{1}{R^{(k)}_{k,k}} \left( v^{(k)} - R^{(k)}_{k-1,k} p^{(k-1)} - R^{(k)}_{k-2,k} p^{(k-2)} \right)$
13     $x^{(k)} = x^{(k-1)} + s^{(k)}_k p^{(k)}$
```

3 Matrix Functions

In this section we discuss how a *function of a matrix* can be defined and what properties can be derived from the given definitions. As an example we will introduce the matrix sign function. Afterwards we will describe how $f(A)$ and—more importantly for our purposes—$f(A)b$ can be approximated.

It turns out that by defining a matrix function as specified below, some properties arise as expected if we assume matrix functions to be a generalisation of scalar functions. For the definitions, properties and methods in this section we mainly follow [Hig08] and [FS08]. A comprehensive overview can be found in the former. Before giving a formal definition consider the following example. If f is given as a polynomial p of degree d,

$$f(z) = p(z) = \sum_{i=0}^{d} a_i z^i,$$

then it is natural to extend this function to $p : \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ as

$$p(A) = \sum_{i=0}^{d} a_i A^i.$$

This canonical way of building a matrix function from a scalar polynomial covers only a subset of all scalar functions, albeit an important one. The expansion of scalar functions to matrix arguments is therefore motivated by the Weierstraß approximation theorem [CC04, Theorem 1.2.2]. However, a more flexible and precise definition than “substitute A for z” is needed.

3.1 Definition and Properties

3.1.1 Jordan Canonical Form Definition

The first definition makes use of the *Jordan canonical form* (see [HJ91]), which exists for every square matrix $A \in \mathbb{C}^{n\times n}$ and expresses A in the form

$$A = ZJZ^{-1} = Z \operatorname{diag}\left(J_1, J_2, \dots, J_p\right) Z^{-1}, \tag{3.1}$$

where the *Jordan blocks* J_k have the form

$$J_k = \begin{bmatrix} \lambda_k & 1 & 0 & \cdots & 0 \\ 0 & \lambda_k & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & 1 \\ 0 & \cdots & \cdots & 0 & \lambda_k \end{bmatrix} \in \mathbb{C}^{m_k \times m_k},$$

the matrix Z is non-singular and $m_1 + m_2 + \cdots + m_p = n$. The following definition introduces a terminology that is needed to express the required properties of f for extending f to matrices.

Definition 3.1. [Hig08, Definition 1.1] *Let $\lambda_1, \lambda_2, \ldots, \lambda_s$, $s \le p$, denote the distinct eigenvalues of A, let p be the number of Jordan blocks of A and let n_i be the order of the largest Jordan block in which λ_i appears. The function f is then said to be* defined on the spectrum *of A, if the values*

$$f^{(j)}(\lambda_i), \quad j = 0, 1, \ldots, n_i - 1, \quad i = 1, 2, \ldots, s$$

exist.

Now the function f can be defined on $\mathbb{C}^{n \times n}$.

Definition 3.2. [Hig08, Definition 1.2] *Let f be defined on the spectrum of $A \in \mathbb{C}^{n \times n}$ and let A have the Jordan canonical form* (3.1). *Then*

$$f(A) := Zf(J)Z^{-1} = Z \operatorname{diag}\left(f(J_1), f(J_2), \ldots, f(J_p)\right) Z^{-1},$$

where

$$f(J_k) := \begin{bmatrix} f(\lambda_k) & f'(\lambda_k) & \cdots & \cdots & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ 0 & f(\lambda_k) & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & f'(\lambda_k) \\ 0 & \cdots & \cdots & 0 & f(\lambda_k) \end{bmatrix} \tag{3.2}$$

Since this definition relies on the Jordan canonical form, in which Z and J are not unique, it is not immediately clear that the definition does not depend on the choice of Z and J. The proof that this is the case is not obvious and can be found in [HJ91, Section 6.2].

Another remark is that the numerical computation of a matrix function by explicitly using the above definition would suffer from stability problems when computing the Jordan canonical form (cf. [GL96, Example 11.1.1]).

3.1.2 Polynomial Interpolation Definition

A second approach for defining a matrix function is via *polynomial interpolation.* Following the notation of Definition 3.1, the *minimal polynomial* ψ of A can be written as

$$\psi(z) = \prod_{i=1}^{s} (z - \lambda_i)^{n_i}.$$

As is known from linear algebra, the minimal polynomial divides any other polynomial p that fulfils $p(A) = 0$. For every matrix $A \in \mathbb{C}^{n \times n}$ and any polynomial $p(z)$, $p(A)$ is defined. Moreover, p is defined on the whole complex plane and particularly on the spectrum of A. The needed property for the next definition is, that for polynomials the values of p on the spectrum of A determine $p(A)$. Definition 3.3 generalises the above to arbitrary functions f.

Definition 3.3. [Hig08, Definition 1.4] *Let f be defined on the spectrum of $A \in \mathbb{C}^{n \times n}$ and let ψ denote the minimal polynomial of A. Then $f(A) := r(A)$, where r is the unique Hermite interpolating polynomial of degree less than*

$$\sum_{i=1}^{s} n_i = \deg \psi$$

that satisfies the interpolation conditions

$$r^{(j)}(\lambda_i) = f^{(j)}(\lambda_i), \quad j = 0, 1, \dots, n_i - 1, \quad i = 1, 2, \dots, s. \tag{3.3}$$

Note that $f(A)$ is therefore defined explicitly as a polynomial in A that depends on the values of f on the spectrum of A. This definition turns out to be most useful in computing approximations to matrix functions and its effect on a vector as is illustrated in Section 3.2 and Section 3.3. The point is that for finding a good approximation to the function of a matrix, one can focus on the approximation properties on the eigenvalues of the matrix or on a set containing the eigenvalues.

Since the definition using the Jordan canonical form only depends on the values $f^{(m)}(\lambda_k)$ as seen in equation (3.2), the interpolation conditions (3.3) show that both definitions are equivalent.

3.1.3 Cauchy Integral Definition

Finally, f can be expanded onto matrices by using the Cauchy integral theorem.

Definition 3.4. [Hig08, Definition 1.11] *Let $A \in \mathbb{C}^{n\times n}$ and $f(z)$ be analytic inside and on a simple closed rectifiable curve Γ, which strictly encloses* $\operatorname{spec}(A)$. *Then $f(A)$ is defined to be*

$$f(A) := \frac{1}{2\pi i}\oint_{\Gamma} f(z)(zI - A)^{-1}dz.$$

In this definition $(zI - A)$ is always invertible since Γ strictly encloses $\operatorname{spec}(A)$. Note that in the two previous definitions the function f was not needed to be analytic. This definition of a matrix function is equivalent to the other two definitions modulo the requirement on f to be analytic, which is proven in [HJ91, Theorem 6.2.28].

3.1.4 Properties

The next few theorems taken from [Hig08] illustrate some fundamental properties of matrix functions.

Theorem 3.5. [Hig08, Theorem 1.13] *Let $A, X \in \mathbb{C}^{n\times n}$ and let f be defined on the spectrum of A. Then*

1. *$f(A)$ commutes with A.*
2. *$f(A^T) = f(A)^T$.*
3. *$f(XAX^{-1}) = Xf(A)X^{-1}$ for non-singular X.*
4. *If X commutes with A then X commutes with $f(A)$.*
5. *If $A = (A_{ij})$ is block triangular then $F = f(A)$ is block triangular with the same block structure as A and $F_{ii} = f(A_{ii})$.*
6. *If $A = \operatorname{diag}(A_{11}, A_{22}, \dots, A_{mm})$ is block diagonal then*

$$f(A) = \operatorname{diag}(f(A_{11}), f(A_{22}), \dots, f(A_{mm})).$$

Proof. See [Hig08]. □

Theorem 3.6. [Hig08, Theorem 1.14] *With the previous notation, $f(A) = g(A)$ if and only if*

$$f^{(j)}(\lambda_i) = g^{(j)}(\lambda_i), \quad j = 0, 1, \dots, n_i - 1, \quad i = 1, 2, \dots, s.$$

Equivalently, $f(A) = 0$ if and only if

$$f^{(j)}(\lambda_i) = 0, \quad j = 0, 1, \dots, n_i - 1, \quad i = 1, 2, \dots, s.$$

Proof. This results from Definition 3.2 and Definition 3.3. □

Theorem 3.7. [Hig08, Theorem 1.15] *Let f and g be functions defined on the spectrum of A.*

1. *If $h(z) = f(z) + g(z)$ then $h(A) = f(A) + g(A)$.*
2. *If $h(z) = f(z)g(z)$ then $h(A) = f(A)g(A)$.*

Proof. See [Hig08]. □

3.1.5 The Matrix Sign Function

Before we move on to describe how to actually compute a matrix function we introduce the matrix sign function to have an example at hand. In the scalar case the *sign function* is defined by

$$\operatorname{sign}(z) = \begin{cases} +1 & , \operatorname{Re}(z) > 0, \\ -1 & , \operatorname{Re}(z) < 0. \end{cases}$$

where $z \in \mathbb{C}$, $\operatorname{Re}(z) \neq 0$. All derivatives $\operatorname{sign}^{(k)}(z)$ of the sign function are zero for $k \geq 1$. Thus, the definition using the Jordan canonical form (Definition 3.2) adjusted for the matrix sign function becomes the following.

Definition 3.8. *Let $A \in \mathbb{C}^{n\times n}$ have no eigenvalues lying on the imaginary axis and be represented by the Jordan canonical decomposition $A = ZJZ^{-1}$,*

$$J = \begin{bmatrix} J_+ & 0 \\ 0 & J_- \end{bmatrix},$$

where the eigenvalues of $J_+ \in \mathbb{C}^{p\times p}$ lie in the open right half-plane and those of $J_- \in \mathbb{C}^{q\times q}$ in the open left half-plane. Then

$$\operatorname{sign}(A) = Z \begin{bmatrix} I_p & 0 \\ 0 & -I_q \end{bmatrix} Z^{-1}.$$

The matrix sign function *is not defined if A has purely imaginary eigenvalues.*

Another representation can be derived by generalising the scalar identity $\operatorname{sign}(z) = z(z^2)^{-1/2}$ which yields

$$\operatorname{sign}(A) = A(A^2)^{-1/2} \tag{3.4}$$

via Theorem 3.7. The requirement on A having no purely imaginary eigenvalues in the previous definition translates to A^2 having no eigenvalues on $\mathbb{R}_-$ in this case which guarantees the existence of a *principal square root* (cf. [Hig08, Chapter 1.7] for information on the principal square root).

The next theorem shows some properties of $\operatorname{sign}(A)$ and finishes our introduction of the matrix sign function.

Theorem 3.9. [Hig08, Theorem 5.1] *Let $A \in \mathbb{C}^{n\times n}$ have no purely imaginary eigenvalues. Then*

1. $\operatorname{sign}(A)^2 = I$.
2. $\operatorname{sign}(A)$ *is diagonalisable with eigenvalues* $+1$ *and* -1.
3. $\operatorname{sign}(A)\, A = A \operatorname{sign}(A)$.

Proof. The first two properties are directly implied by the definition of the matrix sign function. The third property holds for every matrix function, see Theorem 3.5. □

3.2 Approximating $f(A)$

The matrix $f(A)$ can be computed in multiple ways. Some methods are tailored for a particular matrix function and others can be used for general matrix functions. Our main interest will be in the approximation of the vector $f(A)x$. We will postpone the discussion of solutions for this problem until Section 3.3. For completeness, we describe a few ways to compute the whole matrix $f(A)$ in the following. Moreover, some of the following methods can be adapted to approximate $f(A)x$.

3.2.1 Matrix Iterations

Some matrix functions can be computed by iterations that converge to $f(A)$. Using our example—the matrix sign function—we can obtain $\operatorname{sign}(A)$ by generalising the scalar Newton iteration for $z^2-1=0$. This results in the iteration

$$X^{(k+1)} = \frac{1}{2}\left(X^{(k)} + (X^{(k)})^{-1}\right), \quad X^{(0)} = A. \tag{3.5}$$

The convergence of iteration (3.5) is global and asymptotically quadratic. Figure 3.1 displays the relative error after applying a couple of steps of the Newton iteration for the sign function to a rectangular region in $\mathbb{C}$.

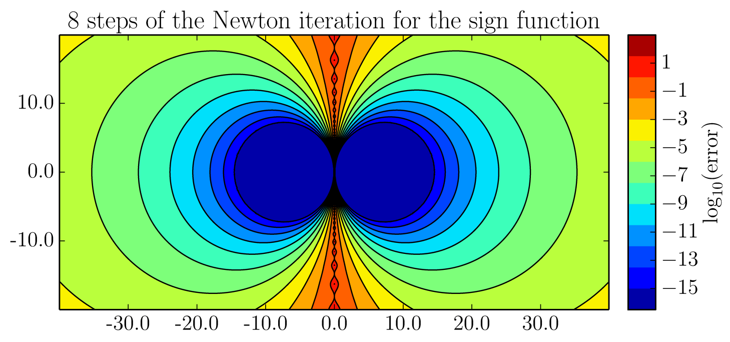


Figure 3.1: Error $|x^{(k)} - \text{sign}(x^{(0)})\,|$ for $x^{(0)} \in \mathbb{C}$ after 8 steps of the Newton iteration for the sign function.

Figure 3.1 suggests, and in [Hig08, Theorem 5.6] it is shown, that the convergence speed suffers if $\rho(A) \gg 1$ or if A has an eigenvalue of small real part. To speed up the initial convergence rate, iteration (3.5) can be scaled to

$$X^{(k+1)} = \frac{1}{2}(\mu^{(k)} X^{(k)} + (\mu^{(k)})^{-1}(X^{(k)})^{-1}), \quad X^{(0)} = A,$$

using a parameter $\mu^{(k)}$. Several strategies for selecting the scaling parameter $\mu^{(k)}$ have been proposed [Hig08; Hig$^+$04], see Table 3.1. All three scaling factors

Table 3.1: Strategies for scaling the Newton iteration for sign(A)

scaling factor	name of the strategy
$\mu^{(k)} = \lvert\det(X^{(k)})\rvert^{-1/n}$	determinantal scaling
$\mu^{(k)} = \sqrt{\rho((X^{(k)})^{-1})/\rho(X^{(k)})}$	spectral scaling
$\mu^{(k)} = \sqrt{\lVert(X^{(k)})^{-1}\rVert \,/\, \lVert X^{(k)}\rVert}$	norm scaling

of Table 3.1 can be cheaply computed or estimated. The determinantal scaling can be obtained from the LU factorisation that is used to compute $(X^{(k)})^{-1}$. The factor μ_k in the spectral scaling variant can be estimated via a power method for $X^{(k)}$ and $(X^{(k)})^{-1}$. Taking the Frobenius norm yields an easy way to calculate the third scaling parameter. The iteration can be continued using (3.5) after an initial phase of scaling, since all three scaling factors converge to 1 as $X^{(k)} \to \text{sign}(A)$. Stopping criteria for the (scaled) Newton iteration are discussed in [Hig08] and comprise, for example,

$$\frac{\left\|X^{(k+1)} - X^{(k)}\right\|}{\left\|X^{(k+1)}\right\|} \leq \left\|X^{(k+1)}\right\|^p \eta$$

for $p \in \{0, 1, 2\}$ and a convergence tolerance η.

3.2.2 Padé Approximations

Using *rational approximations* for a function f instead of polynomial approximations usually results in reaching the same approximation quality with less degrees of freedom [Che82, Chapter 5]. A class of rational approximations are the so-called *Padé approximations* which we define next.

Definition 3.10. *Let $f(z)$ be a scalar function. The rational function*

$$r_{l/m}(z) = \frac{p_l(z)}{q_m(z)}$$

is an $[l/m]$-type Padé approximant *of f if $r_{l/m} \in \mathcal{R}_{l/m}$, $q_m \in \overline{\Pi}_m$ and*

$$f(z) - r_{l/m}(z) = \mathcal{O}(z^{l+m+1}) \tag{3.6}$$

for $z \to 0$.

In order to be able to efficiently apply $r_{l/m}$ to a matrix A we need to write the rational function as a partial fraction expansion. Since every rational function $r_{l/m}$ can be expressed as a *partial fraction expansion* we can write

$$r_{l/m}(z) = \frac{p_l(z)}{q_m(z)} = \pi(z) + \sum_{j=1}^{n} \sum_{i=1}^{m_j} \frac{\omega_{j,i}}{(z - \sigma_j)^i}$$

where m_j is the multiplicity of pole $\sigma_j \in \mathbb{C}$, $\sum_j^n m_j = m$, $\pi(z) \in \Pi_{l-m}$ if $l \geq m$ and $\pi \equiv 0$ otherwise. From here on we assume a multiplicity of 1 for every pole, and applying this to a matrix we get

$$r_{l/m}(A) = \pi(A) + \sum_{j=1}^{n} \omega_j (A - \sigma_j I)^{-1}. \tag{3.7}$$

Inverting multiple matrices $(A - \sigma_j I)$ seems expensive at first sight. But in Chapter 5 we will introduce methods that can efficiently solve families of shifted systems of the kind $(A - \sigma_j I)x = b$.

There are two classes of algorithms for rational approximations. The first one solves the *value problem*, i.e. the algorithm evaluates $r_{l/m}(z)$ for some specific value of z. The second class of algorithms solves the *coefficient problem* which means that the coefficients defining p_l and q_m are computed. Since we want to evaluate the rational function $r_{l/m}(z)$ on a matrix, we are interested in a representation for $r_{l/m}$ and consequently have to solve the coefficient problem. This can be done in the following way, see for instance [BG96]. Let

$$\begin{aligned} p_l(z) &= \eta_0 + \eta_1 z + \eta_2 z^2 + \eta_l z^l \quad \text{ and} \\ q_m(z) &= 1 + \theta_1 z + \theta_2 z^2 + \theta_m z^m. \end{aligned}$$

With (3.6) we see that p_l and q_m need to fulfil

$$f^{(i)}(0) = r^{(i)}_{l/m}(0) \quad \text{for } i = 0, 1, \dots, l + m.$$

Since $q_m(z)$ is normalised to $\theta_0 = 1$, the approximation $r_{l/m}(z)$ has $l + m + 1$ unknown coefficients. Let

$$f(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_{l+m} z^{l+m} + \mathcal{O}(z^{l+m+1})$$

be the *Taylor expansion* of f at $z_0 = 0$. Because of (3.6) we obtain

$$\left(\sum_{i=0}^{l+m} a_i z^i\right)\left(\sum_{i=0}^{m} \theta_i z^i\right) - \left(\sum_{i=0}^{l} \eta_i z^i\right) = 0$$

which means we have to solve

$$\begin{aligned} \theta_m a_{l-m+1} + \theta_{m-1} a_{l-m+2} + \dots + \theta_1 a_l \quad + a_{l+1} &= 0 \\ \theta_m a_{l-m+2} + \theta_{m-1} a_{l-m+3} + \dots + \theta_1 a_{l+1} \quad + a_{l+2} &= 0 \\ &\vdots \\ \theta_m a_l + \theta_{m-1} a_{l+1} \quad + \dots + \theta_1 a_{l+m-1} + a_{l+m} &= 0 \end{aligned} \tag{3.8}$$

for $\theta_1, \dots, \theta_m$ and

$$\begin{aligned} a_0 - \eta_0 &= 0 \\ \theta_1 a_0 + a_1 - \eta_1 &= 0 \\ &\vdots \\ \theta_m a_{l-m} + \theta_{m-1} a_{l-m+1} + \cdots + a_l - \eta_l &= 0 \end{aligned} \tag{3.9}$$

for $\eta_0, \dots, \eta_l$ afterwards. Thus, we obtain all the coefficients for the polynomials $p_l(z)$ and $q_m(z)$ and can form the rational approximation $r_{l/m}(z)$ fulfilling (3.6). One problem that arises while computing the coefficients $\theta_1, \dots, \theta_m$ and

$\eta_0, \dots, \eta_l$ is that the Taylor coefficients $a_0, \dots, a_{l+m}$ may vary significantly in magnitude. This results in very ill-conditioned systems (3.8) and (3.9). Even for relatively small degrees l and m this can result in failing to be able to compute sufficiently good coefficients in `IEEE-754 double` [IEE08] representation and arithmetic. This can be remedied by computing in higher precision. Figure 3.2 and Figure 3.3 display an example for a Padé approximation of $f(z) = z^{-1/2}$ and $f(z) = \mathrm{sign}(z)$ respectively.

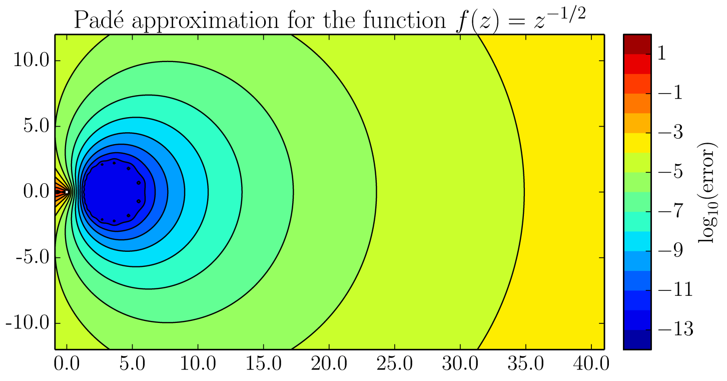


Figure 3.2: Relative error $|r_{l/m}(z) - f(z)|/|f(z)|$ for $z \in \mathbb{C}$ where $r_{l/m}$ is an Padé approximant to $f(z) = z^{-1/2}$ centred at 2.8 with $l = 8$ and $m = 8$.

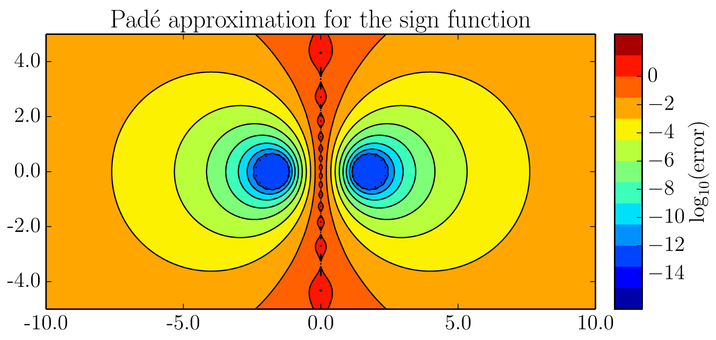


Figure 3.3: Error $|zr_{l/m}(z^2) - \mathrm{sign}(z)|$ for $z \in \mathbb{C}$ where $r_{l/m}$ is the same Padé approximant to $z^{-1/2}$ as in Figure 3.2.

3.2.3 n-Point Padé Approximations

In many applications a function f has to be approximated in a specific region $S \subset \mathbb{C}$. If we want to apply the approximation to a matrix A then we want it to be precise on the spectrum of A as it is suggested by Definition 3.1.

Moreover, if A is Hermitian then S becomes a real interval $[\underline{x}, \overline{x}]$. The Padé approximation defined in Definition 3.10, however, is not well suited for such a situation as can be seen in Figure 3.2 and Figure 3.3. A generalisation of the Padé approximation defined below can remedy this.

Definition 3.11. *Let $f(z)$ be a scalar function. The rational function*

$$r_{l/m}(z) = \frac{p_l(z)}{q_m(z)}$$

*is a $[l/m]$-type n-*point Padé approximant *of f w.r.t. the n-tuple $(z_1, \dots, z_n)$ of points if $r_{l/m} \in \mathcal{R}_{l/m}$, $q_m \in \overline{\Pi}_m$ and $l + m + 1 = n$ as well as*

$$f(z) - r_{l/m}(z) = \mathcal{O}((z - z_i)^{c_i})$$

for all i and $z \to z_i$. Here, c_i denotes the cardinality of the set

$$J_i := \{j : z_j = z_i\}.$$

In the case of $z_1 = \cdots = z_n$ the n-point Padé approximant reduces to the Padé approximant at point $z = z_1$.

Solving the coefficient problem for the n-point Padé approximant can be done using *Kronecker's algorithm* [BG96] which dates back to [Kro81]. Before we introduce the algorithm we want to recall *polynomial interpolation*, where we are interested in finding a polynomial $p(z) = \prod_{i=0}^{n-1} a_i z^i$ that interpolates a scalar function $f(z)$ on the n-tuple of points $(z_1, \dots, z_n)$, i.e. $p(z_i) = f(z_i)$ for all $i \in \{1, 2, \dots, n\}$. In the case of confluent points $z_i = z_{i+1} = \cdots = z_k$ we want to interpolate the derivatives like in Definition 3.11. One method to find such an interpolating polynomial is the *Newton polynomial interpolation* which involves *divided differences* that are defined below.

Definition 3.12. *Let $f(z)$ be a sufficiently often differentiable scalar function. The divided differences on the n-tuple of points $(z_1, z_2, \dots, z_n)$ are defined recursively by*

$$\begin{aligned} f[z_i] &:= f(z_i), \\ f[z_i, \dots, z_{i+j}] &:= \frac{f[z_{i+1}, \dots, z_{i+j}] - f[z_i, \dots, z_{i+j-1}]}{z_{i+j} - z_i}. \end{aligned}$$

For confluent points $z_i = z_{i+1} = \cdots = z_k$ we define

$$f[z_i, \dots, z_k] := \frac{f^{(k-1)}(z_i)}{(k-1)!}.$$

The Newton representation of the polynomial interpolating $f(z)$ on the points $(z_1, z_2, \dots, z_n)$ is then defined as

$$p(z) := \sum_{i=1}^{n} f[z_1, z_2, \dots, z_i] \prod_{j=0}^{i-1} (z - z_j).$$

The idea of Kronecker's algorithm is to start with the $[l+m/0]$-type n-point Padé approximant obtained from a polynomial interpolation. Thereafter, the algorithm successively builds the $[s/t]$-type approximant using the previous two $[s+2/t-2]$ and $[s+1/t-1]$ n-point Padé approximants. The algorithm stops as soon as the $[l/m]$-type n-point Padé approximant is reached. In Algorithm 3.1 we present Kronecker's algorithm in a basic form.

Algorithm 3.1: Kronecker's Algorithm

```
Input  :  $(z_1, \dots, z_n)$             $n$-tuple of interpolation points
                                           allowing confluent points
          $(f(z_1), \dots, f(z_n))$       function values of $f$ at the interpolation points
          $l, m$                          $[l/m]$-type of the $n$-point Padé approximant
                                           $l + m + 1 = n$

Output:   $p_l$, $q_m$   coefficients defining the polynomials $p_l$ and $q_m$ and thus the
                         $n$-point Padé approximant $r_{l/m}$

1 set $p_{m+l}(z) = \sum_{i=1}^{n} f[z_1, \dots, z_i] \prod_{j=1}^{i-1} (z - z_j)$
2 set $q_0(z) = 1$
3 set $p_{m+l+1}(z) = \prod_{j=1}^{n} (z - z_j)$
4 set $q_{-1}(z) = 0$
5 for $s = 1, 2, \dots, m$ do
6     determine $\alpha_s$ and $\beta_s$, s.t. the degree of
          $(\alpha_s z + \beta_s) p_{m+l-s+1}(z) - p_{m+l-s+2}(z)$
      is at most $m + l - s$
7     set $p_{m+l-s}(z) = (\alpha_s z + \beta_s) p_{m+l-s+1}(z) - p_{m+l-s+2}(z)$
8     set $q_s(z) = (\alpha_s z + \beta_s) q_{s-1}(z) - q_{s-2}(z)$
```

We note that in line 6 of Algorithm 3.1 the values α_s and β_s might be determined, so that the resulting polynomial has a degree less than $m+l-s$. Such a polynomial is called *degenerate* and implies that the according interpolant does not exist. It is a well known fact that some interpolants might not exist [BG96]. In this case the algorithm can be modified to skip the respective approximants and only compute the non-degenerate approximants.

For the Padé approximation we have already discussed that the computation can suffer from intermediate values not being accurately representable as `double` values. Thus, we implemented Algorithm 3.1 in Maple allowing computations in arbitrary precision. Only the resulting coefficients of the numerator and denominator polynomial are then converted to `double`. The crucial part here is the choice of the interpolation nodes the approximation is based upon. In Figure 3.4 and Figure 3.5 an n-point Padé approximation of $f(z) = z^{-1/2}$ and of $f(z) = \operatorname{sign}(z)$ respectively is depicted. We computed the interpolation nodes using

$$z_i = \left(\left(1 - \cos\left(\frac{i\pi}{2n}\right)\right)^4 - \frac{1}{2}\right)(b-a) + \frac{a+b}{2} \tag{3.10}$$

for $i = 0, \dots, n$ with $a = 0.1$ and $b = 40$. This results in denser interpolation nodes at the left end of $[a, b]$ improving the otherwise low approximation quality close to a.

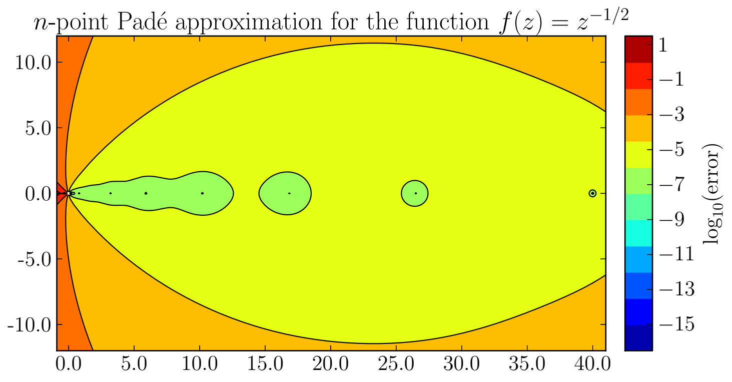


Figure 3.4: Relative error $|r_{l/m}(z) - f(z)|/|f(z)|$ for $z \in \mathbb{C}$ where $r_{l/m}$ is an n-point Padé approximant to $f(z) = z^{-1/2}$ with $l = 8$ and $m = 8$.

An other way to compute a rational approximation would be to apply the Remez algorithm [Che82, Chapter 3.8]. But the one implementation for rational approximations that we found in Maple did not converge for our examples.

3.2.4 Optimal Approximations

For some functions f and subsets $S \subset \mathbb{C}$ there exist directly computable *optimal rational approximations* $r_{l/m}$. Here, we use "optimal" in the sense that $r_{l/m} \in \mathcal{R}_{l/m}$ is minimising $\sup_{z \in S} |f(z) - r(z)|$ among all $r \in \mathcal{R}_{l/m}$.

For the sign function and $S = [-b, -a] \cup [a, b]$ with $0 < a \leq b$ such an approximation can be stated explicitly. It is again based on an approximation

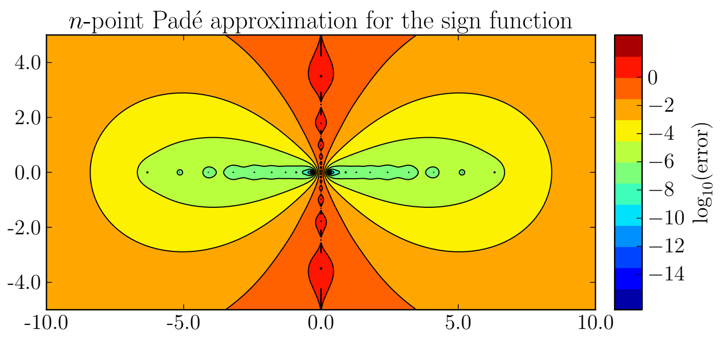


Figure 3.5: Error $|zr_{l/m}(z^2) - \operatorname{sign}(z)|$ for $z \in \mathbb{C}$ where $r_{l/m}$ is the same Padé approximant to $f(z) = z^{-1/2}$ as in Figure 3.4.

of the inverse square root for a positive real interval. In [PP87, Theorem 4.6] it is shown that these two problems—finding a best rational approximation for the sign function and the inverse square root—are equivalent. These best rational approximations were derived by Zolotarev [PP87, Theorem 4.8]. From here on we will call optimal rational approximations of this kind Zolotarev approximations, and it will be stated or clear by context if we refer to the approximation of the sign function or the inverse square root.

We now introduce the *Zolotarev approximation* of the sign function briefly. The following theorem makes use of the *Jacobi elliptic function* $\operatorname{sn}(w, \kappa) = x$ that is defined implicitly by the *elliptic integral*

$$w = \int_0^x \frac{1}{\sqrt{(1-t^2)(1-\kappa^2 t^2)}} dt$$

and the *complete elliptic integral* of the first kind for the modulus κ is defined by

$$K(\kappa) = \int_0^1 \frac{1}{\sqrt{(1-t^2)(1-\kappa^2 t^2)}} dt.$$

The following Theorem 3.13 summarises section 4.3 from [PP87] and its main result [PP87, Theorem 4.8] dates back to Zolotarev [Zol77].

Theorem 3.13. [FS08, Proposition 4] *Let $r_{2t-1/2t}(z) = p_{2t-1}(z)/q_{2t}(z)$ be the Chebyshev best approximation to $\operatorname{sign}(z)$ on the set $[-b, -a] \cup [a, b]$, i.e. the function that minimises*

$$\max_{a<|z|<b} \left|\operatorname{sign}(z) - \tilde{r}_{2t-1/2t}(z)\right|$$

over all rational functions $\tilde{r}_{2t-1/2t}(z) = \tilde{p}_{2t-1}(z)/\tilde{q}_{2t}(z)$. Then the factored *form of* $r_{2t-1/2t}$ *is given by*

$$\begin{aligned} r_{2t-1/2t}(z) &= az \cdot s_{t-1/t}((az)^2) \qquad \textit{with} \\ s_{t-1/t}(z) &= D\frac{\prod_{j=1}^{t-1}(z+c_{2j})}{\prod_{j=1}^{t}(z+c_{2j-1})}, \end{aligned}$$

where

$$c_j = \frac{\operatorname{sn}^2\left(jK(\kappa')/(2t);\kappa'\right)}{1-\operatorname{sn}^2\left(jK(\kappa')/(2t);\kappa'\right)}$$

for $\kappa' := \sqrt{1-(a/b)^2}$ *and* D *is uniquely determined by the condition*

$$\max_{z\in[1,(b/a)^2]}\left(1-\sqrt{z}s_{t-1/t}(z)\right) = -\min_{z\in[1,(b/a)^2]}\left(1-\sqrt{z}s_{t-1/t}(z)\right).$$

The scaling coefficient D can be explicitly computed in the following way. According to [PP87] the rational approximation takes its maximum and minimum value on the interval $[1,(b/a)^2]$ exactly $2t+1$ times. Let the Jacobi elliptic functions $\operatorname{cn}(w,\kappa)$ and $\operatorname{dn}(w,\kappa)$ be defined by

$$\begin{aligned} \operatorname{cn}^2(w,\kappa) &:= 1-\operatorname{sn}^2(w,\kappa) \qquad \text{and} \\ \operatorname{dn}^2(w,\kappa) &:= 1-\kappa^2\operatorname{sn}^2(w,\kappa). \end{aligned}$$

The complementary modulus for κ is defined by $\kappa'^2+\kappa^2=1$ and $K' := K(\kappa')$ denotes the complete elliptic integral of the first kind for the complementary modulus κ'. In [Ken05] it is shown that evaluating the unscaled approximation at points ξ_j^2 for $j=0,1,\dots,2t$ gives the alternating maximum and minimum values. Denoting $K := K(\kappa)$, the points ξ_j^2 are known to be

$$\begin{aligned} \xi_j &= \operatorname{sn}\left(K+\frac{ijK'}{2t},\kappa\right) \\ &= \frac{\operatorname{cn}\left(\frac{ijK'}{2t},\kappa\right)}{\operatorname{dn}\left(\frac{ijK'}{2t},\kappa\right)}, \end{aligned}$$

where the second equality can be derived using addition formulas for Jacobi elliptic functions. By using *Jacobi's imaginary transformations*

$$\begin{aligned} \operatorname{cn}\left(\frac{ijK'}{2t},\kappa\right) &= \frac{1}{\operatorname{cn}\left(\frac{jK'}{2t},\kappa'\right)} \qquad \text{and} \\ \operatorname{dn}\left(\frac{ijK'}{2t},\kappa\right) &= \frac{\operatorname{dn}\left(\frac{jK'}{2t},\kappa'\right)}{\operatorname{cn}\left(\frac{jK'}{2t},\kappa'\right)} \end{aligned}$$

the result is

$$\xi_j = \frac{1}{\mathrm{dn}\left(\frac{jK'}{2t}, \kappa'\right)}.$$

Since the approximation alternates between its extremal values, it is sufficient to evaluate, e.g. at $\xi_0 = 1$ and ξ_1. Thus, the scaling factor D is given by

$$D = \frac{2}{\hat{r}_{2t-1/2t}(\xi_0) + \hat{r}_{2t-1/2t}(\xi_1)},$$

where

$$\hat{r}_{2t-1/2t}(z) = z \cdot \frac{\prod_{j=1}^{t-1}(z^2 + c_{2j})}{\prod_{j=1}^{t}(z^2 + c_{2j-1})}.$$

Like for the Padé approximants this rational function can be rewritten as a partial fraction expansion. In [Esh⁺02] the Zolotarev approximants were compared to Padé approximants stemming from a *Remez algorithm*. The Zolotarev approximation appeared to need less poles for achieving the same accuracy.

Figure 3.6 and Figure 3.7 depict Zolotarev approximations to $f(z) = z^{-1/2}$ and $f(z) = \mathrm{sign}(z)$ respectively. The sign function was approximated on the interval $[a, b]$ with endpoints $a = 0.1$ and $b = 40$.

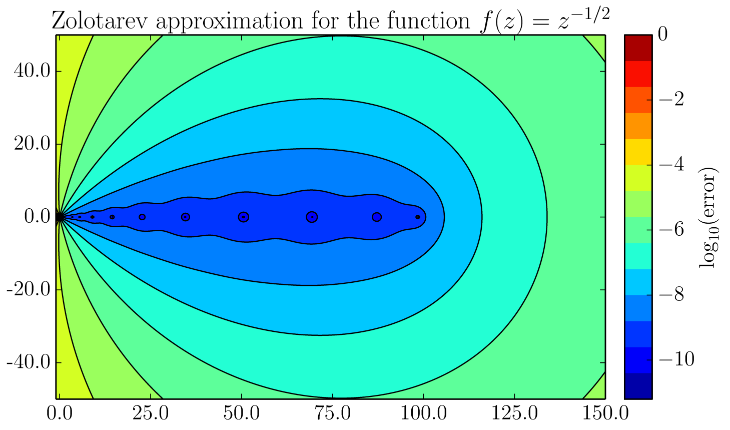


Figure 3.6: Relative error $|s_{t-1/t}(z) - f(z)|/|f(z)|$ for $z \in \mathbb{C}$ where $s_{t-1/t}$ is the Zolotarev approximant to $f(z) = z^{-1/2}$ with $t = 6$.

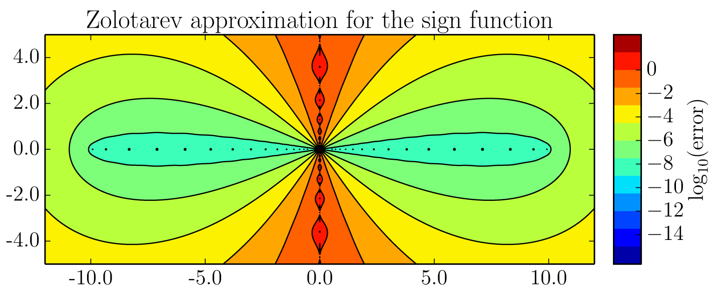


Figure 3.7: Error $|r_{2t-1/2t}(z) - \operatorname{sign}(z)|$ for $z \in \mathbb{C}$ where $r_{2t-1/2t}$ is the Zolotarev approximant to $f(z) = \operatorname{sign}(z)$ with $t = 6$, $a = 0.1$ and $b = 10$.

3.3 Approximating $f(A)b$

We have only described how the complete matrix $f(A)$ can be computed or approximated up till now. The remaining part of this chapter deals with the computational aspects to obtain the action of $f(A)$ on a vector b namely $f(A)b$. As mentioned before, if A is a large sparse matrix then computing the whole matrix $f(A)$ just for obtaining $f(A)b$ is usually far too expensive or even impossible. This is because $f(A)$ is not guaranteed to be sparse and usually will not be. However, the matrix-vector product $f(A)b$ can still be computable at affordable speed and memory usage. Some of the methods presented in Section 3.2 can be adapted for that purpose.

3.3.1 Krylov Subspace Approximation for $f(A)b$

From the Arnoldi process and its matrix representation (2.9) an approximation for $f(A)b$ can be obtained. The idea is to project the problem onto the subspace $\mathcal{K}_k(A, b) = \operatorname{span}\{b, Ab, \dots, A^{k-1}b\}$ of smaller dimension $k < n$. Using the matrices $V^{(k)}$ and $H^{(k)}$ of the Arnoldi process one can formulate the approximation

$$\begin{aligned} f(A)b &\approx f(V^{(k)}H^{(k)}(V^{(k)})^H)b \\ &= V^{(k)}f(H^{(k)})(V^{(k)})^H b \\ &= V^{(k)}f(H^{(k)})e_1 \|b\| , \end{aligned} \tag{3.11}$$

where the equality in the second line holds because of the polynomial definition of a matrix function in Definition 3.3 and e_1 is the first unit k-vector. This reduces the approximation of $f(A)b$ to a computation of a smaller matrix function $f(H^{(k)})$ and can yield good approximations even for $k \ll n$. If A is Hermitian then the Lanczos process (2.11) is applied and the equation reads

$f(A)b \approx V^{(k)} f(T^{(k)}) e_1 \|b\|$. The approximation (3.11) can be interpreted as a polynomial approximation. The following proposition from [FS08, Proposition 2] is a generalisation of [Saa92, Theorem 3.3].

Proposition 3.14. *Let the columns of $V^{(k)}$ form an orthonormal basis of $\mathcal{K}_k(A, b)$ and $H^{(k)} = (V^{(k)})^H A V^{(k)}$. Then the approximation $V^{(k)} f(H^{(k)}) e_1 \|b\|$ represents a polynomial approximation $p(A)b$ to $f(A)b$ in which the polynomial p of degree $k-1$ interpolates f in the Hermite sense on the set of eigenvalues of $H^{(k)}$.*

Proof. With the polynomial definition of a matrix function in Section 3.1.2, $p(H^{(k)})b = f(H^{(k)})b$ for the polynomial $p \in \Pi_{k-1}$ that interpolates f on the eigenvalues of $H^{(k)}$ in the Hermite sense. Considering that

$$p(A)b = V^{(k)} p(H^{(k)}) e_1 \|b\|$$

for all polynomials of degree $\leq k-1$ finishes the proof. □

If f is sufficiently smooth and the matrix A is Hermitian then a bound on the error of the approximation can be obtained. Consider the implicitly built polynomial $p \in \Pi_{k-1}$ of the Lanczos process that approximates the function f in such a way that $\|f(A)b - p(A)b\|_2$ is small. By Proposition 3.14 the polynomial p interpolates f in the Ritz values. The next proposition assumes a general polynomial interpolating f.

Proposition 3.15. [Esh$^+$02, Lemma 2] *For any set of distinct interpolation points $\mu_i, i = 1, \dots, k$ and for any function $f : \mathbb{R} \to \mathbb{C}^k$, let $p \in \Pi_{k-1}$ satisfy $p(\mu_i) = f(\mu_i)$ for $i = 1, \dots, k$ and $q(z) = \prod_{j=1}^{k} (z - \mu_j)$. If all μ_i and all eigenvalues λ_i of the Hermitian matrix A are contained in the interval $[\alpha, \beta]$ then the following estimate holds*

$$\|q(A)b\|_2 \inf_{t \in [a,b]} \left| \frac{f^{(k)}(t)}{k!} \right| \leq \|f(A)b - p(A)b\|_2 \leq \|q(A)b\|_2 \sup_{t \in [a,b]} \left| \frac{f^{(k)}(t)}{k!} \right|.$$

Proof. For interpolating polynomials there exist values $\zeta_i \in [\alpha, \beta]$ (cf. [SB93, Theorem 2.1.4.1]) such that

$$f(\lambda_i) - p(\lambda_i) = q(\lambda_i) \frac{f^{(k)}(\zeta_i)}{k!}.$$

By expressing b as $b = \sum_{i=1}^{n} \gamma_i w_i$, where the w_i are orthogonal eigenvectors of A for the eigenvalue λ_i, the equation

$$\begin{aligned}\|f(A)b - p(A)b\|_2^2 &= \sum_{i=1}^{n} \gamma_i^2 (f(\lambda_i) - p(\lambda_i))^2 \\ &= \sum_{i=1}^{n} \gamma_i^2 q(\lambda_i)^2 \left(\frac{f^{(k)}(\zeta_i)}{k!}\right)^2\end{aligned}$$

holds. Bounding this proves the proposed inequality. □

In this Krylov subspace approximation method, memory consumption grows with k, since the whole matrix $V^{(k)}$ has to be stored. Thus, if k becomes large then it is likely to run out of memory. Even in the Hermitian case this holds true, but a *two-pass method* can circumvent the memory problems. In the first run, the Krylov subspace is built using the Lanczos process, which allows to discard all but the last two columns of $V^{(k)}$. After this run $T^{(k)}$ is available and $f(T^{(k)})e_1 \|b\| = y$ can be computed. In the second run the Lanczos process is used again to rebuild the columns of $V^{(k)}$ and the product $V^{(k)}y = V^{(k)} f(T^{(k)})e_1 \|b\|$ is obtained step by step.

3.3.2 Rational Approximation for $f(A)b$

Another way for computing $f(A)b$ opens up if the function f is given as or approximated by a rational approximation that is written as a partial fraction expansion

$$f(A)b \approx \pi(A)b + \sum_{j=1}^{n} \omega_j (A - \sigma_j I)^{-1} b$$

assuming a multiplicity of 1 for every pole. The expression $\pi(A)v$, π a polynomial, is evaluated straightforwardly, whereas the second term needs more effort. For each j a system $(A - \sigma_j I)^{-1} b$ has to be solved. At first sight, this looks far too expensive, but a property of Krylov subspaces comes in handy at this point. If solved separately, a solver for system j builds up the Krylov subspace $\mathcal{K}_k(A - \sigma_j I, b)$ in step k of the iteration. But $\mathcal{K}_k(A - \sigma_j I, b)$ is identical to $\mathcal{K}_k(A, b)$ as we explain in Chapter 5. Therefore, in principle the iterates of all systems can be updated by performing just one multiplication with A using a Krylov subspace method, e.g. CG. Whether this approach is feasible still depends on the algorithmic formulation of the method. We will elaborate on this in Chapter 5.

In the following, π is neglected. The k-th iterate $x^{(k)}$ for $f(A)b$ is obtained as

$$x^{(k)} = \sum_{j=1}^{m} \omega_j x_j^{(k)} \in \mathcal{K}_k(A, b),$$

where $x_j^{(k)}$ denotes the k-th iterate of system j. This linear combination gives an approximation to the rational approximation for $f(A)b$.

In general, the above gives a different approximation to $f(A)b$ than the one obtained by (3.11). But if f itself is a rational function given by

$$f(z) = \sum_{j=1}^{m} \frac{\omega_j}{z - \sigma_j}$$

and a Krylov subspace method, which imposes a Galerkin condition, is used to obtain the $x_j^{(k)}$ then the following holds

$$\begin{aligned} f(A)b &= \sum_{j=1}^{m} \omega_j (A - \sigma_j I)^{-1} b \\ &\approx \sum_{j=1}^{m} \omega_j V^{(k)} (H^{(k)} - \sigma_j I)^{-1} e_1 \left\| b \right\|_2 \\ &= V^{(k)} f(H^{(k)}) e_1 \left\| b \right\|_2 . \end{aligned}$$

This shows that evaluating the partial fraction expansion in a multi-shift method may coincide with the Krylov subspace approximation (3.11).

4 Applications

In the following we will introduce two categories of applications in which families of shifted systems with multiple right-hand sides (1.1) that we repeat here for convenience

$$(A + \sigma_i I)x_{i,j} = b_j, \quad i = 1, \dots, s, \quad j = 1, \dots, m. \tag{1.1}$$

have to be solved. The first one is the simulation of subatomic particles in *lattice QCD*. We describe only the basics that are needed to understand the nature of the operator for which (1.1) has to be solved. An introduction to lattice QCD can be found in [GL10] or see [Fro$^+$13a; Kah09] for a summary using a notation more familiar for readers with a mathematical background. The second application is the solution of *ill-posed inverse problems* that arise in a plethora of applications. Roughly speaking, they describe problems in which, based on a physical model and some measurements, an entity or object is to be (re-)constructed. We introduce image deconvolution as an example in image restoration [PO02] and describe the involved linear systems. More details on inverse problems can be found in [Tar04] and [Isa98].

4.1 Lattice Quantum Chromodynamics

Quantum Chromodynamics (QCD) is a theory that is part of the standard model of particle physics. It is a quantum field theory in four-dimensional space-time and describes the strong force, i.e. the interaction of quarks and gluons. The particles that these can form are called hadrons and two commonly known representatives are neutrons and protons. In this theory massless gluons bearing a colour charge mediate the strong interactions between quarks. This colour charge describes a property of quarks and can take one of the arbitrarily labelled colours red, green and blue. Besides the colour charge, quarks have an electrical charge, a spin and a mass and come in the six flavours up, down, charm, strange, top and bottom. Since gluons carry a colour charge, they do not only mediate the strong interaction but also take part in it. In the numerical simulations the quarks can be represented by a fermionic field, the Dirac field, whereas the gluons are represented as a vector field, the gauge field.

4.1.1 The Wilson-Dirac Operator

When it comes to simulations, a discretisation onto lattice sites in a finite space is needed. In lattice QCD such a discretisation onto a four-dimensional euclidean space-time lattice with periodic boundary conditions is used. The lattice has the size $n_t \times n_s^3$ where n_t is the number of lattice points in the dimension of time and n_s is the number of lattice points in each dimension of space. Thus, the lattice sites can be indexed by a four-tuple x with

$$x = (i, j, k, l) \in \mathcal{L} := (\mathbb{Z}/n_t\mathbb{Z}) \times (\mathbb{Z}/n_s\mathbb{Z})^3.$$

The Dirac field describing the quarks lives on the lattice sites and is usually written $\psi(x)$. At each lattice site the field ψ consists of a combination of 4 spin and 3 colour components resulting in 12 independent complex variables. Thus, ψ is a function $\psi : \mathcal{L} \to \mathbb{C}^{12}$ with $x \mapsto \psi(x)$. Moreover, we define $\psi_\sigma(x) \in \mathbb{C}^3$ to contain only the three colour components at the lattice site x for spin index $\sigma \in \{0, 1, 2, 3\}$.

The gluons on the other hand live on the links between neighbouring lattice sites. To describe the coupling between the lattice sites we need a few more definitions.

First, we define directions $\mu_i \in \mathcal{L}$ for $i = 0, \dots, 3$ on the lattice as

$$\begin{aligned} \mu_0 &= (1, 0, 0, 0), & \mu_1 &= (0, 1, 0, 0), \\ \mu_2 &= (0, 0, 1, 0) \text{ and} & \mu_3 &= (0, 0, 0, 1). \end{aligned}$$

Therefore, the 8 neighbours of $x \in \mathcal{L}$ are the distinct points $x \pm \mu_i$ with $i = 0, \dots, 3$.

Second, the continuum gauge fields from QCD can be represented on the lattice by matrices $U_\mu^x \in SU(3)$, the so-called gauge links or link variables. Here, $SU(3)$ denotes the set of all unitary complex 3×3-matrices with determinant 1. Each U_μ^x links the lattice site x with $x + \mu$ and $(U_\mu^x)^{-1} = (U_\mu^x)^H$ links $x + \mu$ with x vice versa. The set

$$\mathcal{U} = \left\{ U_\mu^x : x \in \mathcal{L}, \mu \in \{\mu_0, \mu_1, \mu_2, \mu_3\} \right\} \tag{4.1}$$

containing all gauge links U_μ^x is called a *configuration.* In Figure 4.1 a two-dimensional slice of a lattice with the gauge links from the configuration $\mathcal{U}$ is shown to clarify the notation.

Third, the theory has to respect certain spin symmetries. These are modelled using the basis $\gamma_0, \gamma_1, \gamma_2$, and γ_3 of a vector space that forms a Clifford algebra, i.e.

$$\gamma_i \gamma_j + \gamma_j \gamma_i = \delta_{ij}.$$

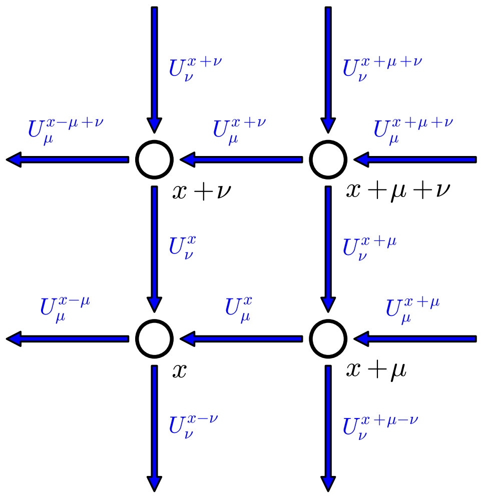


Figure 4.1: Depiction of a two-dimensional slice of a configuration $\mathcal{U} = \{U_\mu^x\}$ with $x \in \mathcal{L}$ and $\mu, \nu \in \{\mu_0, \mu_1, \mu_2, \mu_3\}$ used in lattice QCD.

The elements of this Clifford algebra can be represented as matrices $\gamma_i \in \mathbb{C}^{4\times 4}$ with

$$\gamma_0 := \begin{bmatrix} & & & +i \\ & & +i & \\ & -i & & \\ -i & & & \end{bmatrix}, \gamma_1 := \begin{bmatrix} & & & -1 \\ & & +1 & \\ & +1 & & \\ -1 & & & \end{bmatrix}, \gamma_2 := \begin{bmatrix} & & +i & \\ & & & -i \\ -i & & & \\ & +i & & \end{bmatrix}, \gamma_3 := \begin{bmatrix} & & +1 & \\ & & & +1 \\ +1 & & & \\ & +1 & & \end{bmatrix}$$

and we additionally define

$$\gamma_5 := \gamma_0\gamma_1\gamma_2\gamma_3 = \begin{bmatrix} +1 & & & \\ & +1 & & \\ & & -1 & \\ & & & -1 \end{bmatrix}.$$

Fourth, the continuum theory involves derivatives that can be discretised by finite differences. To this purpose we define the forward finite differences

$$(\Delta^\mu \psi_\sigma)(x) = \frac{U_\mu^x \psi_\sigma(x+\mu) - \psi_\sigma(x)}{a}$$

and the backward finite differences

$$(\Delta_\mu \psi_\sigma)(x) = \frac{\psi_\sigma(x) - (U_\mu^{x-\mu})^H \psi_\sigma(x-\mu)}{a}.$$

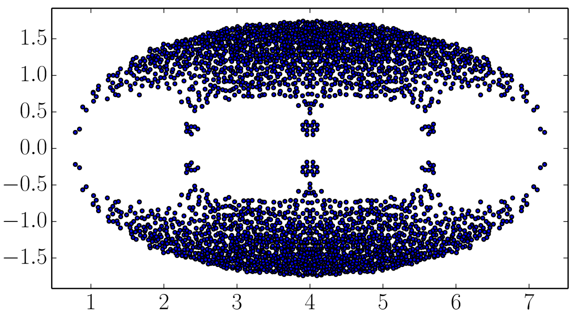
Herein, a denotes the lattice spacing, i.e. the physical distance between neighbouring lattice sites.

Finally, we can define the nearest neighbour coupling on the lattice. For this we use the *Wilson-Dirac operator* which is a discretisation of the continuum Dirac operator and is given by

$$\tilde{D}_W := \frac{m_0}{a} I_{12} + \frac{1}{2} \sum_{i=0}^{3} \Big(\gamma_i \otimes (\Delta_{\mu_i} + \Delta^{\mu_i}) - a I_4 \otimes \Delta_{\mu_i} \Delta^{\mu_i} \Big) . \tag{4.2}$$

Again, a is the lattice spacing and m_0 is a mass parameter which can be adjusted to tune the physical mass of the simulated quark flavour. This version of the Wilson-Dirac operator is not the only way to represent the continuum QCD theory on a lattice. For instance, the Wilson-Dirac operator can be improved to reduce the discretisation error from $\mathcal{O}(a)$ to $\mathcal{O}(a^2)$. Furthermore, there are some more continuum QCD properties that could be represented on the lattice. These may have advantages from the point of view of modelling in physics or numerical simulation, but describing them is out of scope of this thesis. Thus, here we only use the Wilson-Dirac operator $\tilde{D}_W$ from equation (4.2) and just note that in Section 4.1.2 and Section 4.1.4 other lattice operators could be used.

For practical purposes, the Wilson-Dirac operator $\tilde{D}_W$ can be represented by a matrix which takes the form $D_W \in \mathbb{C}^{n \times n}$ with $n = 12 n_t n_s^3$, see [Fro+13a]. In Figure 4.2a the spectrum of a Wilson-Dirac operator with $n_t = n_s = 4$ is displayed and the sparsity pattern of the associated matrix D_W can be found in Figure 4.2b. Note that for relevant quark masses m_0 all the eigenvalues of the Wilson-Dirac operator D_W have positive real part.


(a) Spectrum of D_W plotted in $\mathbb{C}$

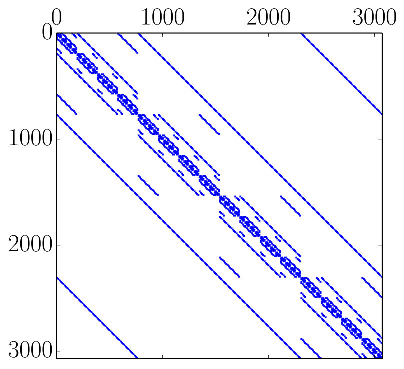

(b) Sparsity pattern of D_W

Figure 4.2: Example of a Wilson-Dirac operator D_W with $m_0 = 0$ and $n_t = n_s = 4$.

One important property of the Wilson-Dirac operator worth mentioning and used later is the so-called γ_5-*symmetry*. For $\tilde{D}_W$ it can be stated as

$$(\gamma_5 \tilde{D}_W)^H = \gamma_5 \tilde{D}_W.$$

For the associated matrix D_W with $\Gamma_5 := I_{n_t n_s^3} \otimes \gamma_5 \otimes I_3$ the symmetry

$$(D_W \Gamma_5)^H = D_W \Gamma_5 \tag{4.3}$$

holds.

In the following we present three applications in lattice QCD that need to solve systems with multiple right-hand sides, multiple shifts or both.

4.1.2 Hadron Spectroscopy

One of Lattice QCD's purposes is to compute observables in order to use the theory for predictions. Amongst these observables are the masses of hadrons like the mass of the neutron. Estimating these masses in lattice QCD computations goes under the name hadron spectroscopy. In computations of this kind so-called quark propagators have to be computed. Basically, these are the 12 columns of the inverse of the Wilson-Dirac operator D_W^{-1} belonging to a single lattice site m. Therefore, the task is to solve the block system

$$D_W \psi_{12} = \phi_{12}$$

where $\phi_{12} = e_m \otimes I_{12}$ and e_m is the m-th unit $n_t n_s^3$-vector. The $n \times 12$-matrix ϕ_{12} is called a point source.

Other computations of observables involve multiple random sources at random lattice sites, hence the system

$$D_W \psi_k = \phi_k$$

with $\psi_k, \phi_k \in \mathbb{C}^{n \times k}$ needs to be solved.

4.1.3 The Overlap Operator

Another important property of the continuum QCD theory is the so-called chiral symmetry for the massless operator D that can be expressed as

$$D\gamma_5 + \gamma_5 D = 0. \tag{4.4}$$

The Wilson-Dirac operator, however, does not fulfil this relation. Ginsparg and Wilson suggested [GW82] to relax the requirement (4.4) for an operator D_L on the lattice to

$$D_L \gamma_5 + \gamma_5 D_L = a D_L \gamma_5 D_L \tag{4.5}$$

which is called the Ginsparg-Wilson equation. For $a \to 0$ the right-hand side vanishes. In [Lüs98] Lüscher showed that (4.5) implies a symmetry which can be interpreted as a chiral symmetry on the lattice even for $a > 0$. An operator

$$D_{ov} = \frac{1}{a}(I + \Gamma_5 \text{sign}(\Gamma_5 D_W))$$

satisfying (4.5) was presented by Neuberger [Neu98; Neu00] and is called the *overlap operator*. Using the γ_5-symmetry (4.3) and the inverse square root relation of the sign function of equation (3.4) from Section 3.1.5 we get

$$\begin{aligned} D_{ov} &= \frac{1}{a}\left(I + \Gamma_5 \Gamma_5 D_W((\Gamma_5 D_W)^2)^{-1/2}\right) \\ &= \frac{1}{a}\left(I + D_W(\Gamma_5 D_W \Gamma_5 D_W)^{-1/2}\right) \\ &= \frac{1}{a}\left(I + D_W(D_W^H D_W)^{-1/2}\right). \end{aligned}$$

Since $\text{spec}\left(D_W^H D_W\right) \in \mathbb{R}^+$, the square root is well defined on the spectrum of $D_W^H D_W$ and could be computed with the methods presented in Chapter 3. Despite D_W being sparse, the matrix $\text{sign}(\gamma_5 D_W)$ in general is not. Thus, explicitly computing the matrix D_{ov} is impractical. The task in simulations though, is to solve systems

$$D_{ov}\psi = \phi. \tag{4.6}$$

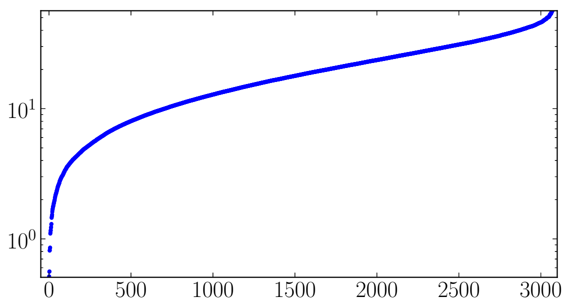
This can be achieved using an inner-outer scheme, thereby avoiding the need to compute D_{ov} explicitly. An outer iterative method building the Krylov subspace $\mathcal{K}_k(D_{ov}, \phi)$ is applied to solve (4.6). Whenever the outer method needs to perform a multiplication with the overlap operator, an inner iterative method is used to approximate the matrix-vector product

$$D_W(D_W^H D_W)^{-1/2}\chi = \tau. \tag{4.7}$$

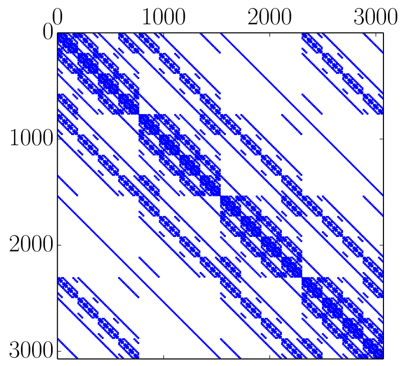
In Figure 4.3 we display the spectrum and sparsity pattern of $D_W^H D_W$ for the same Wilson-Dirac operator as in Figure 4.2. Note that in actual computations $D_W^H D_W$ is not computed explicitly. The matrix-vector product $D_W^H D_W \psi$ is computed via two successive matrix-vector products $D_W^H(D_W\psi)$ instead. Additionally, the identity $D_W^H \varphi = \Gamma_5 D_W \Gamma_5 \varphi$ can be used.

4.1.4 The Rational Hybrid Monte Carlo Algorithm

When it comes to dynamical simulations QCD shows its numerical complexity. Besides computing observables like in Section 4.1.2 the most time consuming



(a) Eigenvalues of $D_W^H D_W$, magnitude (vert. axis) vs. index (hor. axis)



(b) Sparsity pattern of $D_W^H D_W$

Figure 4.3: Example of $D_W^H D_W$ for a Wilson-Dirac operator D_W with $m_0 = 0$ and $n_t = n_s = 4$.

computation in lattice QCD is the generation of gauge configurations. This is done using the *Hybrid Monte Carlo (HMC) algorithm*. In this algorithm the system

$$(D_W^H D_W)\psi = \phi \tag{4.8}$$

for multiple random vectors ϕ has to be solved. If quarks of different mass are simulated then HMC cannot be applied [Cla06]. The *Rational Hybrid Monte Carlo (RHMC) algorithm* can be regarded as a generalisation of the HMC algorithm that is able to simulate multiple quark masses. In the RHMC algorithm the equation (4.8) is generalised to

$$(D_W^H D_W)^\alpha \psi = \phi \tag{4.9}$$

with $-1 \leq \alpha \leq 1$. Thus, we end up with a family of shifted systems with multiple random right-hand sides when for approximating $(D_W^H D_W)^\alpha$ a rational function is used, hence the name. More details of the RHMC algorithm can be found in [GL10; Ken06] and [Cla06].

4.2 Inverse Problems

The term *inverse problem* refers to mathematical problems that fall into two tightly linked categories. In the problems in the first category one tries to construct an object or its properties based on the desired outcome of measurements of the object. One example falling into this category is the computation of an aerodynamic shape like a wing meeting some specifications. While retaining constraints on the structure a solution maximising lift or minimising drag is sought [Dul+12]. The second category consists of problems where an existing object or its properties have to be reconstructed from observed measurements. The following examples are just a small selection from the vast amount of applications this category comprises:

- Measuring seismic waves in geophysical exploration, for example for determining the inner structure of a volcano [Isa98], called the inverse seismic problem.
- Using the information gathered in a couple of two-dimensional x-ray images to compute a three-dimensional model of the body in computed tomography [BP06].
- Finding obstacles by measuring the reflections of acoustic or electromagnetic waves in the inverse scattering problem [Isa98].

In general for inverse problems the equation

$$A(x) = b \tag{4.10}$$

is considered. Here, A represents a physical model like in the examples above, the right-hand side $b \in B$ is given for example by measurements and $x \in X$ is to be computed. The operator A is a continuous mapping from X to B which both are subsets of Banach spaces. But the important property of A, which renders solving system (4.10) difficult, is that A is not continuously invertible.

In many applications A can be linearised and discretised or is already given that form. Thus, from now on we assume the operator to be a linear map from $\mathbb{K}^n$ to $\mathbb{K}^m$, i.e. equation (4.10) becomes $Ax = b$ with $A \in \mathbb{K}^{m \times n}$, $x \in \mathbb{K}^n$ and $b \in \mathbb{K}^m$. Usually, the system $Ax = b$ is overdetermined or (close to) singular because of the lack of continuous invertibility of the underlying infinite-dimensional operator. Hence, an inverse operator A^{-1} does not exist or A is ill-conditioned. In the following we define well- and ill-posed problems and present one method to compute solutions to (4.10). For more details we refer to [Isa98]. Afterwards we finish this section with a deconvolution technique as an example application in image restoration.

4.2.1 Ill-Posed Inverse Problems

The next definition allows a classification of inverse problems.

Definition 4.1. *An inverse problem* (4.10) *is called* well-posed *if*

- *for all $b \in B$ a solution $x \in X$ exists,*
- *for any $b \in B$ this solution is unique and*

- $\|b - \tilde{b}\| \to 0$ *implies* $\|x - \tilde{x}\| \to 0$ *for any* $x \in X$ *and* $b \in B$.

If any one of these conditions is not fulfilled then the problem is called ill-posed.

In applications with data stemming from physical measurements it is the violation of the third condition that usually renders finding a solution for an ill-posed inverse problem non-trivial. The right-hand side b from measurements can be regarded as exact data $b^\star$ being perturbed by some error e, s.t. $b = b^\star + e$ holds. Disregarding this and using methods unaware of the nature of the problem might result in computing an approximate solution $\hat{x}$ that is far off the exact solution $x^\star$ for $Ax^\star = b^\star$. The cause of this is the amplification of the perturbation error during computations, which originates from the ill-conditioned nature of A. That said, special methods are needed that can find suitable approximate solutions to (4.10).

The idea for solving ill-posed inverse problems is to use regularisation, i.e. instead of solving $Ax = b$ we solve a modified equation. One commonly used method is the so-called *Tikhonov regularisation* in which the minimisation problem

$$x_\lambda = \underset{x}{\operatorname{argmin}} \left(\|b - Ax\|_2^2 + \lambda^2 \|x\|_2^2 \right) \tag{4.11}$$

has to be solved for some parameter λ. Therefore, we have to solve

$$(A^H A + \lambda^2 I) x_\lambda = A^H b \tag{4.12}$$

for x_λ with $\lambda \in \mathbb{R}$. It can be shown that x_λ converges to a solution x of (4.10) for $\lambda \to 0$ [Isa98, Lemma 2.3.2]. Solving (4.12) with an appropriate parameter λ amounts to a trade-off between having a small residual norm for (4.10) but an amplified perturbation error or having a larger residual norm whilst damping the perturbation error. In Figure 4.5 on page 64 at the end of this section we show the effect of a too small and too large regularisation parameter in an example of image reconstruction. A large residual norm can be regarded as an error introduced by the regularisation, hence called regularisation error.

The *L-curve* is a well-established criterion for choosing the regularisation parameter λ balancing both norms in (4.11) [Han00]. The origin of its name lies in the shape of the curve

$$\begin{aligned} \phi : \mathbb{R} &\to \mathbb{R}^2, \\ \lambda &\mapsto \phi(\lambda) = (\phi_1(\lambda), \phi_2(\lambda)) \end{aligned}$$

with $\phi_1(\lambda) = \|b - Ax_\lambda\|_2$ and $\phi_2(\lambda) = \|x_\lambda\|_2$ in a log-log plot looking similar to the letter L. As we can see in Figure 4.4 the L-curve has a distinct convex corner for a parameter λ_{LC}. The analysis of the L-curve in [Han00] using the

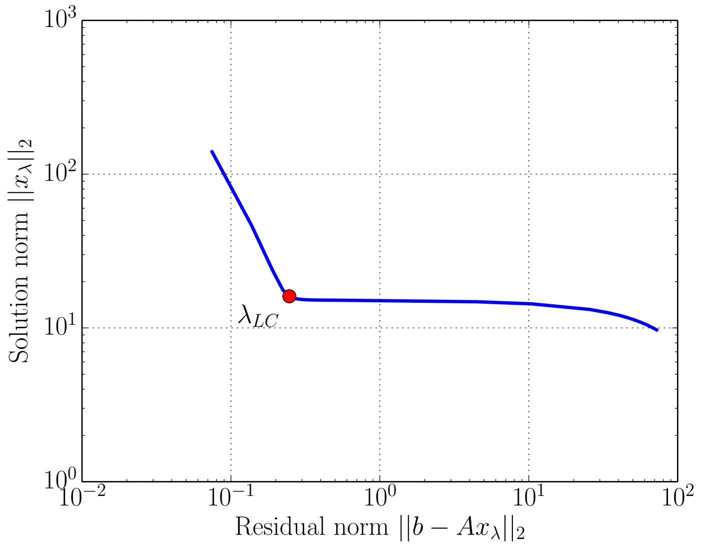


Figure 4.4: Example plot of residual norms plotted against solution norms for solutions x_λ for different regularisation parameters λ resulting in an L-curve.

singular value decomposition of A reveals that solutions belonging to smaller parameters $\lambda < \lambda_{LC}$ (vertical part of the L-curve) are dominated by perturbation errors. On the other hand, parameters $\lambda > \lambda_{LC}$ (horizontal part of the L-curve) correspond to solutions that are dominated by regularisation errors. Thus, for solving (4.10) we compute the solutions x_{λ_i} for the family of shifted systems (4.12) for multiple λ_i, e.g. $\lambda_i \in [10^{-6}, 10]$. Afterwards, the curve ϕ is approximated for example by spline interpolation using the points $\phi(\lambda_i)$. Then, e.g. the curvature of ϕ can be used to compute

$$\lambda_{LC} = \underset{\lambda}{\operatorname{argmax}} \left(\frac{\phi'_x(\lambda)\phi''_y(\lambda) - \phi'_y(\lambda)\phi''_x(\lambda)}{(\phi'_x(\lambda)^2 + \phi'_y(\lambda)^2)^{3/2}} \right). \tag{4.13}$$

Finally, equation (4.12) can be solved with $\lambda = \lambda_{LC}$ giving the solution x_{LC}.

4.2.2 Image Deconvolution

The method described up to here, i.e. Tikhonov regularisation using the L-curve criterion for determining an appropriate regularisation parameter, can be applied to an important image restoration technique called deconvolution. We will introduce briefly how digital images can be stored and describe how deconvolution can restore blurred images. Afterwards, we describe how the

Tikhonov regularisation can be applied to deconvolution and demonstrate it on a few examples.

The retina in the human eye consists of three types of cone cells. These contain distinct photoreceptor proteins resulting in maximum respondence to specific wavelengths of light. Thus, by a combination of three colours in different intensities it is possible to simulate almost all the colours the human eye can perceive. A convenient choice are the distinctive colours red, green and blue. So, a colour image of dimension $m \times n$ can be regarded as a map

$$\begin{aligned} P : M \times N \times C &\to \mathbb{R}, \\ (x, y, c) &\mapsto P_c(x, y) \end{aligned} \tag{4.14}$$

with $M := \{0, 1, \dots, m-1\}$, $N := \{0, 1, \dots, n-1\}$ and $C := \{r, g, b\}$. We call $P(x, y) := (P_r(x, y), P_g(x, y), P_b(x, y))$ a pixel at position (x, y). The three colour components of a pixel can take an intensity in the interval $[0, 1]$ each, so $P_c(x, y) \leq 0$ is interpreted ad the colour component c taking the lowest intensity whereas a value of ≥ 1 corresponds to maximum intensity. For example $P(x, y) = (0, 0, 0)$ corresponds to black, $P(x, y) = (1, 1, 1)$ to white and $P(x, y) = (1, 0, 0)$ to red. According to (4.14) we can represent P_c for a fixed colour c by a matrix

$$P_c \in \mathbb{R}^{m \times n}$$

with $x \in \{1, 2, \dots, n\}$ and $y \in \{1, 2, \dots, m\}$. If we have no colour information, i.e. just a greyscale image, we simply write P.

Images can be corrupted in many ways and even blur in an image can have multiple causes. One reason can be that the camera was moved during exposure of the picture which leaves distinctive artefacts and is called motion blur. Another cause for blur is optical aberration, i.e. in the optical system producing the image the light from one point of the object was not focussed on one single point of the imaging device. In a digital camera this might be caused by defocus. Regardless the cause, having a blurred image raises the question if the image can be enhanced or even more if the blurring can be reversed and a non-blurred image could be computed. Computationally cheap techniques like unsharp masking can be used to increase the apparent sharpness of a blurred image without trying to reconstruct the non-blurred original. In some applications, however, it is desirable to compute an approximation that is as close as possible to the non-blurred original even though it is computationally challenging. One use case was the distortion of early images of the Hubble Space Telescope that were less sharp than expected. The blurring was caused by a flawed mirror but could be partly compensated for by deconvolution methods until the repair of the mirror.

Some image corruption like blurring can be regarded as the result of a discrete convolution of the original uncorrupted image with a shift invariant kernel. In the case of blurring the kernel is called a *point spread function (PSF)*. The PSF K is given as a matrix $K \in \mathbb{R}^{m_k \times n_k}$ and a centre (m_c, n_c) with $m_c \in \{1, \dots, m_k\}$ and $n_c \in \{1, \dots, n_k\}$. A common choice is Gaussian blur where K is given by

$$K = (k_{g,h}) \text{ with } k_{g,h} = \frac{1}{2\pi\sigma^2} e^{-\frac{(g-m_c)^2+(h-n_c)^2}{2\sigma^2}} \tag{4.15}$$

and σ is the standard deviation of the Gaussian distribution. As an example with $m_k = n_k = 5$, $n_c = m_c = 3$ and $\sigma = 1$ we get

$$K = \begin{bmatrix} 0.0029 & 0.0131 & 0.0215 & 0.0131 & 0.0029 \\ 0.0131 & 0.0585 & 0.0965 & 0.0585 & 0.0131 \\ 0.0215 & 0.0965 & 0.1592 & 0.0965 & 0.0215 \\ 0.0131 & 0.0585 & 0.0965 & 0.0585 & 0.0131 \\ 0.0029 & 0.0131 & 0.0215 & 0.0131 & 0.0029 \end{bmatrix}.$$

For colour images the kernel K is applied to all colour channels P_c. Now, we can define the discrete convolution of a greyscale image $P = (p_{i,j}) \in \mathbb{R}^{m \times n}$ with the point spread function $K = (k_{i,j})$ as

$$(P * K) = B = (b_{i,j}) \text{ with } b_{i,j} = \sum_{\substack{g=1,\dots,n_k \\ h=1,\dots,m_k}} k_{g,h} \cdot p_{i+g-m_c, j+h-n_c}. \tag{4.16}$$

For deblurring purposes it turns out to be favourable to use *reflective boundary conditions*: We map those indices $i + g - m_c \notin \{1, 2, \dots, m\}$ and $j + h - n_c \notin \{1, 2, \dots, n\}$ respectively to the according 'reflected' pixel in the image. Thus, if $i + g - m_c < 1$ then we use $2 - (i + g - m_c)$ as index and if $i + g - m_c > m$ we use $2m - (i + g - m_c)$. The index $j + h - n_c$ is mapped in the same way.

If we reshape the image $P \in \mathbb{R}^{m \times n}$ to a vector $x = \text{vec}(P) \in \mathbb{R}^{mn}$ then the convolution in equation (4.16) represents a matrix-vector product and we refer to the convolution operator as a matrix A which in general is ill-conditioned. Hence, the convolution can be written as $Ax = b$ and deconvolution is the process of solving this system for a given reshaped blurred image $b = \text{vec}(B)$. The advantage of this approach over an FFT-based approach is that the kernel K as we introduced it not necessarily has to be uniform across the whole image so that a wider class of problems can be addressed.

Now, imagine that we start with a blurred colour image $B = [b_r | b_g | b_b]$ and we want to reverse the process that led to the blurred image. Note that we assume that the colour channels are already reshaped as vectors. In the best case the PSF of the blurring is known, e.g. from knowledge of the optical apparatus producing the image like for the Hubble Space Telescope. Otherwise, it might be estimated by analysing the blurred image(s) especially if motion blur caused the image corruption [ZŠM12], or it might simply be guessed. Anyhow, we end up with a matrix A representing the blurring operation if we assume that all colour channels are blurred in the same way. For simplicity, we assume this now, although it is not necessarily true, for example for chromatic aberration. So, to solve the deconvolution problem we need to solve the block system $AX = B$ where $X = [x_r|x_g|x_b]$ is the original non-blurred image. Since this is an ill-posed inverse problem we actually have to solve the family of shifted systems with multiple right-hand sides

$$(A^H A + \lambda^2 I)X_\lambda = A^H B \tag{4.17}$$

for a couple of parameters λ. Moreover, if we have multiple images of the same dimensions and which the same convolution operator A acted upon then the right-hand side in equation (4.17) consists of not just three columns. If we have n images we end up with B having $3n$ columns allowing to compute the optimal regularisation parameter for all images and colour channels at the same time.

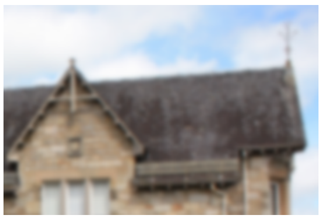
We finish this section with two examples for image deblurring. First, we demonstrate the effect of a too small and too large regularisation parameter in Figure 4.5. We applied a mild amount of Gaussian Blur (4.15) with $m_k = n_k = 11$ and $m_c = n_c = 6$ to the original image. As can be seen in the zoomed in detailed images a good choice of λ is able to remove most of the blur and reconstruct some details. If λ is chosen too small then the perturbation error is amplified and dominates the image. Photos taken with a digital camera always have a certain amount of noise resulting in some perturbation error. The noise originates from the construction of the sensor. On the other hand, if a too big λ is chosen then the computed image is too smooth and almost indistinguishable from the blurred image.

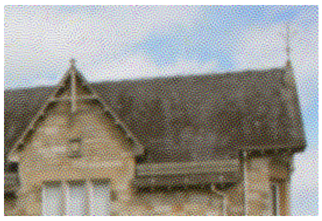
In Figure 4.6 we demonstrate the ability of the deconvolution technique described before to reconstruct an image in which the text was blurred beyond recognition. As it is pointed out in [Han00], the L-curve criterion sometimes chooses a regularisation parameter that is too small. Thus, the perturbation error still dominates the image. They also point out that this is mostly caused by too smooth right-hand sides. In our example we computed the optimal λ in the following way. First, we solved (4.17) for a range of λ from 10^{-6} to 10^1. For finding the corner in the L-curve we did not use (4.13). Instead, we just computed the angle between two consecutive line segments in the L-curve log-log plot. The largest of these angles was assumed to be the corner

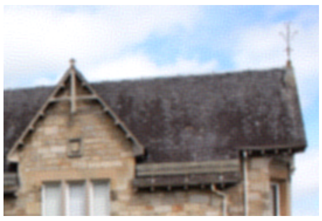
(a) Original

(b) Blurred

(c) Original details


(d) Blurred details


(e) λ too small


(f) Optimal λ

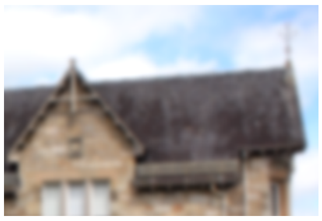
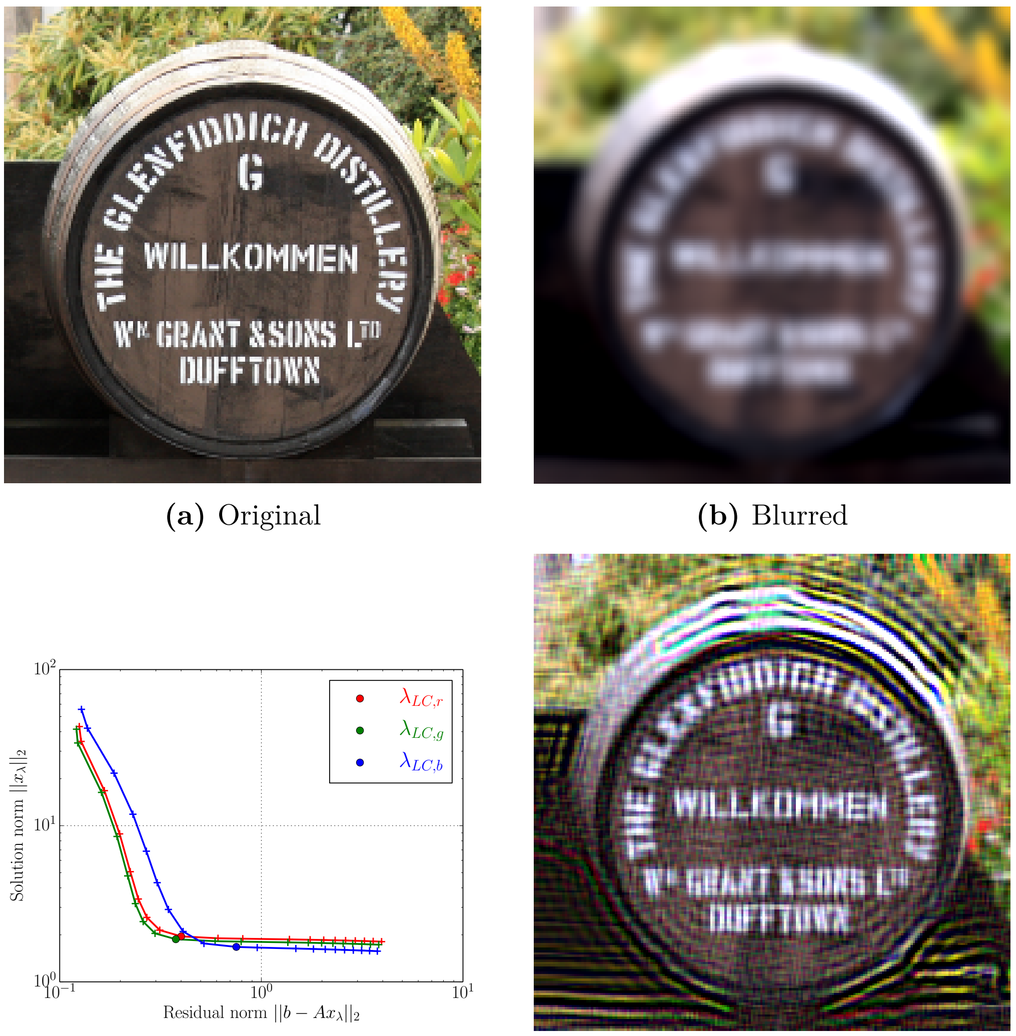

(g) λ too big

Figure 4.5: The effect of the regularisation parameter on the resulting image.

of the L-curve. After finding this corner we successively check the computed solutions x_λ for larger parameters λ for which the L-curve is still convex and stop as soon as it turns concave. This amounts to following the L-curve from the corner to the right into the horizontal part. In our experiments this turned out to result in a sufficiently good choice of λ.

A last aspect worth mentioning for deblurring is the problem of scaling the computed image $X = [x_r|x_g|x_b]$. Since the vectors x_r, x_g and x_b can contain outliers they should not be scaled s.t. the minimum maps to 0 and the maximum maps to 1. This would result in the relevant information to be squeezed into a narrow band of mid-luminosity. It turned out that a decent scaling can be achieved by computing the standard deviation and mean of the blurred image and scale X to have the same standard deviation and mean. The remaining values outside the interval $[0, 1]$ are treated as outliers and set to 0 and 1 respectively.



(a) Original

(b) Blurred

(c) Corresponding L-curves

(d) Reconstructed image

Figure 4.6: Deconvolution of an image containing text.

5 Krylov Subspace Methods for Shifted Systems

In Chapter 4 we have seen a few applications in which solving families of shifted linear systems with multiple right-hand sides is needed. One idea to solve this family of systems (1.1) is to solve them for every right-hand side separately. This leaves us with the task of solving the systems

$$(A + \sigma_i I)x_i = b \tag{5.1}$$

where $A \in \mathbb{C}^{n \times n}, x, b \in \mathbb{C}^n$, the shifts $\sigma_i \in \mathbb{C}$ and $i \in \{1, 2, \dots, s\}$. In this chapter we will present so-called *multi-shift methods*. The idea of these methods is to save computational work by using every matrix-vector multiplication with the matrix A for all s systems instead of solving the systems individually.

For ease of notation throughout most of this chapter we only use the unshifted system $Ax = b$ as well as one shift σ with the shifted system $(A+\sigma I)x = b$. Everything explained here can be trivially generalised to an arbitrary number of shifts. The algorithms in the following sections will use the s shifts $\sigma_1, \dots, \sigma_s$, though. This allows a discussion of their computational cost including the dependence on the number of shifts.

Before introducing specific methods we give some theoretical background on Krylov subspace methods for multi-shifted systems. For that matter, we recall the discussion in Section 2.2 that Krylov subspace methods can be defined by a sequence of polynomials and vice versa. In actual computations the polynomials are usually not constructed explicitly but they provide some insight into the multi-shifted methods. The polynomials $p_{k-1}(t) \in \Pi_{k-1}$ and $q_k(t) = 1 - tp_{k-1}(t) \in \overline{\Pi}_k$ define a method via

$$x^{(k)} = x^{(0)} + p_{k-1}(A)r^{(0)}$$

and

$$r^{(k)} = q_k(A)r^{(0)}$$

with $r^{(0)} = b - Ax^{(0)}$ and a starting vector $x^{(0)} \in \mathbb{C}^n$.

Suppose, now we want to solve the system

$$A_\sigma x_\sigma = b$$

with $A_\sigma = A + \sigma I$. Again, we have a starting vector $x_\sigma^{(0)}$, an initial residual $r_\sigma^{(0)} = b - A_\sigma x_\sigma^{(0)}$ and thus iterates

$$x_\sigma^{(k)} = x_\sigma^{(0)} + p_{\sigma,k-1}(A_\sigma) r_\sigma^{(0)}$$

and residuals

$$r_\sigma^{(k)} = q_{\sigma,k}(A_\sigma) r_\sigma^{(0)}$$

defined by polynomials $p_{\sigma,k-1}(t) \in \Pi_{k-1}$ and $q_{\sigma,k}(t) = 1 - t p_{\sigma,k-1}(t) \in \overline{\Pi}_k$.

The coefficients of the involved polynomials depend on the Krylov subspace method, the matrix and the initial residual. Thus, we cannot expect the polynomials p_{k-1} and $p_{\sigma,k-1}$ or q_k and $q_{\sigma,k}$ to be identical. However, if we require having starting vectors such that the initial residuals $r^{(0)}$ and $r_\sigma^{(0)}$ are collinear with

$$r_\sigma^{(0)} = \rho_\sigma^{(0)} r^{(0)} \tag{5.2}$$

then an observation of utmost importance leading to the development of multi-shift methods can be made. Namely, the Krylov subspaces $\mathcal{K}_k(A, r^{(0)})$ and $\mathcal{K}_k(A_\sigma, r_\sigma^{(0)})$ are identical for all k [PPV95], since

$$\begin{aligned} \mathcal{K}_k(A_\sigma, r_\sigma^{(0)}) &= \left\{x \in \mathbb{C}^n \,:\, x = p(A_\sigma) r_\sigma^{(0)}, p \in \Pi_{k-1}\right\} \\ &= \left\{x \in \mathbb{C}^n \,:\, x = \rho_\sigma^{(0)} p(A + \sigma I) r^{(0)}, p \in \Pi_{k-1}\right\} \\ &= \left\{x \in \mathbb{C}^n \,:\, x = p(A) r^{(0)}, p \in \Pi_{k-1}\right\} \\ &= \mathcal{K}_k(A, r^{(0)}). \end{aligned} \tag{5.3}$$

In other words, Krylov subspaces are *shift-invariant*. Equation (5.3) hints at the important fact that the bases generated with the Lanczos or Arnoldi process for $\mathcal{K}_k(A, r^{(0)})$ and $\mathcal{K}_k(A_\sigma, r_\sigma^{(0)})$ are identical when equation (5.2) holds. Thus, the initially mentioned idea of reusing the multiplications with the matrix A for all the shifted systems is backed by this.

We have seen that collinear initial residuals are vital for multi-shift methods exploiting the shift-invariant Krylov subspaces. Some methods even need to maintain the collinearity for the k-th residual, e.g. the restarted shifted GMRES method from [FG98]. What this means is that we have to have

$$r_\sigma^{(k)} = \rho_\sigma^{(k)} r^{(k)}. \tag{5.4}$$

Therefore, the polynomial $q_{\sigma,k}(t)$ defining our shifted method needs to satisfy

$$\rho_\sigma^{(0)} q_{\sigma,k}(A_\sigma) r^{(0)} = \rho_\sigma^{(k)} q_k(A) r^{(0)} \tag{5.5}$$

and $q_{\sigma,k}(0) = 1$. In [Jeg96] a common framework based on these shifted polynomials is investigated for some Krylov subspace methods like CG, BiCG and BiCGStab. The polynomial $q_{\sigma,k}$ and scalar $\rho_\sigma^{(k)}$ exist if and only if $q_k(-\sigma) \neq 0$ which was shown in [FG98, Lemma 2.1].

In Krylov subspace methods which impose the *Ritz-Galerkin condition*

$$r^{(k)} = q^{(k)}(A) r^{(0)} \perp \mathcal{K}_k(A, r^{(0)})$$

we have

$$r_\sigma^{(k)} = q_{\sigma,k}(A_\sigma) \rho_\sigma^{(0)} r^{(0)} \perp \mathcal{K}_k(A, r^{(0)}) = \mathcal{K}_k(A_\sigma, r_\sigma^{(0)}).$$

Thus, equation (5.5) holds without further ado, since $r^{(k)}, r_\sigma^{(k)} \in \mathcal{K}_{k+1}(A, r^{(0)}) \cap (\mathcal{K}_k(A, r^{(0)}))^\perp$. Moreover, since the shifted polynomials $q_{\sigma,k}$ naturally satisfy the condition (5.5) we are essentially applying the same Krylov subspace method to the shifted systems as we apply to the unshifted system. This, for example, is the case in the methods we will present in Section 5.1 and Section 5.2 which are both based on CG.

If, on the other hand, the Ritz-Galerkin condition does not hold true for the Krylov subspace method on which one wants to base a multi-shifted method then (5.5) will not hold in general. This is not a big deal as long as there is no need to require (5.4). Having collinear initial residuals would still allow for reusing the Krylov subspace $\mathcal{K}_k(A, r^{(0)})$ for the shifted system. But if for example the method needs to be restarted then it is essential that equation (5.4) holds. A GMRES based method that needs restarts to limit the storage requirements and cost of reorthogonalisation can be found, for instance, in [FG98]. It resolves the issue of collinear residuals after k steps by relaxing the condition of applying the same Krylov subspace method to the shifted systems and enforcing collinearity instead. This means that iterates are computed, such that collinear residuals can be maintained but for the shifted system they differ from those that would have been generated if the shifted system had solely been solved with the Krylov subspace method used for solving the unshifted system.

In the following sections of this chapter we will cover two CG based multi-shifted methods. There are a lot of other multi-shifted versions of methods like IDR [GSZ11], COCR [SZ11], restarted FOM [Sim03], and BiCGStab(ℓ) [Fro03] just to name a few.

We conclude this introduction with a few words on stopping criteria for multi-shift methods. The multi-shift methods presented in this chapter allow for accessing the norms of the iterates and residuals for both the shifted and unshifted systems which can be used as indicators when to stop the iteration. So in general a stopping criterion like $\|r_i^{(k)}\|_2 \leq \tau \|b\|_2$ for all shifts σ_i can be used. But, depending on the origin of the shifted systems (5.1) we might be interested in some special choice of τ. Especially, if the shifted systems stem from a partial fraction expansion like in (3.7) then it might be important to be able to use different stopping parameters τ_i for every shift. In this case we need to control two errors in the approximation

$$f(A)b \approx x^\star = \sum_{i=1}^{n} \omega_i (A - \sigma_i I)^{-1} b \approx \sum_{i=1}^{n} \omega_i x_i^{(k)} = x^{(k)}$$

to bound the error $\|f(A)b - x^{(k)}\|_2 \leq \epsilon$. First, we want to make sure that the partial fraction expansion belonging to the rational approximation to $f(A)$ satisfies

$$\|f(A)b - x^\star\|_2 \leq \frac{\epsilon}{2}.$$

Second, we want to have

$$\left\|x^\star - x^{(k)}\right\|_2 \leq \frac{\epsilon}{2}.$$

This can be broken down further to

$$\begin{aligned}\left\|x^\star - x^{(k)}\right\|_2 &= \left\|\sum_{i=1}^{n} \omega_i (x_i^\star - x_i^{(k_i)})\right\|_2 \\ &\leq \sum_{i=1}^{n} \omega_i \left\|x_i^\star - x_i^{(k_i)}\right\|_2 \\ &\leq \sum_{i=1}^{n} \omega_i \epsilon_i = \frac{\epsilon}{2}\end{aligned}$$

where $x_i^\star$ are the exact solutions to the individual shifted systems. So, for example the choice $\epsilon_i := \epsilon/(2n\omega_i)$ would guarantee this bound. Thus, altogether we would have

$$\left\|f(A)b - x^{(k)}\right\|_2 \leq \|f(A)b - x^\star\|_2 + \left\|x^\star - x^{(k)}\right\|_2 \leq \epsilon. \tag{5.6}$$

Unfortunately, the error $\left\|x^\star - x^{(k)}\right\|_2$ is not known. Therefore, one could go with the residuals and require

$$\begin{aligned}\|r_i^{(k)}\|_2 &\le \|A_{\sigma_i}^{-1}\|_2 \cdot \|e_i^{(k)}\|_2 \\ &\le \left\|A_{\sigma_i}^{-1}\right\|_2 \epsilon_i \\ &= \frac{\kappa(A_{\sigma_i})}{\left\|A_{\sigma_i}\right\|_2}\epsilon_i = \tau_i\end{aligned}$$

for equation (5.6) to hold. This is only feasible if $\kappa(A_{\sigma_i})$ and $\left\|A_{\sigma_i}\right\|_2$ are either known or good estimates are available. Otherwise, a common choice in practical computations is to relax on the bound (5.6) and require

$$\|r_i^{(k)}\|_2 \le \epsilon/(2n\omega_i) \tag{5.7}$$

as a stopping criterion.

5.1 Shifted CG

In the following we will describe one version of a shifted CG algorithm for solving the family of shifted systems (5.1) efficiently. The algorithm is based on [FM99] and assumes that A and $A + \sigma I$ are hpd. It uses a three-term recurrence for computing the Lanczos vectors unlike the CG Algorithm 2.4 formulated in Section 2.2.4 which uses a coupled two-term recurrence. This simplifies the understanding of the incorporation of the shifted systems.

5.1.1 Recurrences for the Unshifted System

Our derivation of the shifted CG variant below is based on the derivation of CG in Section 2.2.4 and closely related to the shifted CG in [FM99]. Here, we will cover the derivation of the recurrence coefficients in more detail than in Chapter 2. But first, we recall some notation introduced in Chapter 2, because in the following we will make extensive use of the Lanczos process and the entities

$$T^{(k)} = \begin{bmatrix} \alpha^{(1)} & \beta^{(1)} & & \\ \beta^{(1)} & \ddots & \ddots & \\ & \ddots & \ddots & \beta^{(k-1)} \\ & & \beta^{(k-1)} & \alpha^{(k)} \end{bmatrix}, \quad T^{(k+1,k)} = \begin{bmatrix} T^{(k)} \\ 0 \cdots 0 \; \beta^{(k)} \end{bmatrix}$$

and $V^{(k)} = [v^{(1)}| \ldots |v^{(k)}]$ therein. Further, we note that the computation of the CG iterates can be based on the Lanczos relation (2.11) resulting in

$$x^{(k)} = x^{(0)} + V^{(k)}(T^{(k)})^{-1} \left\|r^{(0)}\right\|_2 e_1.$$

Using the root-free Cholesky decomposition $T^{(k)} = L^{(k)}D^{(k)}(L^{(k)})^H$ and the definition $P^{(k)} := V^{(k)}(L^{(k)})^{-H}$ in Section 2.2.4 we came up with the recurrences for the vectors $p^{(k)}$ and $x^{(k)}$ for the unshifted system in the standard formulation of CG. Differently than in the standard formulation we can explicitly state the required scalars for the recurrences in terms of entries of $T^{(k)}$ by taking a closer look at the Cholesky decomposition. If we write

$$\begin{aligned}
T^{(k)} &= \begin{bmatrix} T^{(k-1)} & (t^{(k)})^H \\ t^{(k)} & \alpha^{(k)} \end{bmatrix} \\
&= L^{(k)}D^{(k)}(L^{(k)})^H \\
&= \begin{bmatrix} L^{(k-1)} & \\ \ell^{(k)} & 1 \end{bmatrix} \begin{bmatrix} D^{(k-1)} & \\ & d^{(k)} \end{bmatrix} \begin{bmatrix} (L^{(k-1)})^H & (\ell^{(k)})^H \\ & 1 \end{bmatrix} \\
&= \begin{bmatrix} L^{(k-1)}D^{(k-1)}(L^{(k-1)})^H & L^{(k-1)}D^{(k-1)}(\ell^{(k)})^H \\ \ell^{(k)}D^{(k-1)}(L^{(k-1)})^H & \ell^{(k)}D^{(k-1)}(\ell^{(k)})^H + d^{(k)} \end{bmatrix}
\end{aligned}$$

with $t^{(k)} = \begin{bmatrix} 0 & \cdots & 0 & \beta^{(k-1)} \end{bmatrix}$ and $\ell^{(k)} = \begin{bmatrix} 0 & \cdots & 0 & l^{(k-1)} \end{bmatrix}$ then we can deduce that

$$l^{(k-1)} = \frac{\beta^{(k-1)}}{d^{(k-1)}}$$

and

$$d^{(k)} = \alpha^{(k)} - d^{(k-1)} l^{(k-1)} \bar{l}^{(k-1)}$$

with $l^{(1)} = 1$ and $d^{(1)} = \alpha^{(1)}$. Thus, we have means to update the Cholesky decomposition easily from step $k-1$ to k. Moreover, we see that

$$\begin{aligned}
x^{(k)} &= x^{(0)} + V^{(k)}(T^{(k)})^{-1} \left\|r^{(0)}\right\|_2 e_1 \\
&= x^{(0)} + V^{(k)}(L^{(k)})^{-H}(D^{(k)})^{-1}(L^{(k)})^{-1} \left\|r^{(0)}\right\|_2 e_1 \\
&= x^{(0)} + \begin{bmatrix} P^{(k-1)} & p^{(k)} \end{bmatrix} (D^{(k)})^{-1}\tilde{u}^{(k)} \\
&= x^{(k-1)} + \frac{\tilde{u}_k^{(k)}}{d^{(k)}} p^{(k)}
\end{aligned}$$

with

$$\tilde{u}^{(k)} = (L^{(k)})^{-1} \left\|r^{(0)}\right\|_2 e_1$$

and we obtain

$$p^{(k)} = v^{(k)} - \bar{l}^{(k-1)} p^{(k-1)}.$$

Note that we can update $\tilde{u}^{(k-1)} = \begin{bmatrix} u^{(1)} & u^{(2)} & \cdots & u^{(k-1)} \end{bmatrix}^T$ to $\tilde{u}^{(k)}$ via

$$\begin{aligned} u^{(k)} &= -\ell^{(k)} \tilde{u}^{(k-1)} \\ &= -l^{(k-1)} u^{(k-1)} \end{aligned}$$

for $k > 1$ and $u^{(1)} = \left\| r^{(0)} \right\|_2$. We thus arrive at a non-standard formulation of CG which is nonetheless useful when it is extended for families of shifted systems.

5.1.2 Recurrences for the Shifted System

If the Lanczos process is applied to the shifted matrix $A_\sigma = A + \sigma I$ then because of the shift-invariance of Krylov subspaces it generates the same basis vectors $v^{(1)}, \dots, v^{(k)}$ for $\mathcal{K}_k(A_\sigma, b)$ as it would for $\mathcal{K}_k(A, b)$ and we have

$$\begin{aligned} (V^{(k)})^H A_\sigma V^{(k)} &= (V^{(k)})^H (A + \sigma I) V^{(k)} \\ &= (V^{(k)})^H A V^{(k)} + \sigma (V^{(k)})^H V^{(k)} \\ &= T^{(k)} + \sigma I =: T_\sigma^{(k)}. \end{aligned}$$

Then the relation

$$\begin{aligned} A_\sigma V^{(k)} &= V^{(k)} (T^{(k)} + \sigma I) + \beta^{(k)} v^{(k+1)} (e_k)^H \\ &= V^{(k+1)} T_\sigma^{(k+1,k)} \end{aligned}$$

holds with

$$T_\sigma^{(k+1,k)} := \begin{bmatrix} T_\sigma^{(k)} \\ 0 \cdots 0 \ \beta^{(k)} \end{bmatrix}.$$

Now the whole derivation of the iteration vectors and coefficients from above can be repeated based on $T_\sigma^{(k)}$. The entries of the root-free Cholesky decomposition $L_\sigma^{(k)} D_\sigma^{(k)} (L_\sigma^{(k)})^H$ of $T_\sigma^{(k)}$ can be updated using

$$\begin{aligned} l_\sigma^{(k-1)} &= \frac{\beta^{(k-1)}}{d_\sigma^{(k-1)}}, \\ d_\sigma^{(k)} &= \alpha^{(k)} + \sigma - d_\sigma^{(k-1)} l_\sigma^{(k-1)} \bar{l}_\sigma^{(k-1)} \end{aligned}$$

and

$$u_\sigma^{(k)} = -l_\sigma^{(k-1)} u_\sigma^{(k-1)}$$

with $l_\sigma^{(1)} = 1, d_\sigma^{(1)} = \alpha^{(1)} + \sigma$ and $u_\sigma^{(1)} = \left\|r_\sigma^{(0)}\right\|_2$. We end up with the vector recurrences

$$p_\sigma^{(k)} = v^{(k)} - \bar{l}_\sigma^{(k-1)} p_\sigma^{(k-1)}$$

and

$$x_\sigma^{(k)} = x_\sigma^{(k-1)} + \frac{u_\sigma^{(k)}}{d_\sigma^{(k)}} p_\sigma^{(k)}$$

for the shifted system.

5.1.3 A Feasible Stopping Criterion

Up till here the developed shifted CG-method is lacking a quantity that can be used as a feasible stopping criterion, though. But we can obtain the residual norm easily without actually computing the residual. For this we notice that by construction $r^{(k)} \in r^{(0)} + A\mathcal{K}_k(A, r^{(0)})$, $r^{(k)} \perp v^{(j)}$ for $j \leq k$, and $\left\|v^{(j)}\right\| = 1$. Thus, on the one hand we see that the residuals $r^{(k)}$ and $r_\sigma^{(k)}$ are *collinear* and on the other hand we have

$$\left\|r^{(k)}\right\|_2 = \left|(v^{(k+1)})^H r^{(k)}\right| \tag{5.8}$$

with

$$\begin{aligned}
(v^{(k+1)})^H r^{(k)} &= (v^{(k+1)})^H (b - Ax^{(k)}) \\
&= (v^{(k+1)})^H (b - A(x^{(0)} + V^{(k)} (T^{(k)})^{-1} \left\|r^{(0)}\right\| e_1)) \\
&= (v^{(k+1)})^H (r^{(0)} - AV^{(k)} (T^{(k)})^{-1} \left\|r^{(0)}\right\| e_1) \\
&= \underbrace{(v^{(k+1)})^H r^{(0)}}_{=0 \text{ for } k>0} - (v^{(k+1)})^H V^{(k+1)} T^{(k+1,k)} (T^{(k)})^{-1} \left\|r^{(0)}\right\| e_1 \\
&= -\left[0 \,|\, \dots \,|\, 0 \,|\, \beta^{(k)}\right] (T^{(k)})^{-1} \left\|r^{(0)}\right\| e_1 \\
&= -\beta^{(k)} e_k^H (L^{(k)})^{-H} (D^{(k)})^{-1} \tilde{u}^{(k)} \\
&= -\beta^{(k)} \frac{u^{(k)}}{d^{(k)}} \\
&= -l^{(k)} u^{(k)} \\
&= u^{(k+1)}.
\end{aligned}$$

Therefore, we have

$$r^{(k)} = u^{(k+1)} v^{(k+1)}$$

and the same holds for the residuals of the shifted system,

$$r_\sigma^{(k)} = u_\sigma^{(k+1)} v^{(k+1)}.$$

5.1.4 The Shifted CG Algorithm

Before stating the final algorithm we have to note one more thing. All the scalars $l^{(k)}, d^{(k)}$ and $u^{(k)}$ are real as well as their counterparts in the shifted case. This can easily be seen since the Lanczos quantities $\alpha^{(k)}$ and $\beta^{(k)}$ are real. Together with a real shift σ in the shifted case this implies that the Cholesky decomposition is real. Therefore, we have $\overline{l}_\sigma^{(k-1)} = l_\sigma^{(k-1)}$. This allows for a reformulation of the computations of

$$\begin{aligned} d^{(k)} &= \alpha^{(k)} - \beta^{(k-1)} l^{(k-1)} \text{ and} \\ d_\sigma^{(k)} &= \alpha^{(k)} + \sigma - \beta^{(k-1)} l_\sigma^{(k-1)}. \end{aligned}$$

With these observations we formulate the shifted CG algorithm in Algorithm 5.1. Note that we allow for an arbitrary number of shifts $\sigma_1, \dots, \sigma_s$. Despite computing the iteration coefficients differently the algorithm is in the spirit of [FM99].

The memory consumed by Algorithm 5.1 consists of $3 + 2s$ vectors of length n plus a couple of scalars. As for the computational complexity we have the following result.

Proposition 5.1. *Let k_i denote the number of iteration steps of CG applied to solve system i in the family of shifted systems* (5.1)*. Assume that in Algorithm 5.1 we only execute the loop in line 11 for those systems that are not yet converged. The total number of operations in Algorithm 5.1 for solving* (5.1) *is*

$$o_{\text{scg}} = \max_{i \in \{1,\dots,s\}} \{k_i\} \cdot (c_A + 4) + 2 \sum_{i=1}^{s} k_i. \tag{5.9}$$

Proof. The outer loop in line 5 is executed as long as there is an unconverged system, i.e. $\max_{i \in \{1,\dots,s\}} \{k_i\}$ times. In this loop we have one multiplication with A, the orthogonalisation against 2 previous vectors, a dot product and the normalisation for generating the next Lanczos vector resulting in $(c_A + 4)$ operations. The algorithm runs $\sum_{i=1}^{s} k_i$ times through the inner loop starting in line 11. In there, two vector updates of p and x are computed. All together, this amounts to o_{scg} vector operations. □

Now, we can compare the shifted CG algorithm for solving (5.1) to solving all s shifted systems separately with CG.

Proposition 5.2. *Let k_i denote the number of iteration steps of CG applied to solve system i in the family of shifted systems* (5.1)*. Let $a_k = \sum_{i=1}^{s} k_i$ be the overall number and $m_k = \max_{i \in \{1,\dots,s\}} \{k_i\}$ be the maximum number of*

Algorithm 5.1: Shifted CG

```
Input :  $A \in \mathbb{C}^{n\times n}$          system matrix
         $b \in \mathbb{C}^n$                   right-hand side
         $\sigma_1, \dots, \sigma_s$            shifts with $\sigma_i \in \mathbb{R}$, s.t. $A + \sigma_i I$ is hpd
         $x_1^{(0)}, \dots, x_s^{(0)} \in \mathbb{C}^n$   initial guesses yielding collinear residuals
Output:  approximate solutions $x_i^{(k)}$ to $(A + \sigma_i I)x_i = b$

1  $r_i^{(0)} = b - (A + \sigma_i I)x_i^{(0)}$ for $i = 1, 2, \dots, s$
2  $\beta_i^{(0)} = \left\| r_i^{(0)} \right\|_2$ for $i = 1, 2, \dots, s$
3  $v^{(0)} = 0, v^{(1)} = r_1^{(0)} / \beta_1^{(0)}$
4  $d_i^{(0)} = l_i^{(0)} = u_i^{(0)} = 0$ for $i = 1, 2, \dots, s$
5  for $k = 1, 2, \dots$ until convergence do
       // Lanczos step
6      $v = Av^{(k)} - \beta^{(k-1)} v^{(k-1)}$
7      $\alpha^{(k)} = \langle v, v^{(k)} \rangle$
8      $v = v - \alpha^{(k)} v^{(k)}$
9      $\beta^{(k)} = \|v\|_2$
10     $v^{(k+1)} = v / \beta^{(k)}$
       // CG iterates for all shifted systems
11     for $i = 1, 2, \dots, s$ do
12         $d_i^{(k)} = (\alpha^{(k)} + \sigma_i) - \beta^{(k-1)} l_i^{(k-1)}$
13         $u_i^{(k)} = \beta_i^{(0)} \delta_{1k} - l_i^{(k-1)} u_i^{(k-1)}$        // using the Kronecker delta
14         $p_i^{(k)} = v^{(k)} - l_i^{(k-1)} p_i^{(k-1)}$
15         $x_i^{(k)} = x_i^{(k-1)} + \frac{u_i^{(k)}}{d_i^{(k)}} p_i^{(k)}$
16         $l_i^{(k)} = \frac{\beta^{(k)}}{d_i^{(k)}}$
```

iteration steps. The number of vector operations in shifted CG is less than the number of vector operations in CG, hinting at shifted CG being faster than CG for solving all systems in (5.1)*, if*

$$(a_k - m_k)(c_A + 3) \geq m_k \tag{5.10}$$

holds.

Proof. Comparing the number of operations of shifted CG and CG yields

$$\begin{aligned} & a_k o_{\mathrm{cg}} \geq o_{\mathrm{scg}} \\ \Leftrightarrow\quad & a_k(c_A + 5) \geq m_k \cdot (c_A + 4) + 2a_k \\ \Leftrightarrow\quad & a_k(c_A + 3) \geq m_k \cdot (c_A + 4) \\ \Leftrightarrow\quad & (a_k - m_k)(c_A + 3) \geq m_k. \end{aligned}$$

□

The proposition suggests that shifted CG should be faster than CG in almost any situation, especially for large c_A. The left-hand side of the inequality (5.10) usually is much larger than the right-hand side. Thus, Proposition 5.2 backs the commonly made claim that when solving one of the systems in (5.1) the remaining ones come alongside almost for free by using shifted CG. Only if one system converges very slowly whilst all others converge very rapidly (5.10) will not hold. But this is seen rarely in applications.

There are numerous optimisations that can be applied to this basic version of shifted CG and we give only three of them now.

First, an obvious improvement that was already mentioned is to only execute the loop in the lines from 11 to 16 if the corresponding system is not yet converged. For our analysis in the previous two propositions we already assumed that this improvement is applied. We will see below that systems belonging to different shifts in general converge at different iteration steps which makes this a valuable improvement.

Second, coupled two-term recurrences for CG show superior numerical stability [GS00], as already mentioned in Chapter 2. This is where the shifted CG algorithm from [ES04] improves upon compared to Algorithm 5.1. However, the final algorithm (Algorithm 2) in that paper is a multi-shift CGLS variant especially tailored for problems of the kind $(A^H A + \sigma I)x_\sigma = A^H b$ like they arise in ill-posed inverse problems (see Section 4.2.1) making use of a CGLS-Lanczos variant. This is not immediately applicable to equation (5.1). Besides that, we remark that in [ES04] the stationary qd transformation (dstqds) from [DP03] is used to compute the iteration coefficients that turn out to be similar to the ones we derived above.

Third, this algorithm can be used as a starting point to develop a restarted shifted CG algorithm. This constitutes an optimisation in the sense that shifted systems stemming from a partial fraction expansion where the individual solutions x_i are not of primary interest can be solved more efficiently. We elaborate on this in more detail in Section 5.2.

5.1.5 Convergence of Shifted CG

Theorem 2.10 analysed the convergence behaviour of CG and stated that the error fulfils equation (2.21) which essentially reads

$$\frac{\|e^{(k)}\|_A}{\|e^{(0)}\|_A} \leq \frac{2}{\gamma^k + \gamma^{-k}},$$

where

$$\gamma = \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)$$

and $\kappa = \lambda_{max}/\lambda_{min}$ is the condition number of A. Since applying the shifted CG algorithm is mathematically the same as applying CG to all systems individually, i.e. computing identical iterates, the bound holds true for the shifted system $(A+\sigma I)x_\sigma = b$ with $\kappa_\sigma = \frac{\lambda_{max}+\sigma}{\lambda_{min}+\sigma}$ and the according errors $e_\sigma^{(k)}$ during the iteration. Furthermore, we have

$$\begin{aligned} & \kappa > \kappa_\sigma \\ \Leftrightarrow \quad & \frac{\lambda_{max}}{\lambda_{min}} > \frac{\lambda_{max}+\sigma}{\lambda_{min}+\sigma} \\ \Leftrightarrow \quad & 1 > \frac{\lambda_{max}\lambda_{min}+\sigma\lambda_{min}}{\lambda_{max}\lambda_{min}+\sigma\lambda_{max}} \\ \Leftrightarrow \quad & \sigma > 0 \end{aligned}$$

since $\lambda_{min} < \lambda_{max}$. Thus, the condition number of the shifted system is smaller than the one of the original system if $\sigma > 0$ and it is larger if $\sigma \in (-\lambda_{min}, 0)$. This has a profound impact on the speed of convergence as indicated by equation (2.21). Moreover, we can show in the next proposition that the 2-norm of residuals belonging to a shifted system with shift $\sigma > 0$ drops faster than the norm of the unshifted residuals.

Proposition 5.3. *Let $r^{(k)}$ and $r_\sigma^{(k)}$ denote the residuals computed by Algorithm 5.1 for a hpd matrix A, zero initial guesses and two shifts, 0 and $\sigma > 0$. Then*

$$\left\|r^{(k)}\right\|_2 > \left\|r_\sigma^{(k)}\right\|_2 \tag{5.11}$$

holds for $k \geq 1$.

Proof. For the initial residuals we have $\left\|r^{(0)}\right\|_2 = \left\|r_\sigma^{(0)}\right\|_2 = \|b\|_2$. From (5.8) we know that

$$\left\|r^{(k)}\right\|_2 = \left|l^{(k)}u^{(k)}\right| = \left|\frac{\beta^{(k)}u^{(k)}}{d^{(k)}}\right| \quad \text{and} \quad \left\|r_\sigma^{(k)}\right\|_2 = \left|l_\sigma^{(k)}u_\sigma^{(k)}\right| = \left|\frac{\beta^{(k)}u_\sigma^{(k)}}{d_\sigma^{(k)}}\right|.$$

We will show via induction that

$$u^{(k)} \geq u_\sigma^{(k)} > 0 \quad \text{and} \quad 0 < d^{(k)} < d_\sigma^{(k)}$$

for $k \geq 1$. But, first we note that since A is hpd so is $T^{(k)}$ and this implies that $\alpha^{(i)} > 0$ and $d_\sigma^{(i)} > 0$ for $i > 0$.

We start with $k = 1$, and using lines 12 and 13 of the algorithm yields

$$u^{(1)} = \beta^{(0)} = u_\sigma^{(1)} \quad \text{and}$$
$$d^{(1)} = \alpha^{(1)} < \alpha^{(1)} + \sigma = d_\sigma^{(1)}.$$

Since $\beta^{(0)} > 0$ and $\alpha^{(1)} > 0$ we have $u^{(1)} > 0$ and $d^{(1)} > 0$. Hence, $\left\|r^{(1)}\right\|_2 > \left\|r_\sigma^{(1)}\right\|_2$ holds.

Now, we assume that $u^{(k)} \geq u_\sigma^{(k)}$ and $d^{(k)} < d_\sigma^{(k)}$. Then we have

$$u^{(k+1)} = \frac{\beta^{(k)} u^{(k)}}{d^{(k)}} > \frac{\beta^{(k)} u_\sigma^{(k)}}{d_\sigma^{(k)}} = u_\sigma^{(k+1)}$$

as well as

$$d^{(k+1)} = \alpha^{(k+1)} - \beta^{(k)} l^{(k)} < \alpha^{(k+1)} + \sigma - \beta^{(k)} l_\sigma^{(k)} = d_\sigma^{(k+1)}$$

by noting that $l^{(k)} = \frac{\beta^{(k)}}{d^{(k)}}$ and $l_\sigma^{(k)} = \frac{\beta^{(k)}}{d_\sigma^{(k)}}$. This shows that (5.11) holds. $\square$

With Proposition 5.3 we have a theoretical background for the advantage of stopping the iteration individually for different shifts in Algorithm 5.1.

5.2 Restarted Shifted CG

Restarting CG does not look compelling at first. After all, CG has short recurrences in the first place and restarting is suspected to slow down convergence. When using, for example, GMRES it is immediately clear that restarting is needed to limit the recurrence length and amongst others the memory requirements. However, there is a good reason why one might come up with the idea of a restarted shifted CG. Again, in this section we assume that A and $A + \sigma I$ are hpd.

The algorithm that we work out below is equivalent to Algorithm 2 in [Afa+08] when applied to hpd matrices. The approach taken there does not start with solving multiple shifted linear systems with Krylov subspace methods. Instead, the authors improve on a previously proposed Krylov subspace algorithm for computing $f(A)b$, which required solving systems that grow with the number of restarts. In [Afa+08] the relation to CG is pointed out and in this section we will derive the algorithm that we call restarted shifted CG starting from shifted CG.

5.2.1 Idea of Restarted Shifted CG

We already got a glimpse of the idea in a remark to Algorithm 5.1. If the shifts in equation (5.1) stem from a partial fraction expansion which approximates a function f then one is probably not interested in the individual solutions x_i but rather the vector x in

$$\begin{aligned} f(A)b &\approx \pi(A)b + \sum_{i=1}^{s} \omega_j (A + \sigma_i I)^{-1} b \\ &= \pi(A)b + \sum_{i=1}^{s} \omega_i x_i \\ &= x. \end{aligned} \tag{5.12}$$

In the following, for simplicity of notation, we will omit the $\pi(A)b$-term. To see how x might be approximated more efficiently we need to take one step back and recall that we defined the CG iterates $x_i^{(k)}$ for an individual shift σ_i as

$$\begin{aligned} x_i^{(k)} &= x_i^{(0)} + V^{(k)} (T^{(k)} + \sigma_i I)^{-1} \left\| r_i^{(0)} \right\|_2 e_1 \\ &= x_i^{(0)} + V^{(k)} (L_i^{(k)})^{-H} (D_i^{(k)})^{-1} (L_i^{(k)})^{-1} \left\| r_i^{(0)} \right\|_2 e_1 \\ &= x_i^{(0)} + V^{(k)} y_i^{(k)} \end{aligned}$$

with

$$y_i^{(k)} = (L_i^{(k)})^{-H} (D_i^{(k)})^{-1} (L_i^{(k)})^{-1} \left\| r_i^{(0)} \right\|_2 e_1.$$

The vectors $y_i^{(k)}$ have length k and can be obtained via a simple forward and backward substitution intertwined with a scaling with the matrix $(D_i^{(k)})^{-1}$. In terms of computational cost this amounts more or less to three vector operations of size k per shift. So we can compute an iterate $x^{(k)}$ for $f(A)b$ as

$$\begin{aligned} x^{(k)} &= \sum_{i=1}^{s} \omega_i x_i^{(k)} \\ &= \sum_{i=1}^{s} \omega_i (x_i^{(0)} + V^{(k)} y_i^{(k)}) \\ &= \sum_{i=1}^{s} \omega_i x_i^{(0)} + V^{(k)} \sum_{i=1}^{s} \omega_i y_i^{(k)} \\ &= x^{(0)} + V^{(k)} \sum_{i=1}^{s} \omega_i y_i^{(k)}. \end{aligned} \tag{5.13}$$

For this we need to keep the Lanczos vectors $v_1, \dots, v_k$ as well as the matrices $L_i^{(k)}$ and $D_i^{(k)}$ for all s shifts. The structure of the latter two allows to store each as a vector of size not larger than k. Moreover, the recurrences from Section 5.1 for $l^{(k)}, d^{(k)}, u^{(k)}$ and their shifted equivalents can be used for cheap updates. This means that at the expense of storing k vectors of length n and $2s$ vectors of length k we obtain the iterate $x^{(k)}$ at the cost of only $s+k$ vector operations of size n.

Note that we can monitor the residuals of the shifted systems in exactly the same way as in the shifted CG algorithm even though we do not compute any of the individual iterates or residuals. Thus, along with the other values we compute $u^{(k)}$ for every shift which gives us a means to stop the iteration.

The above would make a nice method if it were feasible to keep the whole matrix $V^{(k)}$ until convergence. But at some point memory limits are reached and restarting the method is inevitable. As explained in Subsection 5.1.3 we have seen that shifted CG has the property of retaining collinear residuals. This can be exploited to restart the method by starting the Lanczos process anew with the vector $\tilde{v}^{(1)} = v^{(k+1)}$. Besides this, we need to take $x^{(k)}$ from equation (5.13) as the new starting vector $\tilde{x}^{(0)}$. Note that we do not need the individual iterates $x_i^{(k)}$. The new initial residuals for the individual systems are

$$r_i^{(k)} = u_i^{(k+1)} v^{(k+1)}.$$

We actually can do without the residual vectors. Only their norms are needed for the computation of the new $\tilde{y}_i^{(k)}$ and $\tilde{u}_i^{(k)}$.

5.2.2 The Restarted Shifted CG Algorithm

Summarising the above we formulate the restarted shifted CG algorithm in Algorithm 5.2.

The memory consumed by Algorithm 5.2 consists of $k+1$ vectors of length n (V and x) plus $3s$ vectors of length k (d, u and y) and a couple of scalars. The computational complexity is summarised in the next proposition.

Proposition 5.4. *Let $\hat{k}$ denote the number of iteration steps of Algorithm 5.2 applied to a family of shifted systems* (5.1) *for approximating* (5.12). *The total number of operations in the algorithms is*

$$o_{\text{rscg}} = s + \hat{k}(c_A + 5). \tag{5.14}$$

Proof. As explained in Proposition 5.1 we have $(c_A + 4)$ operations for generating the next Lanczos vector per step of the for-loop in line 7. Additionally,

Algorithm 5.2: RESTARTED SHIFTED CG (for matrix function evaluation)

```
Input  :  $A \in \mathbb{C}^{n\times n}$        system matrix
          $b \in \mathbb{C}^n$                   right-hand side
          $k$                                    Lanczos steps after which a restart is performed
          $\sigma_1, \dots, \sigma_s$            shifts with $\sigma_i \in \mathbb{R}$, s.t. $A + \sigma_i I$ is hpd
          $\omega_1, \dots, \omega_s$            weights with $\omega_i \in \mathbb{R}$
          $x_1^{(0)}, \dots, x_s^{(0)} \in \mathbb{C}^n$   initial guesses implying collinear residuals
Output:   approximation to $f(A)b \approx \sum_{i=1}^{n} \omega_i (A + \sigma_i I)^{-1} b$

1  $r_i^{(0)} = b - (A + \sigma_i I) x_i^{(0)}$ for $i = 1, 2, \dots, s$
2  $\beta_i^{(0)} = \left\| r_i^{(0)} \right\|_2$ for $i = 1, 2, \dots, s$
3  $d_i^{(0)} = l_i^{(0)} = u_i^{(0)} = 0$ for $i = 1, 2, \dots, s$
4  $v^{(0)} = 0, v^{(1)} = r_1^{(0)} / \beta_1^{(0)}$
5  $x^{(0)} = \sum_{i=1}^{s} \omega_i x_i^{(0)}$
6  for $c = 1, 2, \dots$ until convergence do
7      for $j = 1, 2, \dots, k$ do
               // Lanczos steps
8          compute $v^{(j+1)}, \alpha^{(j)}$ and $\beta^{(j)}$ as in Algorithm 5.1
               // iteration coefficients for all shifted systems
9          for $i = 1, 2, \dots, s$ do
10             $d_i^{(j)} = (\alpha^{(j)} + \sigma_i) - (l_i^{(j-1)})^2 d_i^{(j-1)}$
11             $u_i^{(j)} = \beta_i^{(0)} \delta_{1k} - l_i^{(j-1)} u_i^{(j-1)}$        // using the Kronecker delta
12             $l_i^{(j)} = \frac{\beta^{(j)}}{d_i^{(j)}}$
           // compute iterate and prepare for restart
13     $y_i^{(k)} = (L_i^{(k)})^{-H} (D_i^{(k)})^{-1} \left[ u_i^{(1)} \ \dots \ u_i^{(k)} \right]^T$
14     $x^{(c)} = x^{(c-1)} + V^{(k)} \sum_{i=1}^{s} \omega_i y_i^{(k)}$
15     $v^{(1)} = v^{(k+1)}$
16     $\beta_i^{(0)} = u_i^{(k+1)} = -l_i^{(k)} u_i^{(k)}$
```

in line 14 we have to compute the update for x using the k vectors $v^{(1)}, \dots, v^{(k)}$. All together this amounts to $k(c_A + 4) + k$ operations per restart or more conveniently $c_A + 5$ operations per generated Lanczos vector in average. Furthermore, in line 5 the starting iterate $x^{(0)}$ is computed via s vector operations. □

Although (5.14) seems to be independent of the *restart length k* it has a profound impact on the number of iteration steps $\hat{k}$. Since some generated subspace information is lost when a restart is performed, as a rule $\hat{k}$ increases as k decreases. This higher iteration count results in more computed Lanczos vectors of Algorithm 5.2 compared to Algorithm 5.1. Therefore, we need to compare the computational costs of Algorithm 5.1 with restarted shifted CG.

Proposition 5.5. *Let $\tilde{k}_i$ denote the number of iteration steps of CG applied to solve system i in the family of shifted systems* (5.1)*. Furthermore, let $\tilde{k} = \max_{i\in\{1,\dots,s\}} \left\{\tilde{k}_i\right\}$ and $\hat{k}$ denote the number of iteration steps of Algorithm 5.2. The number of vector operations in restarted shifted CG is less than the number of vector operations in shifted CG, hinting at restarted shifted CG being faster than shifted CG for approximating* (5.12)*, if*

$$1 + \frac{1 + 2\sum_{i=2}^{s}(\frac{\tilde{k}_i}{\tilde{k}})}{c_A + 5} \geq \frac{\hat{k}}{\tilde{k}} \tag{5.15}$$

holds.

Proof. We start with a trivial comparison

$$\max_{i\in\{1,\dots,s\}} \left\{\tilde{k}_i\right\} \cdot (c_A + 4) + 2\sum_{i=1}^{s} \tilde{k}_i + s \geq s + \hat{k}(c_A + 5)$$

of $o_{\text{scg}} + s$ from (5.9) and o_{rscg} from (5.14) and note that we have s additional operations for shifted CG since we need to compute a weighted sum of the individual solutions. Dividing by $\tilde{k}(c_A + 5)$ yields

$$\frac{c_A + 4 + 2\sum_{i=1}^{s}(\frac{\tilde{k}_i}{\tilde{k}})}{c_A + 5} \geq \frac{\hat{k}}{\tilde{k}}$$

resulting in (5.15). □

A large restart length k decreases the iteration steps $\hat{k}$ at the expense of higher memory requirements and ultimately can influence whether restarted shifted CG is faster than shifted CG according to (5.15). Thus, k should be chosen as large as the memory allows. Furthermore, we can see that as compared to shifted CG the restarted shifted CG methods benefits from a larger number of shifts and lower cost c_A.

In Algorithm 5.2 there is still room for some optimisations. Minor improvements can be achieved if a system belonging to a particular shift σ_i converges in step $j < k$ between restarts. Then updating its scalars after $d_i^{(j)}, u_i^{(j)}$ and $l_i^{(j)}$ can be stopped and $y_i^{(j)}$ can be computed immediately. A major issue of Algorithm 5.2 is that restarting deteriorates convergence, see Section 5.2.3. The runtime benefit comparing to shifted CG can vanish for short restart lengths since overall there might be more iteration steps needed. On the other hand, restart lengths providing no substantial slowdown might need infeasible restart lengths. In [EEG11] a remedy for this behaviour is proposed for methods based on the Arnoldi process. It is based on deflated restarts or sometimes called thick restarts. The idea there is to keep a few vectors carrying some eigenspace information generated from the Krylov subspace built up before the restart. It turns out that using those Ritz vectors belonging to Ritz values close to the shifts can improve convergence significantly.

5.2.3 Convergence of Restarted Shifted CG

The convergence of restarted shifted CG can again be characterised by Theorem 2.10, now applied to every restart cycle. If we perform a restart after k steps and number our iteration steps with c like in Algorithm 5.2 then we can bound the error $e^{(c)}$ belonging to iterate $x^{(c)}$ by

$$\left\|e^{(c)}\right\|_A \leq \left(\frac{2}{\gamma^k + \gamma^{-k}}\right)^c \left\|e^{(0)}\right\|_A, \tag{5.16}$$

where

$$\gamma = \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)$$

and $\kappa = \lambda_{max}/\lambda_{min}$ is the condition number of A. Furthermore, we can compare the error bound (2.21) for shifted CG after ck steps with the bound (5.16) for restarted shifted CG after c restarts of k steps each and can show that

$$\frac{2}{\gamma^{ck} + \gamma^{-ck}} \leq \left(\frac{2}{\gamma^k + \gamma^{-k}}\right)^c.$$

Hence, what we need to show here is

$$\frac{1}{\cosh(cky)} \leq \frac{1}{\cosh(ky)^c} \tag{5.17}$$

or equivalalently $\cosh(ky)^c \leq \cosh(cky)$ where we used $y = \cosh^{-1}(z)$ and $z = \frac{-\lambda_{max}-\lambda_{min}}{\lambda_{max}-\lambda_{min}}$. We can prove this via induction. First, we note that equation (5.17) is trivially true for $c = 1$. If we now assume that (5.17) holds for $c - 1$ then we get

$$\begin{aligned}
\cosh(ky)^c &= \cosh(ky)\cosh(ky)^{c-1} \\
&\leq \cosh(ky)\cosh((c-1)ky) \\
&\leq \cosh(ky)\cosh((c-1)ky) + \sinh(ky)\sinh((c-1)ky) \\
&= \cosh(ky + (c-1)ky) \\
&= \cosh(cky).
\end{aligned}$$

The first inequality holds because of the induction hypothesis and the second holds, since $y > 0$. Moreover, for $c > 1$ we even have $\cosh(ky)^c < \cosh(cky)$. This suggests that in general restarted shifted CG can take more steps for achieving convergence.

6 Krylov Subspace Methods for Multiple Right-Hand Sides

Many applications not only need to solve a system $Ax = b$ with a single right-hand side b but require solutions to multiple systems

$$Ax_j = b_j, \quad j = 1, \dots, m \tag{6.1}$$

for the same matrix A. For example, the systems (1.1) could be solved for every right-hand side separately resulting in (6.1). Here, we focus on the case that A is hpd. In some types of applications the right-hand sides are known in advance; in other applications the systems (6.1) can only be solved sequentially.

For small dense matrices in most cases it is advisable to compute a matrix decomposition of A once and use it for solving all systems in (6.1). This, however, is not always feasible if A is a large sparse matrix, since sparse matrix decompositions cannot always be applied successfully.

The family of systems (6.1) could be solved one-by-one using iterative solvers for every system $Ax_j = b_j$ separately. This, however, often is not a good choice, since information gathered while solving one system might be used to accelerate solving the other systems. Therefore, some effort has been put into the development of methods that are well-suited for (6.1).

In contrast to the shifted systems of Chapter 5 the family of systems (6.1) in general cannot be solved by building a common Krylov subspace for all right-hand sides. Hence, other techniques have to be applied for solving systems with multiple right-hand sides efficiently. This chapter covers Krylov subspace methods that can exploit the fact that we solve systems involving the same matrix A. We present them grouped in three sections representing different concepts.

First, Section 6.1 introduces *seed methods*. Seed methods solve the systems (6.1) sequentially. During the iteration of the current system they generate starting vectors for the remaining unsolved systems. Ideally, this creates good initial guesses resulting in increased speed of convergence. We focus on CG based seeding methods from [AMW08; CW97; SPM89].

Second, *deflation methods* introduced in Section 6.2 share the property of solving (6.1) for each system separately with seed methods. In contrast to seed methods, deflation methods try to gather some eigenspace information of the matrix A and use this to speed up subsequent solves for the remaining systems. Since the eigenspace information is independent from particular right-hand sides, deflation methods do not require all right-hand sides to be known in advance. As an example we introduce incremental eigCG [SO10].

Lastly, we describe *block methods* in Section 6.3. As opposed to the other two methods, most of them do not work on single vectors but group multiple vectors to blocks, hence the name. By doing so, block methods span a *block Krylov subspace*. The idea behind this is to have an enriched subspace to project onto. This can result in increased convergence speed for the individual systems. Besides this, block methods might be able to make better use of modern hardware by performing computations on matrices instead of vectors [GG08; KLL98]. We present BlockCG [OLe80] and some variants thereof [Dub01].

6.1 Seed Methods

As for all methods aimed at solving families of systems with multiple right-hand sides (6.1) the basic idea is to somehow reuse information generated during solving one of the systems for the remaining ones. Particularly, the rationale behind seed methods is that if the right-hand sides stem from a discretisation of a continuous problem and depend for example on a time step parameter, i.e. $b_j := b(t_j)$, then it is reasonable to assume that for small $\Delta t_j := t_{j+1} - t_j$ the vectors b_{j+1} and b_j and probably x_{j+1} and x_j are close too. As a first idea one could simply solve the m systems one at a time and use the solution of system j as an initial guess for system $j+1$. On the other hand, if there is some theoretical background available on the underlying problem then one could even extrapolate from the approximate solution of $Ax = b_j$ to the initial guess $x_{j+1}^{(0)}$, see, for instance, [Bro+97].

The *seed methods* that will be presented in this section pursue a slightly different approach. They reuse the whole Krylov subspace generated while solving one system of the family (6.1) to reduce the number of iterations for solving the remaining systems. This is done by performing orthogonal projections of the initial residuals of the unsolved systems onto the Krylov subspace that is spanned while solving the current system. The approximate solutions of the unsolved systems can then be updated accordingly. This concept can be applied recursively.

An immediate disadvantage of this approach is that we either need to know all right-hand sides (or, more precisely, at least the next right-hand side every time) in advance or we need to save the vectors that are used for the projections. The consequence of the latter would be high memory requirements for the storage. A more subtle disadvantage is how the projections affect the remaining systems. Since each generated vector that is spanning the Krylov subspace, is used for projecting the subsequent initial guesses and residuals, the contribution of every single one of these vectors is small. Even more so when the right-hand sides are not closely related. Then only minor improvements can be expected. This observation leads to *deflation based methods* which we will cover in Section 6.2.

In this section we will present two seed methods—the Single Seed Method from [SPM89] and Seed-CG from [AMW08; AMW13]. For non-Hermitian matrices there exist seed methods like the methods presented in [SG95; Smi87].

6.1.1 Single Seed Method

As a first seed method we present the Single Seed Method from [SPM89] which was analysed more thoroughly in [CW97]. It extends the CG algorithm, such that with every generated search direction $p_j^{(k)}$ of the seed system j the initial guess $x_i^{(0)}$ and residual $r_i^{(0)}$ of the non-seed systems $i > j$ are updated using

$$\begin{aligned}\tilde{x}_i^{(0)} &= x_i^{(0)} + \eta_{i,j} p_j^{(k)} \quad \text{and} \\ \tilde{r}_i^{(0)} &= r_i^{(0)} - \eta_{i,j} A p_j^{(k)}.\end{aligned}$$

The scalar $\eta_{i,j}$ is chosen such that the error $A^{-1}\tilde{r}_i^{(0)}$ is minimised in the A-norm in the search direction $p_j^{(k)}$. This can be achieved by an orthogonal projection of the residual onto the Krylov subspace of the seed system, i.e.

$$\begin{aligned} & 0 = \left\langle p_j^{(k)}, \tilde{r}_i^{(0)} \right\rangle = \left\langle p_j^{(k)}, r_i^{(0)} \right\rangle - \left\langle p_j^{(k)}, \eta_{i,j} A p_j^{(k)} \right\rangle \\ \Leftrightarrow \qquad & \eta_{i,j} = \frac{\left\langle p_j^{(k)}, r_i^{(0)} \right\rangle}{\left\langle p_j^{(k)}, A p_j^{(k)} \right\rangle}.\end{aligned}$$

The projections could also be based on the Lanczos vectors [PS90] or the residuals [Vor87] instead of the search directions.

In Algorithm 6.1 we state the algorithm from [SPM89] and [CW97] using our notation and the formulation of CG from Algorithm 2.4. In [SPM89] it is suggested to monitor the quantity

$$\log_{10} \left| \frac{\left\langle A p_j^{(k)}, A p_j^{(0)} \right\rangle}{\left\| A p_j^{(k)} \right\| \left\| A p_j^{(0)} \right\|} \right| \tag{6.2}$$

that is related to losing orthogonality. If (6.2) is significantly larger than machine precision then Algorithm 6.1 should be restarted by using the most recent iterate as a new initial guess. Moreover, when a restart happens a different system might be chosen as the new seed system. The authors in [SPM89] suggest to choose the "system with the worst error" which probably translates to largest residual norm. For simplicity of notation we did not incorporate these restarts and the dynamically switching of seed systems.

Algorithm 6.1: SINGLE SEED METHOD

```
Input :  $A \in \mathbb{C}^{n\times n}$, hpd                          system matrix
         $b_1, \dots, b_m \in \mathbb{C}^n$                        $m$ right-hand sides
         $x_1^{(0)}, \dots, x_m^{(0)} \in \mathbb{C}^n$            $m$ initial guesses

Output:  approximate solutions $x_j^{(k)}$ to $Ax_j = b_j$ for $j = 1, \dots, m$

1  $r_j^{(0)} = b_j - Ax_j^{(0)}$ for $j = 1, 2, \dots, m$
2  for $j = 1, \dots, m$ do                              // loop over the right-hand sides
       // w.l.o.g. choose $j$ as seed system
3      $p_j^{(0)} = r_j^{(0)}$
4      for $k = 1, 2, \dots$ until convergence of system $j$ do        // CG loop
5          $z_j^{(k-1)} = Ap_j^{(k-1)}$
6          $\mu_j^{(k-1)} = \frac{\langle r_j^{(k-1)}, r_j^{(k-1)} \rangle}{\langle p_j^{(k-1)}, z_j^{(k-1)} \rangle}$
7          $x_j^{(k)} = x_j^{(k-1)} + \mu_j^{(k-1)} p_j^{(k-1)}$
8          $r_j^{(k)} = r_j^{(k-1)} - \mu_j^{(k-1)} z_j^{(k-1)}$
9          for $i = j+1, \dots, m$ do                    // update the non-seed systems
10             $\eta = \frac{\langle p_j^{(k-1)}, r_i^{(0)} \rangle}{\langle p_j^{(k-1)}, z_j^{(k-1)} \rangle}$
11             $x_i^{(0)} = x_i^{(0)} + \eta p_j^{(k-1)}$
12             $r_i^{(0)} = r_i^{(0)} - \eta z_j^{(k-1)}$
13         $\nu_j^{(k-1)} = \frac{\langle r_j^{(k)}, r_j^{(k)} \rangle}{\langle r_j^{(k-1)}, r_j^{(k-1)} \rangle}$
14         $p_j^{(k)} = r_j^{(k)} + \nu_j^{(k-1)} p_j^{(k-1)}$
```

Algorithm 6.1 needs to store $2m$ vectors of length n—namely the residuals and iterates of all m systems—as well as two additional vectors z and p for the current seed system. For its computational complexity we have the following result.

Proposition 6.1. *Let k_j denote the number of iteration steps needed to solve system j with the Single Seed Method. Then the total number of operations in*

Algorithm 6.1 for solving (6.1) *is*

$$o_{\text{ssm}} = \sum_{j=1}^{m} k_j(\underbrace{c_A + 5}_{\textit{CG part}} + \underbrace{4(m-j)}_{\textit{seeding}}). \tag{6.3}$$

Proof. One iteration step of Algorithm 6.1 involves the $c_A + 5$ operations from (2.16) and the operations for seeding in the loop starting in line 9. Inside the loop two inner products and two axpy operations are computed. □

For CG applied to multiple right-hand sides sequentially the number of operations is

$$o_{\text{cg-mrhs}} = \sum_{j=1}^{m} \tilde{k}_j(c_A + 5) \tag{6.4}$$

where it takes $\tilde{k}_j$ steps to solve the j-th system. Note that solving the first system with Algorithm 6.1 takes exactly the same number of steps $\tilde{k}_1 = k_1$ as with ordinary CG.

In general, it is not possible to determine a priori how much computational time can be saved using seeding since the number of iteration steps depends on the right-hand sides. But we can analyse what influences the possible runtime benefit of the Single Seed Method over CG. For this the comparison of $o_{\text{cg-mrhs}}$ and o_{ssm}

$$\begin{aligned} & o_{\text{cg-mrhs}} \geq o_{\text{ssm}} \\ \Leftrightarrow \quad & \sum_{j=1}^{m} \tilde{k}_j(c_A + 5) \geq \sum_{j=1}^{m} k_j(c_A + 5) + 4\sum_{j=1}^{m} k_j(m-j) \\ \Leftrightarrow \quad & (c_A + 5)\sum_{j=2}^{m}(\tilde{k}_j - k_j) \geq 4\sum_{j=1}^{m} k_j(m-j) \\ \Leftrightarrow \quad & \sum_{j=2}^{m}(\tilde{k}_j - k_j) \geq \frac{4}{c_A + 5}\sum_{j=1}^{m} k_j(m-j) \end{aligned}$$

results in the following proposition.

Proposition 6.2. *The number of vector operations in the Single Seed Method is less than the number of vector operations in CG, hinting at the Single Seed Method being faster than CG for solving all systems in* (6.1)*, if*

$$\sum_{j=2}^{m}(\tilde{k}_j - k_j) \geq \frac{4}{c_A + 5}\sum_{j=1}^{m} k_j(m-j) \tag{6.5}$$

holds.

Note that the proposition relates the number of vector operations to the number of saved iteration steps $\sum_{j=2}^{m}(\tilde{k}_j - k_j)$. Based on the proposition, we can deduce that, in order for the Single Seed Method to still be faster than CG, the number of steps saved as compared to CG can be smaller if the multiplications with A are more expensive or fewer projections need to be done.

The convergence of the current seed system in the Single Seed Method is analysed in [CW97]. It is shown there that the convergence of the chosen seed system is better if it starts with an initial guess that was generated during the solve of previous seed systems than starting with a random vector. A convergence behaviour as if the extreme ends of the spectrum of A had been removed—yielding an improved effective condition number—can be achieved [CW97, Theorem 3.4].

Finally, we note that the name Single Seed Method originates from [CW97] where the authors introduce the name to differentiate the method from an additionally proposed Block Seed Method. The latter is a hybrid version of Algorithm 6.1 and block conjugate gradient methods presented in Section 6.3.

6.1.2 Seed-CG

The major concern when using the Single Seed Method from the previous section is that a lot of work is wasted if seeding only marginally improves the initial guess of the remaining systems. And indeed, observations back this claim at least for unrelated right-hand sides. Thus, it might be beneficial to limit the amount of seeding. This is where the Seed-CG method from [AMW08; AMW13] to be discussed now improves upon the Single Seed Method.

First, it is suggested in [AMW13] that seeding is done only for the first system to limit the work spent on seeding. But, we do not want to lose the benefit of seeding for related right-hand sides completely. One remedy presented in [Lan03] for seed-GMRES is a sliding window for the seed. This means that the initial guesses of only a fixed number of upcoming right-hand sides are updated by the seeding. This is thought to carry more information on related right-hand sides to later solved systems than just seeding with the first one whilst maintaining a moderate amount of additional work spent on seeding. In [AMW13], however, a different approach is pursued. In addition to seeding from the first system it is suggested to use the CG corrections, i.e. the difference of the computed solution and the initial guess, for projecting the remaining systems. Like in the Single Seed Method an orthogonal projection of the initial residuals of the subsequent systems onto the space spanned by the CG corrections is performed. In detail, after solving system j we compute new initial guesses $\tilde{x}_i^{(0)}$ and residuals $\tilde{r}_i^{(0)}$ for every system $i > j$. We use the *CG correction*

$$\Delta x_j^{(k)} = x_j^{(k)} - x_j^{(0)}$$

to update the initial guess $x_i^{(0)}$ which yields the new initial guess

$$\tilde{x}_i^{(0)} = x_i^{(0)} + \eta_{i,j} \Delta x_j^{(k)} \tag{6.6}$$

for system i. The corresponding residual is given as

$$\begin{aligned} \tilde{r}_i^{(0)} &= b_i - A(x_i^{(0)} + \eta_{i,j} \Delta x_j^{(k)}) \\ &= r_i^{(0)} - \eta_{i,j}(r_j^{(0)} - r_j^{(k)}). \end{aligned} \tag{6.7}$$

In the CG algorithm the iterates are generated so that the A-norm of the error is minimised which is done by enforcing orthogonal residuals. This idea can be applied here and to obtain an orthogonal projection the parameter $\eta_{i,j}$ needs to be computed using

$$\begin{aligned} 0 = \left\langle \Delta x_j^{(k)}, \tilde{r}_i^{(0)} \right\rangle &= \left\langle \Delta x_j^{(k)}, r_i^{(0)} \right\rangle - \eta_{i,j} \left\langle \Delta x_j^{(k)}, r_j^{(0)} - r_j^{(k)} \right\rangle \\ \Leftrightarrow \qquad \eta_{i,j} &= \frac{\left\langle \Delta x_j^{(k)}, r_i^{(0)} \right\rangle}{\left\langle \Delta x_j^{(k)}, r_j^{(0)} - r_j^{(k)} \right\rangle}. \end{aligned}$$

To avoid introducing yet another index here, we simply note that the vectors $x_i^{(0)}$ and $r_i^{(0)}$ might already have been updated during solves of preceding right-hand sides. The updates can stem from seeding with the first system or updates (6.6) and (6.7) of previously solved systems. Using $\Delta x_j^{(k)}$ instead of $x_j^{(k)}$ for the projection was found to be a better choice since it does not increase the influence of components belonging to small eigenvalues too much [AMW13]. Note that since $r_j^{(k)}$ belongs to a converged system $r_j^{(k)} = 0$ is assumed in [AMW08] which leads to some simplifications in the above formulae. In our numerical tests, however, we could not confirm that it is safe to assume $r_j^{(k)} \approx 0$ and used $r_j^{(k)}$ in the computations.

Second, in [AMW13] the first seed system is solved past convergence. The rationale behind this is to further reduce the influence of small eigenvalues in the later solved systems.

Third, in some examples reorthogonalisation was found to reduce the number of iteration steps for solving subsequent systems.

Algorithm 6.2: Seed-CG

Input :	$A \in \mathbb{C}^{n\times n}$, hpd	system matrix
	$b_1,\dots,b_m \in \mathbb{C}^n$	m right-hand sides
	$x_1^{(0)},\dots,x_m^{(0)} \in \mathbb{C}^n$	m initial guesses
	c	additional steps for the first system

Output: approximate solutions $x_j^{(k)}$ to $Ax_j = b_j$ for $j = 1,\dots,m$

```
$r_j^{(0)} = b_j - Ax_j^{(0)}$ for $j = 1, 2, \dots, m$
for $j = 1,\dots,m$ do                                  // loop over the right-hand sides
    $p_j^{(0)} = r_j^{(0)}$
    for $k = 1, 2, \dots$ until convergence of system $j$ or $c$ further steps for system $1$ do
        $z_j^{(k-1)} = Ap_j^{(k-1)}$
        $\mu_j^{(k-1)} = \frac{\langle r_j^{(k-1)}, r_j^{(k-1)}\rangle}{\langle p_j^{(k-1)}, z_j^{(k-1)}\rangle}$
        $x_j^{(k)} = x_j^{(k-1)} + \mu_j^{(k-1)} p_j^{(k-1)}$
        $r_j^{(k)} = r_j^{(k-1)} - \mu_j^{(k-1)} z_j^{(k-1)}$
        if $j = 1$ then                                 // update the non-seed systems
            for $i = 2,\dots,m$ do
                $\eta = \frac{\langle p_j^{(k-1)}, r_i^{(0)}\rangle}{\langle p_j^{(k-1)}, z_j^{(k-1)}\rangle}$
                $x_i^{(0)} = x_i^{(0)} + \eta p_j^{(k-1)}$
                $r_i^{(0)} = r_i^{(0)} - \eta z_j^{(k-1)}$
        $\nu_j^{(k-1)} = \frac{\langle r_j^{(k)}, r_j^{(k)}\rangle}{\langle r_j^{(k-1)}, r_j^{(k-1)}\rangle}$
        $p_j^{(k)} = r_j^{(k)} + \nu_j^{(k-1)} p_j^{(k-1)}$
    $\Delta x_j^{(k)} = x_j^{(k)} - x_j^{(0)}$
    $\Delta r_j^{(k)} = r_j^{(0)} - r_j^{(k)}$
    for $i = j+1,\dots,m$ do                    // use solution to project remaining right-hand sides
        $\eta = \frac{\langle \Delta x_j^{(k)}, r_i^{(0)}\rangle}{\langle \Delta x_j^{(k)}, \Delta r_j^{(k)}\rangle}$
        $x_i^{(0)} = x_i^{(0)} + \eta \Delta x_j^{(k)}$
        $r_i^{(0)} = r_i^{(0)} - \eta \Delta r_j^{(k)}$
```

In Algorithm 6.2 we display the Seed-CG algorithm from [AMW13] without reorthogonalisation.

The computational cost for Algorithm 6.2 can be significantly lower than the cost of Algorithm 6.1 as we will see below. Again, Algorithm 6.2 needs to store $2m$ vectors of length n the residuals and iterates of all m systems as well as two additional vectors z and p for the current seed system which both can be reused for storing $\Delta x_j^{(k)}$ and $\Delta r_j^{(k)}$ after the convergence of system j. Its computational complexity is stated in the following proposition.

Proposition 6.3. *Let k_j denote the number of iteration steps needed to solve system j with the Seed-CG algorithm. Then the total number of operations in Algorithm 6.2 for solving* (6.1) *is*

$$o_{\text{seedcg}} = 4k_1(m-1) + 2m^2 + (c_A+5)\sum_{j=1}^{m} k_j. \tag{6.8}$$

Proof. Every iteration step of Algorithm 6.2 consists of the c_A+5 operations of the CG part. Computing the solution of the first system has the additional cost for seeding to the $m-1$ remaining right-hand sides—with 4 vector operations each in every iteration step. Moreover, after solving system j the initial guesses and residuals of the remaining $m-j$ systems are updated which involves $2+4(m-j)$ vector operations. All in all, we end up with

$$\begin{aligned} o_{\text{seedcg}} &= \underbrace{4k_1(m-1)}_{\text{seeding}} + \sum_{j=1}^{m}(k_j \underbrace{(c_A+5)}_{\text{CG part}} + \underbrace{2+4(m-j)}_{\text{lines 16 to 21}}) \\ &= 4k_1(m-1) + (c_A+5)\sum_{j=1}^{m} k_j + 2m + 2m(m-1) \\ &= 4k_1(m-1) + 2m^2 + (c_A+5)\sum_{j=1}^{m} k_j. \end{aligned}$$

□

Now, we can compare Seed-CG to CG with the same assumptions as for the Single Seed Method in the previous section. For simplicity we assume that the first system is not solved past convergence in Seed-CG. With

$$\begin{aligned} & o_{\text{cg-mrhs}} \geq o_{\text{seedcg}} \\ \Leftrightarrow\quad & (c_A+5)\sum_{j=1}^{m}\tilde{k}_j \geq 4k_1(m-1) + 2m^2 + (c_A+5)\sum_{j=1}^{m} k_j \\ \Leftrightarrow\quad & (c_A+5)\sum_{j=2}^{m}(\tilde{k}_j - k_j) \geq 4k_1(m-1) + 2m^2 \\ \Leftrightarrow\quad & \sum_{j=2}^{m}(\tilde{k}_j - k_j) \geq \frac{4k_1(m-1)+2m^2}{c_A+5} \end{aligned}$$

we obtain the following proposition which relates the number of saved iteration steps $\sum_{j=2}^{m}(\tilde{k}_j - k_j)$ to the number of vector operations.

Proposition 6.4. *If the inequality*

$$\sum_{j=2}^{m}(\tilde{k}_j - k_j) \geq \frac{4k_1(m-1) + 2m^2}{c_A + 5} \tag{6.9}$$

holds then Seed-CG needs less vector operations than CG which suggests that Seed-CG can be faster than CG.

Like the Single Seed Method the Seed-CG algorithm benefits from more expensive multiplications with A. Moreover, we can now compare the inequalities (6.5) for the Single Seed Method and (6.9) for Seed-CG. The largest term in the numerator of the right-hand side of (6.9) is of order $\mathcal{O}(k_1 m)$ whereas in (6.5) for the Single Seed Method it is of order $\mathcal{O}(k_j m^2)$ because of the sum. Thus, if Seed-CG is able to achieve almost the same drop in the number of iteration steps as the Single Seed Method then it can be faster than CG in some cases where the Single Seed Method is slower than CG. In other words, Seed-CG needs a smaller drop in iteration steps as the Single Seed Method to outperform CG.

6.2 Deflation Methods

The seed methods presented in the previous section have two major drawbacks.

First, they can only be applied if more than one right-hand side is available at a time. The only way around this limitation would be to store the generated vectors that span the Krylov subspace. Before the next system is solved we could use these vectors for projecting the initial guess. This is possible, albeit not advisable.

Second, every vector spanning the Krylov subspace of the current seed system is contributing to the initial guess of the subsequent systems. This contribution is expected to be small—even more so in the case of non-related right-hand sides.

When solving systems with non-related right-hand sides the idea of *deflation methods* we will describe below is more targeted than the idea for seed methods. In a nutshell, deflation consists of suitable projections involving acquired eigenspace information applied either to the initial guess or during the iteration [Saa$^+$00]. These aim to alleviate the effect of extreme eigenvalues that are assumed to slow down convergence. For example, for minimal residual methods it is shown in [EES00] that deflating with a nearly A-invariant subspace removes the undesired components from the residual almost completely.

Applying the projections, therefore, improves the effective condition number of the involved system.

Deflation methods are closely related to *augmentation methods* [Gau+13]. There, the Krylov subspace is augmented by a space that enriches the approximation space with information assumed to improve convergence. For example, this can be some eigenspace information on the matrix A obtained earlier. Notably, if *Ritz vectors* or *harmonic Ritz vectors* are used for augmentation then the generated augmented space is a Krylov subspace with respect to A and a different starting vector [SS07].

The first idea one might come up with is to explicitly compute a couple of eigenvectors $w_1, \dots, w_d$ belonging to the d smallest eigenvalues of the hpd matrix A. If a system $Ax = b$ with an initial guess $x^{(0)}$ is to be solved then one way to use the eigenvectors w_i is to apply an A-orthogonal projection and compute a new initial guess

$$\tilde{x}^{(0)} = x^{(0)} + U(U^H AU)^{-1} U^H (b - Ax^{(0)}) \tag{6.10}$$

where $U = [w_1| \dots |w_d]$. If we now start a Krylov subspace method with the initial guess $\tilde{x}^{(0)}$ instead of $x^{(0)}$ then we build a Krylov subspace with respect to A and $\tilde{r}^{(0)} = b - A\tilde{x}^{(0)}$ which in exact arithmetic implies

$$\operatorname{colspan}\{U\} \perp \mathcal{K}_k(A, \tilde{r}^{(0)}) \tag{6.11}$$

for all k. Note that with $\Pi_A = U(U^H AU)^{-1} U^H$ we have

$$\mathcal{K}_k(A, \tilde{r}^{(0)}) = \mathcal{K}_k(A(I - A\Pi_A), r^{(0)}),$$

since $A\Pi_A = \Pi_A A$ is an A-orthogonal projection with $(A\Pi_A)^2 = A\Pi_A$. In numerical computations it might be better to perform the projection in every step emphasised by building $\mathcal{K}_k(A(I - A\Pi_A), r^{(0)})$ to prevent the projected components to re-enter the Krylov subspace. A Krylov subspace method based on this deflated subspace would behave as if the matrix had a condition number $\frac{\lambda_{max}}{\lambda_{d+1}}$ instead of $\frac{\lambda_{max}}{\lambda_{min}}$.

Instead of using eigenvectors one could compute Ritz vectors $u_1, \dots, u_d$ approximating the d smallest eigenvalues of the hpd matrix A. If the initial guess is again computed as in (6.10) as suggested in [EG00; Saa87] then (6.11) and its implications do not hold anymore. However, even for approximated eigenvectors or in numerical computations, where the deflated vectors can reappear in the spanned Krylov subspace due to round-off, this can improve the convergence rate significantly.

A CG-based algorithm applying the above oblique projection technique for the initial guesses to CG is called Init-CG [EG00]. Some problems arise with this approach.

On the one hand, additional time has to be dedicated solely for computing the matrix U before solving $Ax = b$ can even be started. Thus, the break-even point for the additional work is hard to estimate. Of course, this is expected to be beneficial the more right-hand sides are considered. In this case we can compute U once and apply it to all systems in (6.1).

Furthermore, it is not immediately clear how accurately the eigenvector approximations in U have to be computed. As an example, in [GRT06] the convergence behaviour of Init-CG versus CG is compared when computing the eigenvectors in U to different accuracies ϵ. The examples there suggest that Init-CG converges more rapidly than CG until a backward error of ϵ is reached. After this, it stagnates and returns to the convergence curve of plain CG resulting in no actual gain in computational time.

Besides Init-CG, some other ideas exist for using U for deflation. For example, one could use U to project the residual vector in every step of the CG iteration or to construct a preconditioner. Each of these two ideas can be found in the two versions of the AugCG algorithm in [EG00]. Compared to Init-CG, these can show superior convergence behaviour but every step of the iteration is more costly.

Until now we have considered U to be static and computed in advance. Deflation methods in general can accelerate solving a single system $Ax = b$, but they need to obtain U first. In the case of multiple right-hand sides we are in a favourable position, because we can gather information for U while solving the systems (6.1) one after the other. This is the ansatz from [SO10] which we will describe in the following. There, the idea is to constantly improve U. For this, we will first present the eigCG algorithm followed by the incremental eigCG algorithm. An overview of other deflation methods can be found in [SS07] and an analysis of deflated CG is conducted in [KR12]. The most prominent one is deflated block GMRES from [DMW08; Mor05].

6.2.1 eigCG

The method from [SO10] that we want to present in this section is called eigCG. Its idea is to build up eigenvector approximations and collect them in the columns of the matrix U alongside an ordinary CG iteration without interfering with CG. This means that the CG iteration part is completely unaware of the additional computations. In fact, neither the search directions nor the residuals of eigCG deviate from the according vectors of CG. After the first system is solved U can be used to deflate the next system. For this, one could use the Init-CG approach described above to compute an initial guess for the next system. This method can now be repeated recursively to improve the eigenvector approximations in U and/or increase the number of approximated eigenvectors while solving every subsequent system and is then called incremental eigCG [SO10].

The main benefit of the eigCG approach is that the multiplications with the matrix A that CG has to perform anyhow can be reused by the eigensolver. This is done by incorporating the eigenspace computations directly into CG. Moreover, there is no need to decide how accurately the eigenvectors have to be computed except for maybe stopping the growth of U after solving a couple of systems from equation (6.1).

In the following, we will describe the eigCG and incremental eigCG methods in more detail and state the algorithms. As in [SO10] we incorporate the eigensolver part into our formulation of CG (Algorithm 2.4).

First of all, for the eigenvalue approximation part of eigCG we need some entities from the Lanczos process. Algorithm 2.4 is not using the vectors $v^{(1)}, \dots, v^{(k)}$ that are constructed in the Lanczos process explicitly. However, we can still retrieve them as the normalised residuals,

$$v^{(i)} = \frac{1}{\left\|r^{(i)}\right\|_2} r^{(i)}.$$

Furthermore, the matrix $T^{(k)}$ from the Lanczos process can be obtained from the CG scalars in Algorithm 2.4, see [Saa03, Chapter 6], as

$$T^{(k)} = \begin{bmatrix} \frac{1}{\mu^{(0)}} & \frac{\sqrt{\nu^{(0)}}}{\mu^{(0)}} & & \\ \frac{\sqrt{\nu^{(0)}}}{\mu^{(0)}} & \frac{1}{\mu^{(1)}} + \frac{\nu^{(0)}}{\mu^{(0)}} & \ddots & \\ & \ddots & \ddots & \frac{\sqrt{\nu^{(k-2)}}}{\mu^{(k-2)}} \\ & & \frac{\sqrt{\nu^{(k-2)}}}{\mu^{(k-2)}} & \frac{1}{\mu^{(k-1)}} + \frac{\nu^{(k-2)}}{\mu^{(k-2)}} \end{bmatrix} \tag{6.12}$$

One way of approximating eigenvalues and eigenvectors of A is the Rayleigh-Ritz procedure. If the columns of $V^{(k)} = [v^{(1)}| \dots |v^{(k)}]$ form an orthonormal basis of $\mathcal{K}_k(A, v^{(1)})$ then we can compute the Ritz-pairs of A with respect to $V^{(k)}$ using $T^{(k)} = (V^{(k)})^H A V^{(k)}$. Those Ritz-pairs $(\gamma_j, V^{(k)} y_j)$ with $j \in J \subseteq \{1, 2, \dots, k\}$ are selected whose Ritz-values are close to the eigenvalues we are interested in. The Ritz-vectors $V y_j$ can then be used as approximations to the eigenvectors of A belonging to those eigenvalues. If the orthonormal basis $\left\{v^{(1)}, \dots, v^{(k)}\right\}$ is generated by the Lanczos process then the described procedure is called the *Lanczos method*.

If we are interested—and for eigCG we are—in eigenvectors belonging to eigenvalues on either end of the spectrum of A then the Rayleigh-Ritz procedure can be a good choice since this method usually converges faster for non-interior eigenvalues. There is a catch, though. For good approximations the number of basis vectors k has to be rather large. Thus, in many applications the Lanczos method is impractical. Especially, if we consider a CG based method with short recurrences that normally does not need to store $V^{(k)}$ completely. This, however, can be remedied by using restarting techniques for the eigenvalue search space.

The choice of the restarting technique can have great influence on the convergence behaviour of the eigenvalues we are after. In [SS98] a number of restarting techniques has been compared and thick restarting was found to be efficient. Thick restarting means that more Ritz vectors are kept during a restart than we actually want to compute. This is backed by [MRD92] where computing the eigenvector belonging to the smallest eigenvalue—called *lowest eigenvector* from now on—was discussed. There, it is pointed out that if the lowest eigenvector is sought then the lowest Ritz vector from the last and the next to last step should be kept during a restart. This shows almost the same speed of convergence towards the eigenvector as a non-restarted Lanczos method. Restarting only with the last Ritz vector on the other hand can come with a significant deterioration of the speed of convergence.

Details of eigCG

Algorithm 6.3 presents the eigCG algorithm from [SO10]. Because of the eigensolver restarts, we need an additional iteration counter that is independent from the counter k of the unrestarted CG part in eigCG. Therefore, we introduce c for counting the eigensolver steps in Algorithm 6.3, which is increased in every iteration step but reset when a restart is performed.

Then, we select a restart length c_{rest} and a number n_{ev} of Ritz vectors to keep during a restart with $c_{\text{rest}} > 2n_{\text{ev}}$. In every step k of the CG method we compute the Lanczos vector $v^{(k)}$ from $r^{(k)}$. For the corresponding counter c this is used to extend $V^{(c)}$ and the matrix $T^{(c)} = (V^{(c)})^H AV^{(c)}$ using (6.12) (lines 11, 12 and 22).

If c reaches c_{rest} we restart by applying the Rayleigh-Ritz procedure and compute the n_{ev} lowest eigenvectors of $T^{(c)}$ and $T^{(c-1)}$ (lines 14 and 15). The eigenvectors are then used to form an A-orthonormal basis for the $2n_{\text{ev}}$ Ritz vectors we want to use for the restart (lines from 16 to 20). Then we can reset c to $2n_{\text{ev}}$. Now, the only thing left to do for the restart is the computation of the missing orthogonalisation coefficients in row $2n_{\text{ev}} + 1$ of $T^{(2n_{\text{ev}}+1)}$ for the current residual $r^{(k)}$ with respect to the new basis. In principle, this would mean that we need to compute $(r^{(k)})^H AV^{(c)}$. But we can manage without the multiplication with A by noting that

$$Ap^{(k)} = Ar^{(k)} + \nu^{(k-1)} Ap^{(k-1)}.$$

Therefore, we can compute the coefficients in row $2n_{\text{ev}} + 1$ of the matrix $T^{(2n_{\text{ev}}+1)}$ efficiently (line 21). Note that $Ap^{(k)} = t^{(k)}$ can be reused in the next iteration step and does not account for an additional multiplication with the matrix A.

When CG has converged, the matrices $V^{(c)}$ and $T^{(c)}$ are used to compute n_{ev} Ritz pairs. These can be used for deflating all subsequent systems afterwards—either by the Init-CG approach or some other deflation technique.

Discussion of the eigCG algorithm

In the algorithm we initialise the matrix T as a $c_{\text{rest}} \times c_{\text{rest}}$-matrix as we would do in an actual implementation. We use $T^{(c)}$ as a short notation for $T_{1:c,1:c}$ in those places where we need to operate on a submatrix of T.

Algorithm 6.3 needs to store the residual $r^{(k)}$, the iterate $x^{(k)}$, the search direction $p^{(k)}$ as well as two temporary vectors $t^{(k-1)}$ and $t^{(k)}$. Moreover, the matrix V with at most c_{rest} columns resides in memory. This all amounts to $c_{\text{rest}} + 5$ vectors of length n and some storage of $\mathcal{O}(c_{\text{rest}}^2)$.

The computational complexity of Algorithm 6.3 is the sum of the computational complexity for the CG part and the eigensolver part. In the CG part we have the same operations as in the CG algorithm, i.e. a multiplication with A, three axpy operations for the vector updates and four inner products of which we can reuse two. The eigCG additions perform $2n_{\text{ev}}(c_{\text{rest}} + 1) + 1$ vector operations in the lines 19 ($2n_{\text{ev}}c_{\text{rest}}$) and 21 ($2n_{\text{ev}} + 1$). These operations occur the first time after c_{rest} iterations and then every $c_{\text{rest}} - 2n_{\text{ev}}$ iterations. We simplify this by assuming that restarts always happen after $\tilde{c}_{\text{rest}}$ iteration steps with $c_{\text{rest}} - 2n_{\text{ev}} \leq \tilde{c}_{\text{rest}} \leq c_{\text{rest}}$. If we assume further that eigCG performs k steps then we have an computational cost of

$$(2n_{\text{ev}}(c_{\text{rest}} + 1) + 1)\lfloor k/\tilde{c}_{\text{rest}} \rfloor \tag{6.13}$$

per iteration for the eigensolver part of eigCG. Based on (6.13), one could compare the computational cost of eigCG and CG. But we postpone this discussion to the end of Section 6.2.2 after we will have introduced incremental eigCG.

We end this subsection with a couple of remarks on eigCG. Algorithm 6.3 can be rearranged such that reusing some calculated vectors like $Ap^{(k)}$ in line 21 becomes easier. Reorthogonalisation could be implemented, e.g. during every

Algorithm 6.3: EIGCG

```
Input  : $A \in \mathbb{C}^{n\times n}$    system matrix
         $b \in \mathbb{C}^{n}$    right-hand side
         $x^{(0)} \in \mathbb{C}^{n}$    initial guess
         $n_{\mathrm{ev}}$    number of lowest eigenvectors to approximate
         $c_{\mathrm{rest}}$    restart length of the Lanczos method
Output: approximate solution $x^{(k)}$ to $Ax = b$
        approximations to the $n_{\mathrm{ev}}$ lowest eigenpairs in $V$ and $\Lambda$
1  $p^{(0)} = r^{(0)} = b - Ax^{(0)}$
2  $c = 1, \nu^{(-1)} = 0, \mu^{(-1)} = 1$
3  $V = 0 \in \mathbb{C}^{n\times c_{\mathrm{rest}}}, T = 0 \in \mathbb{C}^{c_{\mathrm{rest}}\times c_{\mathrm{rest}}}$
4  for $k = 1, 2, \dots$ until convergence do
       // CG iteration
5      $t^{(k-1)} = Ap^{(k-1)}$
6      $\mu^{(k-1)} = \frac{\langle r^{(k-1)}, r^{(k-1)}\rangle}{\langle p^{(k-1)}, t^{(k-1)}\rangle}$
7      $x^{(k)} = x^{(k-1)} + \mu^{(k-1)} p^{(k-1)}$
8      $r^{(k)} = r^{(k-1)} - \mu^{(k-1)} t^{(k-1)}$
9      $\nu^{(k-1)} = \frac{\langle r^{(k)}, r^{(k)}\rangle}{\langle r^{(k-1)}, r^{(k-1)}\rangle}$
10     $p^{(k)} = r^{(k)} + \nu^{(k-1)} p^{(k-1)}$
       // start of eigCG additions to CG
11     $T_{c,c} = \frac{1}{\mu^{(k-1)}} + \frac{\nu^{(k-2)}}{\mu^{(k-2)}}$
12     $V_{:,c} = r^{(k-1)} / \left\|r^{(k-1)}\right\|_2$
13     if $c = c_{\mathrm{rest}}$ then                         // restart of eigensolver
14         compute the $n_{\mathrm{ev}}$ lowest eigenvectors of $T^{(c_{\mathrm{rest}})}$ as the columns of $Y$
15         compute the $n_{\mathrm{ev}}$ lowest eigenvectors of $T^{(c_{\mathrm{rest}}-1)}$ as the columns of $\tilde{Y}$
16         orthogonalise $\begin{bmatrix} Y & \begin{matrix} \tilde{Y} \\ 0 \cdots 0 \end{matrix} \end{bmatrix}$ yielding $Q$
17         $H = Q^H T^{(c_{\mathrm{rest}})} Q$
18         compute the eigenpairs $(\lambda_i, z_i)$ of $H$ and set $Z = [z_1 | \cdots | z_{2n_{\mathrm{ev}}}]$
19         $V = V(QZ)$, $c = 2n_{\mathrm{ev}}$
20         $T^{(c)} = \mathrm{diag}\left(\lambda_1, \dots, \lambda_{2n_{\mathrm{ev}}}\right)$
21         $T_{c+1,1:c} = (T_{1:c,c+1})^H = (Ap^{(k)} - \nu^{(k-1)} t^{(k-1)})^H V / \left\|r^{(k)}\right\|_2$
22     else $T_{c+1,c} = T_{c,c+1} = -\sqrt{\nu^{(k-1)}} / \mu^{(k-1)}$
23     $c = c + 1$
24 compute the $n_{\mathrm{ev}}$ lowest eigenpairs $(\lambda_i, z_i)$ of $T^{(c)}$
25 return $x^{(k)}, V = V_{:,1:c} \cdot [z_1 | \cdots | z_{n_{\mathrm{ev}}}]$ and $\Lambda = \mathrm{diag}\left(\lambda_1, \dots, \lambda_{n_{\mathrm{ev}}}\right)$
```

restart of the eigensolver by orthogonalising the current residual $r^{(k)}$ against the columns of matrix V from line 19. This can improve the convergence of CG and resolve some issues with converged eigenvalues [SO10]. Instead of using CG with the Init-CG deflation approach for the remaining systems we can apply eigCG successively leading us to incremental eigCG which we will present in the next subsection. More details on the eigCG algorithm can be found in [SO10].

6.2.2 Incremental eigCG

As described before, the eigenvector approximations we obtained in the matrix V by running Algorithm 6.3 can be used to accelerate the solves of subsequent systems in (6.1). Of course, we are not bound to keep the space that we used for deflation unmodified for all remaining systems. Instead we could incrementally improve the space used for deflating the remaining systems. This can be useful if the lowest eigenpairs do not converge during one CG iteration or if the convergence is deteriorated by a larger number of small eigenvectors.

Essentially, the idea of the incremental eigCG algorithm proposed in [SO10, Section 4] is to obtain more Ritz vectors and improved approximations for the eigenvectors. This is thought to result in faster convergence than only solving the first system with eigCG and use other deflation techniques for the remaining systems. The algorithm solves $1 \leq m_{\text{eig}} \leq m$ of the systems in (6.1) with eigCG and uses CG with the Init-CG deflation approach for the remaining systems. This is meant to address the observation that the achievable reduction in iteration steps after solving a couple of systems stagnates. Every system is deflated with an accumulation of eigenvector approximations obtained from solving previous systems. These are simply collected in a matrix U that grows after each call to eigCG. The projection $H = U^H AU$ of the matrix A is updated for every n_{ev} new columns in U. Algorithm 6.4 states the incremental eigCG algorithm from [SO10, Section 4] using our notation.

Before analysing the computational cost of Algorithm 6.4 we give a few remarks. In experiments in [SO10] it was observed that non-converged Ritz vectors after one iteration of eigCG can converge after solving more systems with incremental eigCG. For the first system the computation of the initial guess in line 3 in an actual implementation is replaced by either setting $x_1^{(0)}$ to zero or to an actual initial guess that is passed to the algorithm.

Algorithm 6.4 needs to store the matrices W and U which amounts to $n_{\text{ev}} + m_{\text{eig}} n_{\text{ev}}$ vectors of length n plus the $(m_{\text{eig}} n_{\text{ev}})2$ entries of matrix H additionally to the requirements of the eigCG and CG calls. As in the Seed-CG section we

Algorithm 6.4: INCREMENTAL EIGCG

Input :	$A \in \mathbb{C}^{n\times n}$, hpd	system matrix
	$b_1, \dots, b_m \in \mathbb{C}^n$	m right-hand sides
	n_{ev}	number of lowest eigenvalues to approximate
	c_{rest}	restart length of the Lanczos method
	m_{eig}	number of systems to solve with eigCG

Output: approximate solutions $\hat{x}_j$ to $Ax_j = b_j$ for $j = 1, \dots, m$

```
1  $U_0 = [\,], H_0 = [\,]$
2  for $j = 1, \dots, m$ do                                   // loop over the right-hand sides
3      $x_j^{(0)} = U_{j-1} H_{j-1}^{-1} U_{j-1}^H b_j$
4      if $m \le m_{\text{eig}}$ then                         // solve the system and compute eigenvectors
5          call eigCG (Alg. 6.3) with $A, b_j, x_j^{(0)}, n_{\text{ev}}$ and $c_{\text{rest}}$
               yielding $V_j$, $M_j$ and approximate solution $\hat{x}_j$ to $Ax_j = b_j$
6          orthonormalise $V_j$ against $U_{j-1}$ yielding $\tilde{V}_j$
7          $W_j = A\tilde{V}_j$
8          $H_j = \begin{bmatrix} H_{j-1} & U_{j-1}^H W_j \\ W_j^H U_{j-1} & \tilde{V}_j^H W_j \end{bmatrix}$
9          $U_j = \left[U_{j-1} \,|\, \tilde{V}_j\right]$
10     else                                                   // solve system without computing eigenvectors
11         call CG (Alg. 2.4) with $A, b_j, x_j^{(0)}$
               yielding approximate solution to $Ax_j = b_j$
12         $U_j = U_{j-1}, H_j = H_{j-1}$
```

quantify the computational cost of Algorithm 6.4.

Proposition 6.5. *Let k_j denote the number of iteration steps needed to solve system j with the incremental eigCG algorithm. Then the total number of operations in Algorithm 6.4 for solving* (6.1) *is*

$$\begin{aligned} o_{\text{eigcg}} &= m_{\text{eig}} n_{\text{ev}} \left(c_A + 2m - m_{\text{eig}} - 1 + \frac{n_{\text{ev}}}{2}(3m_{\text{eig}} - 1) \right) \\ &\quad + (c_A + 5) \sum_{j=1}^{m} k_j + (2n_{\text{ev}}(c_{\text{rest}} + 1) + 1) \sum_{j=1}^{m_{\text{eig}}} \lfloor k_j / \tilde{c}_{\text{rest}} \rfloor . \end{aligned} \tag{6.14}$$

Proof. The first m_{eig} systems are solved with eigCG from Algorithm 6.3 and the remaining systems with CG. In both cases the initial guess is computed using the Init-CG deflation approach in line 3 of Algorithm 6.4. For the number of operations we obtain

$$
\begin{aligned}
o_{\text{eigcg}} =& \sum_{j=1}^{m_{\text{eig}}} \Bigg(\underbrace{2(j-1)n_{\text{ev}}}_{\text{line 3}} + \underbrace{2(j-1)n_{\text{ev}}^2}_{\text{line 6}} + \underbrace{c_A n_{\text{ev}}}_{\text{line 7}} + \underbrace{(j-1)n_{\text{ev}}^2 + n_{\text{ev}}^2}_{\text{line 8}} \Bigg) \\
&+ \underbrace{\sum_{j=1}^{m_{\text{eig}}} \left(k_j(c_A+5) + (2n_{\text{ev}}(c_{\text{rest}}+1)+1) \lfloor k_j/\tilde{c}_{\text{rest}} \rfloor \right)}_{\text{line 5 using (6.4) and (6.13)}} \\
&+ \sum_{j=m_{\text{eig}}+1}^{m} \Bigg(\underbrace{k_j(c_A+5)}_{\text{line 11}} + \underbrace{2m_{\text{eig}}n_{\text{ev}}}_{\text{line 3}} \Bigg) \\
=& \sum_{j=1}^{m_{\text{eig}}} \left((j-1)(2n_{\text{ev}}+3n_{\text{ev}}^2) + c_A n_{\text{ev}} + n_{\text{ev}}^2\right) + (c_A+5)\sum_{j=1}^{m} k_j \\
&+ \sum_{j=1}^{m_{\text{eig}}} (2n_{\text{ev}}(c_{\text{rest}}+1)+1) \lfloor k_j/\tilde{c}_{\text{rest}} \rfloor + (m-m_{\text{eig}})2m_{\text{eig}}n_{\text{ev}} \\
=& m_{\text{eig}} \left(n_{\text{ev}}(c_A + 2(m-m_{\text{eig}})) + n_{\text{ev}}^2 + \frac{1}{2}(2n_{\text{ev}}+3n_{\text{ev}}^2)(m_{\text{eig}}-1) \right) \\
&+ (c_A+5)\sum_{j=1}^{m} k_j + (2n_{\text{ev}}(c_{\text{rest}}+1)+1)\sum_{j=1}^{m_{\text{eig}}} \lfloor k_j/\tilde{c}_{\text{rest}} \rfloor. \qquad \square
\end{aligned}
$$

From Proposition 6.5 we can deduce that the computational cost mainly depends on the number of eigenvectors that are approximated. Compared to this the dependence on the restart length is negligible. Moreover, we see that we can effectively limit the most costly $\mathcal{O}(m_{\text{eig}}^2 n_{\text{ev}}^2)$ term by choosing m_{eig} not too large.

As we did for Seed-CG, we now compare incremental eigCG to solely running CG for all systems.

Proposition 6.6. *Let $\tilde{k}_j$ be the number of steps that CG performs for solving system j. The number of vector operations in incremental eigCG is less than the number of vector operations in CG, hinting at incremental eigCG being able to outperform CG for solving all systems in* (6.1)*, if the inequality*

$$
\begin{aligned}
\sum_{j=2}^{m} (\tilde{k}_j - k_j) \geq \frac{1}{c_A+5} \Bigg[& m_{\text{eig}} \left(c_A n_{\text{ev}} + n_{\text{ev}}^2 + (n_{\text{ev}} + \frac{3}{2}n_{\text{ev}}^2)(m_{\text{eig}}-1) \right) \\
& + (2n_{\text{ev}}(c_{\text{rest}}+1)+1) \sum_{j=1}^{m_{\text{eig}}} \lfloor k_j/\tilde{c}_{\text{rest}} \rfloor \Bigg]
\end{aligned}
\tag{6.15}
$$

holds.

Proof. We note that solving the first system with Algorithm 6.4 takes exactly the same number of steps as with ordinary CG, since the eigenvalue computations do not interfere with the CG part in Algorithm 6.3. Then, simply comparing $o_{\text{cg-mrhs}} \geq o_{\text{eigcg}}$ yields (6.15). □

According to the proposition, we can expect a faster runtime of incremental eigCG than CG if not too many eigenvectors are approximated, the matrix-vector product is expensive enough and the number of saved iteration steps $\sum_{j=2}^{m}(\widetilde{k}_j - k_j)$ is sufficiently large.

6.3 Block Methods

The methods for solving the family of systems (6.1) introduced in the previous two sections had one idea in common. They aimed at carrying some kind of information from solving one of the systems to subsequent ones thereby improving the speed of convergence of these subsequently solved systems. This section is dedicated to introduce so-called *block methods*. These methods generate iterates for all the systems in (6.1) at the same time by operating on blocks of vectors, which gives them their name. This allows for sharing information amongst the systems and somehow extends the idea of seed methods to a kind of all-to-all seeding. The idea of using a block of vectors instead of single vectors dates back to methods for computing eigenvalues and eigenvectors with the Lanczos method, see [CD74; GU77; PS79; Ruh79] amongst others. For solving linear systems the block idea was adopted in [OLe80] where block-versions of the BiCG and the CG method were developed.

Block methods, strictly speaking, do not build Krylov subspaces to compute iterates $x_j^{(k)}$ for solving (6.1). Instead, they augment the space in which $x_j^{(k)}$ is sought to include all the Krylov subspaces $\mathcal{K}_k(A, b_i)$, $i = 1, \dots, m$. Moreover, many block methods are not bound to operate on a single vector. These methods can work on a set of vectors simultaneously by combining them into columns of matrices. This favourable property, however, comes at the price of requiring that all right-hand sides are available at the same time. In some cases this limits the applicability of block methods.

As mentioned before, block methods combine vectors into matrices. Thus, we need a handy notation for working on a set of vectors. For this, we arrange the m right-hand sides and the corresponding solutions of the systems (6.1) as block-vectors

$$B = [b_1|\cdots|b_m] \in \mathbb{C}^{n\times m} \text{ and } X = [x_1|\cdots|x_m] \in \mathbb{C}^{n\times m}.$$

Now we can rewrite the family of systems (6.1) as a *block system*

$$AX = B. \tag{6.16}$$

When using block methods there are two major observations which could turn out as pitfalls if one is not aware of them.

The first one is the observation that block methods achieve gains in computational time with a certain—sometimes small—number of right-hand sides but tend to get more expensive as the number m of right-hand sides increases. This is caused by costs that are more or less proportional to the product of the number of right-hand sides and the iteration steps. Hence, the costs are increasing at least linearly with the number of right-hand sides. At the same time, the number of iteration steps often decreases tremendously for the first few right-hand sides but in general shows a saturation as the number of right-hand sides increases further. This can even cross the break-even point when comparing the runtime to non-block methods. Therefore, it is sometimes advisable to partition the m right-hand sides into a couple of block systems with less right-hand sides. However, there is no known criterion to decide a priori how many right-hand sides result in the most savings in computational time.

The second observation is that during the iteration some columns of the involved block-vectors, for instance the block-vector of residuals, can become (nearly) linearly dependent on the other columns. On the one hand, this is considered a favourable property since it allows for reducing the amount of work for solving equation (6.16). On the other hand, (nearly) linearly dependent columns of block-vectors can result in ill-conditioned or even singular intermediate matrices which—depending on the method—need to be inverted. Obviously, this is a situation that has to be dealt with. We will come back to the treatment of (almost) linearly dependent columns in Section 6.3.3.

Before introducing particular methods we will discuss some theoretical background for block methods. Afterwards, we will present some block methods based on CG—namely the block conjugate gradient algorithm [OLe80] and variants thereof [Dub01]—since our focus here lies on these CG based methods. Other block methods include block BiCG [OLe80], block QMR [FM97], multiple block GMRES variants [Mor05; SG96], Block BiCGStab [EJS03; TSK09] and block MINRES [Soo14] as well as three methods—BlQMR, BlBiORes, and BlBiOMin—in [Loh06]. Only a few of these methods address the potential problems with linearly dependent columns of the involved block-vectors. An overview of block methods and some theory can be found in [Gut06].

6.3.1 Block Krylov Subspaces

As already discussed, block methods do not compute the iterates for the individual systems using the Krylov subspaces from Definition 2.2. The next definition introduces the spaces needed to describe block methods.

Definition 6.7. *Let $A \in \mathbb{C}^{n\times n}$ and $B \in \mathbb{C}^{n\times m}$. The k-th* block Krylov subspace *generated by the matrix A and block-vector B is defined as*

$$\begin{aligned}\mathcal{K}_k^{\square}(A,B) &:= \operatorname{colspan}\left\{[\ B \mid AB \mid A^2B \mid \cdots \mid A^{k-1}B\ ]\right\} \\ &= \operatorname{span}\left\{b_1,\dots,b_m,Ab_1,\dots,Ab_m,A^2b_1,\dots,A^{k-1}b_m\right\}.\end{aligned}$$

Clearly, the Krylov subspaces $\mathcal{K}_k(A,b_j) = \operatorname{span}\left\{b_j, Ab_j,\dots,A^{k-1}b_j\right\}$ are contained in $\mathcal{K}_k^{\square}(A,B)$. Unless we have reached an invariant subspace, the dimension of each space $\mathcal{K}_k(A,b_j)$ is k. The dimension of the block Krylov subspace $\mathcal{K}_k^{\square}(A,B)$, however, can be smaller than mk. This is due to the fact that some of the subspaces $\mathcal{K}_k(A,b_j)$ can have non-trivial intersections even when all the b_j are linearly independent.

In the following we will introduce some theory on block Krylov subspaces and their properties that can be found, e.g. in [GS08; GS09]. Note, however, that we use a slightly different notation for the block Krylov subspace. In [GS09] an analogue of the grade of a Krylov subspace from Lemma 2.3 is defined in the following way.

Definition 6.8. [GS09] *The* block grade *of A with respect to B is the positive integer $g := g(A,B)$ defined by*

$$g(A,B) := \min\left\{k : \dim(\mathcal{K}_k^{\square}(A,B)) = \dim(\mathcal{K}_{k+1}^{\square}(A,B))\right\}.$$

We continue by stating some properties of block Krylov subspaces taken from [GS09].

Lemma 6.9. *Let $A \in \mathbb{C}^{n\times n}$ be non-singular, $B \in \mathbb{C}^{n\times m}$ and $g = g(A,B)$ from Definition 6.8. Then the following properties hold.*

1. *If $k \le g(A,B)$ then $k \le \dim(\mathcal{K}_k^{\square}(A,B)) \le km$.*
2. *$g(A,B) \le \max\limits_{j=1,\dots,m}\left\{g(A,b_j)\right\}$.*
3. *$\mathcal{K}_{g(A,B)}^{\square}(A,B) = \mathcal{K}_{g(A,b_1)}(A,b_1) + \cdots + \mathcal{K}_{g(A,b_m)}(A,b_m)$ and for all $k < g(A,B)$ we have $\mathcal{K}_{g(A,B)}^{\square}(A,B) \neq \mathcal{K}_k^{\square}(A,B)$.*
4. *$\mathcal{K}_{g(A,B)}^{\square}(A,B)$ is the smallest A-invariant subspace of $\mathbb{C}^n$ that contains b_j for all $j \in \{1,\dots,m\}$.*

5. $\mathcal{g}(A,B) = \min\{k : A^{-1}B \in \mathcal{K}_k^{\square}(A,B)\}$.

Proof. For the proofs of these properties we refer to [GS09]. $\square$

The following theorem is a block Krylov subspace analogue of Theorem 2.4 for Krylov subspaces.

Theorem 6.10. [GS09, Theorem 9] *Let* $A \in \mathbb{C}^{n\times n}$, $X^{(0)} = [x_1^{(0)}|\dots|x_m^{(0)}] \in \mathbb{C}^n$, $R^{(0)} = B - AX^{(0)}$ *and let* $\mathcal{g}$ *be as in Definition 6.8. If* A *is non-singular then*

$$x_j^\star \in x_j^{(0)} + \mathcal{K}_{\mathcal{g}}^{\square}(A, R^{(0)}) \text{ for all } j \in \{1,\dots,m\} \text{ and}$$
$$x_j^\star \notin x_j^{(0)} + \mathcal{K}_{\mathcal{g}-1}^{\square}(A, R^{(0)}) \text{ for one } j \in \{1,\dots,m\},$$

where $[x_1^\star|\dots|x_m^\star] = X^\star = A^{-1}B$ *denotes the exact solution to* $AX = B$.

Proof. See [GS09, Theorem 9]. $\square$

Theorem 6.10—not unlike its non-block analogue—states the rationale behind using block Krylov subspaces for solving the block system (6.16). And now we can define what we will be referring to as a block Krylov subspace method. Again, this follows Definition 2.5.

Definition 6.11. *A* block Krylov subspace method *for solving* $AX = B$ *is an iterative method that generates iterates* $x_j^{(k)}$ *which fulfil*

$$x_j^{(k)} \in x_j^{(0)} + \mathcal{K}_k^{\square}(A, R^{(0)}),$$

where $R^{(0)} = B - AX^{(0)}$, *and* $X^{(0)} = [x_1^{(0)}|\dots|x_m^{(0)}]$ *is the starting block-vector.*

For further use, let us introduce the notation $X^{(k)} = [x_1^{(k)}|\dots|x_m^{(k)}]$ to describe the block-vector of the k-th iterates.

6.3.2 Bases for Block Krylov Subspaces

Block Krylov subspace methods need to generate bases of block Krylov subspaces in the same way that Krylov subspace methods rely on building bases for Krylov subspaces. For example, we would like to build orthonormal basis vectors because of the favourable properties they possess. There exist straight forward generalisations of the Arnoldi process and the Lanczos process. These run into problems as soon as $\dim(\mathcal{K}_k^{\square}(A,B)) \neq km$. We present them here by assuming $\dim(\mathcal{K}_k^\square(A,B)) = km$ and will discuss the shortcomings in the case of $\dim(\mathcal{K}_k^\square(A,B)) < km$ later in one place in Section 6.3.3.

In Algorithm 6.5 we display the block Arnoldi process as it is stated for instance in [Saa03, Chapter 6]. It generates an orthonormal basis for $\mathcal{K}_k^\square(A,B)$ and is the block generalisation of Algorithm 2.1.

Algorithm 6.5: Block Arnoldi Process

```
Input  :  $A \in \mathbb{C}^{n\times n}$          system matrix
          $V^{(1)} \in \mathbb{C}^{n\times m}$    unitary starting block-vector
          $k$                                    number of steps to perform

Output:  $\{V^{(1)}, V^{(2)}, \dots, V^{(k)}\}$   orthonormal basis of $\mathcal{K}_k^\square(A,B)$

1 for $c = 1, 2, \dots, k$ do
2     $V = AV^{(c)}$
3     for $i = 1, 2, \dots, c$ do                   // modified block Gram-Schmidt
4         $H_{i,c} = (V^{(i)})^H V$
5         $V = V - V^{(i)} H_{i,c}$
6     $V^{(c+1)} H_{c+1,c} = V$                     // QR decomposition of $V$
```

Like in the non-block case the matrices obtained by the block Arnoldi process satisfy the *block Arnoldi relation*

$$A\boldsymbol{V}^{(k)} = \boldsymbol{V}^{(k)}\boldsymbol{H}^{(k)} + V^{(k+1)}H_{k+1,k}\boldsymbol{E}_k^H = \boldsymbol{V}^{(k+1)}\boldsymbol{H}^{(k+1,k)}. \tag{6.17}$$

Here, we used the n-by-km matrix $\boldsymbol{V}^{(k)} = [V^{(1)}|\dots|V^{(k)}]$, the $(k+1)m$-by-km *upper block Hessenberg matrix*

$$\boldsymbol{H}^{(k+1,k)} = \begin{bmatrix} H_{1,1} & H_{1,2} & \cdots & H_{1,k} \\ H_{2,1} & H_{2,2} & \ddots & H_{2,k} \\ & \ddots & \ddots & \vdots \\ & & H_{k,k-1} & H_{k,k} \\ & & & H_{k+1,k} \end{bmatrix},$$

the matrix $\boldsymbol{H}^{(k)} = (\boldsymbol{H}^{(k+1,k)})_{1:km,:}$ and $\boldsymbol{E}_k = e_k \otimes I_m$.

If the matrix A is Hermitian, we can derive a block Lanczos process like we did in Algorithm 2.2 for non-block systems. This block Lanczos process is presented in Algorithm 6.6 and originates from [GU77]. Note that we still assume $\dim(\mathcal{K}_k^\square(A,B)) = km$.

In the same way we obtained the block Arnoldi relation in equation (6.17), we see that the *block Lanczos relation*

$$A\boldsymbol{V}^{(k)} = \boldsymbol{V}^{(k)}\boldsymbol{T}^{(k)} + V^{(k+1)}T_{k+1,k}\boldsymbol{E}_k^H = \boldsymbol{V}^{(k+1)}\boldsymbol{T}^{(k+1,k)},$$

Algorithm 6.6: Block Lanczos Process

Input :	$A \in \mathbb{C}^{n \times n}$, hpd	system matrix
	$V^{(1)} \in \mathbb{C}^{n \times m}$	unitary starting block-vector
	k	number of steps to perform
Output:	$\{V^{(1)}, V^{(2)}, \dots, V^{(k)}\}$	orthonormal basis of $\mathcal{K}_k^{\square}(A, B)$

```
1 for $c = 1, 2, \dots, k$ do
2     $V = AV^{(c)} - V^{(c-1)}T_{c-1,c}$
3     $T_{c,c} = (V^{(c)})^H V$
4     $V = V - V^{(c)}T_{c,c}$
5     $V^{(c+1)}T_{c+1,c} = V$                    // QR decomposition of $V$
6     $T_{c,c+1} = (T_{c+1,c})^H$
```

for the matrices from Algorithm 6.6 holds. Additional to the matrices used for the block Arnoldi relation we need to define the block tridiagonal matrix

$$\boldsymbol{T}^{(k+1,k)} = \begin{bmatrix} T_{1,1} & T_{1,2} & & \\ T_{2,1} & T_{2,2} & \ddots & \\ & \ddots & \ddots & T_{k-1,k} \\ & & T_{k,k-1} & T_{k,k} \\ & & & T_{k+1,k} \end{bmatrix}$$

and the matrix $\boldsymbol{T}^{(k)} = (\boldsymbol{T}^{(k+1,k)})_{1:km,:}$.

We note that the matrices $T^{(c+1,c)}$ and $H^{(c+1,c)}$ are endowed with an upper triangular structure. In the case of the block Lanczos relation this implies that the matrices $T^{(c,c+1)}$ are lower triangular. Unlike in the non-block Arnoldi and Lanczos processes for $m > 1$ the matrices $\boldsymbol{T}^{(k)}$ and $\boldsymbol{H}^{(k)}$ can contain complex values. Only the entries on the diagonal and the m-th lower diagonal are guaranteed to be real values. Again, for $\boldsymbol{T}^{(k)}$ the same is true for the m-th upper diagonal.

In [Ruh79] an equivalent block Lanczos process is presented that constructs $\boldsymbol{V}^{(k)}$ vector-by-vector. This means, that $\boldsymbol{V}^{(k)}$ in contrast to Algorithm 6.6 is extended by one vector at a time until $\boldsymbol{V}^{(k+1)}$ is formed after m steps. The algorithm in [Ruh79] also features a technique to take care of (nearly) linearly dependent vectors. We will come back to this vector-by-vector approach in Chapter 7. But for the methods presented in this chapter we stick to extending the block Krylov subspace block-wise.

There also exist two-sided block Lanczos processes for the non-Hermitian case [Ali+00; Loh06]. The method from [Ali+00] will be explained in detail in Chapter 7.

6.3.3 The Need for Deflation

In the block Lanczos and block Arnoldi methods from Section 6.3.2 we assumed that $\dim(\mathcal{K}_k^{\square}(A,B)) = km$. But we cannot guarantee this prerequisite for every matrix A and block right-hand side B. In the following we will sometimes only refer to the block Lanczos process but everything said is true also for the block Arnoldi process.

If $\dim(\mathcal{K}_k^{\square}(A,B)) < km$ then the block Lanczos process faces a matrix V of rank less than m in some step. The QR decomposition of V prevents $V^{(k)}$ from being rank-deficient by guaranteeing a unity $n \times m$ factor $Q = V^{(k)}$. This means that we introduce a new Lanczos vector and do not generate bases of the block Krylov subspaces but have $\mathcal{K}_k^{\square}(A,B) \subset \operatorname{colspan}\left\{\boldsymbol{V}^{(k)}\right\}$ which is not a problem by itself. However, this new Lanczos vector is not orthogonal to $V^{(c)}$ for $c < k$ and we therefore lose the property $(\boldsymbol{V}^{(k)})^H \boldsymbol{V}^{(k)} = I$. Moreover, $(\boldsymbol{V}^{(k)})^H A \boldsymbol{V}^{(k)} = \boldsymbol{T}^{(k)}$ does not hold any longer and $\boldsymbol{T}^{(k)}$ can even become singular or ill-conditioned. An indicator for this is that the R factor of the QR decomposition, i.e. $T_{k+1,k}$, is singular.

In numerical computations $\dim(\mathcal{K}_k^{\square}(A,B)) < km$ is rarely seen and we face the similarly severe problem of ending up with an ill-conditioned matrix $T_{k+1,k}$. This leads to the same problems as described before. In any case Algorithm 6.6 would happily move along and build the matrices $\boldsymbol{V}^{(k)}$ as well as $\boldsymbol{T}^{(k+1,k)}$. This can raise problems for block Krylov subspace methods that are based on the block Lanczos process or the block Arnoldi process either explicitly or implicitly.

For example, consider using Definition 6.11 for computing iterates in the following way

$$x_j^{(k)} = x_j^{(0)} + \boldsymbol{V}^{(k)}((\boldsymbol{V}^{(k)})^H A \boldsymbol{V}^{(k)})^{-1}(\boldsymbol{V}^{(k)})^H R_{:,j}^{(0)}. \tag{6.18}$$

If $(\boldsymbol{V}^{(k)})^H A \boldsymbol{V}^{(k)} = \boldsymbol{T}^{(k)}$ does not hold, as described before, then (6.18) is impractical to obtain iterates.

Besides that, computing iterates as in (6.18) hints at how block methods can be more efficient than their non-block cousins. If during the QR decomposition in the creation of $\boldsymbol{V}^{(k)}$ some of the columns of V in Algorithm 6.5 of 6.6 are identified as linearly dependent then these columns do not contribute to the iterate $x_j^{(k)}$. And if we have columns that are almost linearly dependent then their contribution is negligible.

A common way to get rid of the numerical problems is to remove those vectors that have been identified as (almost) linearly dependent in the block Lanczos process or the block Arnoldi process. This approach stems from block

Lanczos methods for computing eigenvalues and eigenvectors. There, it is easier to remove undesired vectors than in block Krylov subspace methods for linear systems, since in the latter we have to take care how we can still compute approximations to the solution after removing a vector. In block Krylov subspace methods where $\boldsymbol{V}^{(k)}$ is not built explicitly the residual block-vector $R^{(k)} = B - AX^{(k)}$ can be monitored for (near) rank deficiency. In the block method context this removal of vectors is called *deflation*. Note that this is not to be confused with deflation methods from Section 6.2. Deflation, however, needs a thorough examination how the removal of vectors impacts the corresponding methods. We distinguish between *exact deflation* and *inexact deflation*.

Exact deflation describes the situation when we end up with the zero vector after orthogonalising against the previous basis vectors of the block Krylov subspace. In the block Lanczos process this would result in a rank-deficient matrix $T_{k+1,k}$. In numerical computations this is unlikely to be seen. Therefore, the condition for exact deflation should be relaxed. Instead of checking for the zero vector one should regard ending up with a vector of very small norm, i.e. close to machine precision, as exact deflation.

We use the term inexact deflation when a vector has small but not negligible norm after orthogonalisation. This is indicated by an ill-conditioned matrix $T_{k+1,k}$ in the block Lanczos process. This is a more delicate situation than exact deflation since we cannot easily remove this vector and its removal introduces a *deflation error* that limits the accuracy of the computed solution. Then again, keeping such a vector can introduce numerical instability. Typically, one chooses a tolerance for controlling when inexact deflation is applied, hence limiting the deflation error. For computing eigenvalues and eigenvectors with the block Lanczos method one can find inexact deflation for example in the method presented in [Ruh79].

We will present some methods that implement deflation in Chapter 7 in detail. Besides deflation there are more options to treat (almost) linearly dependent columns in the involved matrices and we will briefly describe three of them.

The first approach is to restart with a reduced number of right-hand sides as soon as losing full rank is observed, e.g. suggested in [Gut06]. The removed systems then have to be treated separately and cannot benefit any more from the block method approach.

The second option to keep block methods from failing is to formulate the algorithms in a robust way by changing how the methods compute needed intermediate matrices. Again, this idea stems from block Lanczos methods. For example, in [GU77] a QR decomposition is used to guarantee linearly independent columns in the used block-vectors. For block Krylov subspace methods, the idea consists of incorporating matrix decompositions in crucial places where the algorithm might rely on inverses of potentially ill-conditioned matrices [Dub01]. Then the inversion is only performed using a well-conditioned factor of the decomposition. We will introduce and discuss methods which implement this approach in Section 6.3.5.

The third strategy consists of substituting the vector that would need to be deflated by a random vector that is orthogonalised against all previous Lanczos vectors [Soo14]. This approach has the advantage of maintaining the block size and keeping the algorithm simple. Moreover, the unrelated random vector might contribute to the solution in a block Krylov subspace method. The downside is that we need to have a set of random vectors right from the start which we have to keep orthogonal to all generated Lanczos vectors so that we can use these as replacement when needed.

6.3.4 Block CG

In Chapter 2 we chose to derive the CG method via the basis generated by the Lanczos process which underlines that it is a Krylov subspace method. There is, however, a different approach to derive the CG method. The starting point of this is the quadratic form

$$f(x) = x^H A x - 2 b^H x \tag{6.19}$$

which takes its minimum at $x^\star$ which is the solution of $Ax = b$ for a hpd matrix A.

One idea to find the minimum of $f(x)$ then is to do line searches and minimise $f(x)$ along one direction at a time. For instance, one can define an iterative scheme

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} r^{(k)}$$

where $r^{(k)}$ is the residual in step k and $\alpha^{(k)}$ is chosen s.t. the error $e^{(k+1)}$ is A-orthogonal to $r^{(k)}$. This minimises the A-norm of the error $e^{(k+1)}$ along the direction $r^{(k)}$. The coefficient $\alpha^{(k)}$ can be explicitly stated as

$$\alpha^{(k)} = \frac{\left\langle r^{(k)}, r^{(k)} \right\rangle}{\left\langle r^{(k)}, A r^{(k)} \right\rangle}.$$

The resulting method goes under the name *steepest descent.*

The CG method improves on this by introducing search directions $p^{(k)}$ that are different from $r^{(k)}$ and along which the error is minimised. These are used to update the iterate via

$$x^{(k+1)} = x^{(k)} + \mu^{(k)} p^{(k)}.$$

Furthermore, the error $e^{(k+1)}$ corresponding to the next iterate not only can be made A-orthogonal to $p^{(k)}$ but also to $p^{(k-1)}$. This is done by keeping the search directions A-orthogonal and compute them as

$$p^{(k+1)} = r^{(k+1)} + \nu^{(k)} p^{(k)}.$$

As stated in equation (2.17), cf. [Gre97, Theorem 2.3.2], this results in the error being minimised over the whole space $e^{(0)} + \operatorname{span}\left\{p^{(0)}, \dots, p^{(k)}\right\}$,

The whole procedure was derived only by enforcing the orthogonality conditions

$$\begin{aligned} (r^{(i)})^H r^{(j)} &= 0, \quad i \neq j, \\ (p^{(i)})^H A p^{(j)} &= 0, \quad i \neq j, \text{ and} \\ (r^{(i)})^H p^{(j)} &= 0, \quad i \neq j. \end{aligned} \tag{6.20}$$

These orthogonality considerations result in the scalars

$$\begin{aligned} \mu^{(k)} &= \frac{\left\langle r^{(k)}, r^{(k)} \right\rangle}{\left\langle p^{(k)}, A p^{(k)} \right\rangle} \quad \text{and} \\ \nu^{(k)} &= \frac{\left\langle r^{(k+1)}, r^{(k+1)} \right\rangle}{\left\langle r^{(k)}, r^{(k)} \right\rangle}, \end{aligned}$$

exactly as in Chapter 2. Finally, we end up with the CG algorithm of Algorithm 2.4. A more detailed description of this derivation approach of the CG method can be found in [Gre97].

The above idea of imposing suitable orthogonality conditions on some search directions can be ported to block methods. The first publication introducing a block CG algorithm was [OLe80]. There, the starting point is a block biconjugate gradient algorithm which is presented first. This is then used to derive a block minimum residual and two block conjugate gradients algorithms—one using a three-term formulation and the other a coupled two-term formulation. We will present the latter one in the following and refer to it as BlockCG. We follow the derivation of BlockCG from [OLe80] which is why we have introduced the alternative derivation of CG above.

For multiple right-hand sides like in block systems of the kind (6.16) we can state a minimisation problem—generalising (6.19)—as minimising

$$F(X) = \operatorname{tr}\left(X^H A X - 2 B^H X\right) \tag{6.21}$$

which is equivalent to minimising $f_j(x_j) = x_j^H A x_j - 2b_j^H x_j$ for $j = 1, \dots, m$. Like before, the $f_j(x)$ take their minimum at the solutions $x_j^\star$ of the systems $Ax_j = b_j$. Thus, the solution $X^\star = [x_1^\star | \dots | x_m^\star]$ of $AX = B$ minimises $F(X)$ in equation (6.21).

We use the iterative scheme

$$X^{(k+1)} = X^{(k)} + P^{(k)} M^{(k)}$$

for updating the block-vector iterate and the block-vector of search directions is updated via

$$P^{(k+1)} = R^{(k+1)} + P^{(k)} N^{(k)}.$$

Now we can impose the block equivalent of the conditions in (6.20), i.e.

$$\begin{aligned} (R^{(i)})^H R^{(j)} &= 0, \quad i \neq j, \\ (P^{(i)})^H A P^{(j)} &= 0, \quad i \neq j, \text{ and} \\ (R^{(i)})^H P^{(j)} &= 0, \quad i \neq j, \end{aligned} \tag{6.22}$$

on the block-vectors. Then we can compute the matrices $M^{(k)}$ and $N^{(k)}$ as

$$\begin{aligned} M^{(k)} &= [(P^{(k)})^H A P^{(k)}]^{-1} (R^{(k)})^H R^{(k)} \quad \text{and} \\ N^{(k)} &= [(R^{(k)})^H R^{(k)}]^{-1} (R^{(k+1)})^H R^{(k+1)}. \end{aligned}$$

With the above, the BlockCG method is a block Krylov subspace method. It creates block iterates $X^{(k)} = [x_1^{(k)} | \dots | x_m^{(k)}]$ with $x_j^{(k)} \in x_j^{(0)} + \mathcal{K}_k^\square(A, R^{(0)})$ and advances from $\mathcal{K}_k^\square(A, R^{(0)})$ to $\mathcal{K}_{k+1}^\square(A, R^{(0)})$ in each step. The iterates $X^{(k)}$ are obtained such that they satisfy the Galerkin condition

$$\begin{gathered} X^{(k)} \in X^{(0)} + \mathcal{K}_k^\square(A, R^{(0)}) \text{ and} \\ \operatorname{colspan}\left\{R^{(k)}\right\} = \operatorname{colspan}\left\{B - AX^{(k)}\right\} \perp \mathcal{K}_k^\square(A, R^{(0)}), \end{gathered}$$

which in the non-block case reduces to the CG iterates. If we denote with $\boldsymbol{V}^{(k)}$ a matrix whose columns form a basis of $\mathcal{K}_k^\square(A, B)$ then the Galerkin condition is equivalent to

$$X^{(k)} = X^{(0)} + \boldsymbol{V}^{(k)} \cdot ((\boldsymbol{V}^{(k)})^H A \boldsymbol{V}^{(k)})^{-1} \cdot (\boldsymbol{V}^{(k)})^H R^{(0)}.$$

In Algorithm 6.7 we present the resulting BlockCG algorithm from [OLe80]. There, it is called B-CG and—additional to our version—features a QR decomposition of the block-vectors $P^{(k)}$ for monitoring linear dependency. However, it is suggested to perform a restart as soon as linearly dependent columns

Algorithm 6.7: BlockCG

Input :	$A \in \mathbb{C}^{n\times n}$, hpd	system matrix
	$B = [b_1 \vert \dots \vert b_m] \in \mathbb{C}^{n\times m}$	right-hand side block-vector
	$X^{(0)} = [x_1^{(0)} \vert \dots \vert x_m^{(0)}] \in \mathbb{C}^{n\times m}$	initial guess block-vector

Output: approximate solution $X^{(k)}$ to $AX = B$

```
1 $R^{(0)} = B - AX^{(0)}$
2 $P^{(0)} = R^{(0)}$
3 for $k = 1, 2, \dots$ until convergence do
4     $Z^{(k-1)} = AP^{(k-1)}$
5     $M^{(k-1)} = [(P^{(k-1)})^H Z^{(k-1)}]^{-1} (R^{(k-1)})^H R^{(k-1)}$
6     $X^{(k)} = X^{(k-1)} + P^{(k-1)} M^{(k-1)}$
7     $R^{(k)} = R^{(k-1)} - Z^{(k-1)} M^{(k-1)}$
8     $N^{(k-1)} = [(R^{(k-1)})^H R^{(k-1)}]^{-1} (R^{(k)})^H R^{(k)}$
9     $P^{(k)} = R^{(k)} + P^{(k-1)} N^{(k-1)}$
```

in $R^{(k)} + P^{(k-1)}N^{(k-1)}$ are detected. No further details on this are given in [OLe80].

The BlockCG algorithm (Algorithm 6.7) will break down due to a singular matrix $(R^{(k-1)})^H R^{(k-1)}$ in line 8 if the block Krylov subspace $\mathcal{K}^{\square}_{k-1}(A, R^{(0)})$ has dimension less than $m(k-1)$. To avoid these breakdowns one can monitor the rank of the matrices $P^{(k)}$ and $R^{(k)}$ as suggested in [OLe80]. If one of these matrices loses full rank, the system belonging to the linearly dependent column has to be removed and the iteration can continue on the remaining systems. In [NY95] it is shown how the block size can be reduced dynamically.

Algorithm 6.7 has to store 4 block-vectors, each having m columns, plus 2 $m \times m$-matrices.

As for the seed and deflation methods for solving systems with multiple right-hand sides we want to compare the computational complexity of BlockCG with CG applied to all right-hand sides separately. We will assume that an operation with an m-column block-vector of length n is as expensive as m vector operations as explained in Section 1.3. In actual computations often times we can benefit from one m-column block-vector operation performing faster than m vector operations, though. This leads us to the computational complexity of Algorithm 6.7 presented in the following result.

Proposition 6.12. *Let k denote the number of iteration steps needed to solve the block system* (6.16) *with the BlockCG algorithm. The total number of operations in Algorithm 6.7 is*

$$o_{\text{blockcg}} = km(c_A^{\square}(m) + 5m). \qquad (6.23)$$

Proof. For Algorithm 6.7, we count

- one matrix-block-vector product,
- three block-vector updates of $X^{(k)}, R^{(k)}$ and $P^{(k)}$ and
- one computation of $(P^{(k-1)})^H Z^{(k-1)}$ and $(R^{(k)})^H R^{(k)}$—the latter one can be computed once and then reused in two more places.

We end up with (6.23) by adding everything. □

Note that we assumed $m \ll n$ in Proposition 6.12 and thus neglect the $\mathcal{O}(m^3)$ operations—the two inversions of $m \times m$-matrices and the products of $m \times m$-matrices.

Now, we can compare BlockCG for solving (6.16) to solving all m systems in (6.1) with CG.

Proposition 6.13. *We assume that solving* (6.1) *with the CG algorithm takes an average of* $\tilde{k}$ *steps. The number of vector operations in BlockCG is less than the number of vector operations in CG, hinting at BlockCG being able to outperform CG for solving all systems in* (5.1)*, if the inequality*

$$\frac{c_A + 5}{c_A^{\square}(m) + 5m} \geq \frac{k}{\tilde{k}} \tag{6.24}$$

holds.

Proof. With the above assumption and using the definition of $o_{\text{cg-mrhs}}$ in (6.4) we get

$$\begin{aligned} & o_{\text{cg-mrhs}} \geq o_{\text{blockcg}} \\ \Leftrightarrow \quad & (c_A + 5)m\tilde{k} \geq km(c_A^{\square}(m) + 5m) \\ \Leftrightarrow \quad & (c_A + 5)\tilde{k} \geq k(c_A^{\square}(m) + 5m) \\ \Leftrightarrow \quad & \tilde{k}\frac{c_A + 5}{c_A^{\square}(m) + 5m} \geq k. \end{aligned}$$

□

This suggests that BlockCG should perform at most $\frac{c_A+5}{c_A^{\square}(m)+5m}$ times the number of steps that CG does in order to be faster. If, for example, BlockCG manages to reduce the number of iteration steps to $\frac{k}{\tilde{k}} = \frac{1}{2}$ and $c_A = c_A^{\square}(m)$ then $c_A \geq 5(m-2)$ needs to hold for BlockCG to perform less vector operations than CG. In reality, it is more complicated, since BlockCG gets more costly when counting the $\mathcal{O}(m^3)$ operations.

The inequality (6.24) backs two observations often made with block methods. First, as the number of right-hand sides is increased, the number k of steps of

BlockCG must drop accordingly for it to be faster than CG. Second, the first observation is alleviated by a high cost c_A of the matrix-vector product and even more so if $c_A^{\square}(m)$ is significantly smaller than c_A. This means, that if the fraction in the left-hand side of (6.24) is dominated by a costly matrix-vector product with the matrix A then BlockCG can lead to savings in computational time even for a small reduction in the number of iteration steps.

In [OLe80, Section 4] the convergence of the BlockCG algorithm was analysed. We state the result which is similar to that of Theorem 2.10 whilst omitting the quite long and technical proof.

Theorem 6.14. [OLe80, Theorem 5] *Let $e_j^{(k)}$ be the error of system j at step k of the BlockCG algorithm applied to the Hermitian positive definite linear block system $AX = B$ with $X, B \in \mathbb{C}^{n\times m}$. Let $U^H AU = \Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$ with $U^H U = I_n$ and $0 < \lambda_1 \leq \dots \leq \lambda_n$ and let $\kappa = \lambda_n/\lambda_m$. Then the bound*

$$\left\| e_j^{(k)} \right\|_A \leq c \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k$$

holds. The constant c depends on A, the initial right-hand sides—i.e. on the initial error as in Theorem 2.10—and on m but not on k.

Proof. For the proof and the definition of the constant c see [OLe80, Theorem 5]. □

Theorem 6.14 shows that the convergence rate of BlockCG depends on the initial guess $X^{(0)}$ and more importantly on the eigenvalue distribution of A. If the $n - m + 1$ largest eigenvalues are clustered then κ might be significantly smaller than λ_n/λ_1. This can even contribute to alleviate the penalty from more right-hand sides in (6.24).

We finish our discussion of BlockCG with noting that breakdowns of the BlockCG method in case of indefinite A have been investigated in [Bro97]. In the following we give a couple of remarks on related methods.

A deflated version of BlockCG can be attributed to [NY95; NY03]. The *VBPCG (variable block preconditioned conjugate gradient) algorithm* presented there can dynamically reduce the block size of the involved matrices without restarting the whole process. The method, however, is only capable of solving a single right-hand side system $Ax = b$ whilst accelerating the solution of this system with the block method idea. This is done in the VBPCG method by starting with a block system $A[x|\dots] = [b|\dots]$ where only the solution of the first system is of interest. The last $m-1$ columns in $B = [b|\dots]$, i.e. all but the first column, are chosen randomly, s.t. B has full rank. When computations in parallel are used this might result in solving $Ax = b$ faster than with CG. But, in [NY95; NY03] there is no explanation given how the deflated systems could be solved and thus whether VBPCG can be used to solve $AX = B$.

In [CW97]—besides the Single Seed Method we presented in Section 6.1—a hybrid of BlockCG and the Single Seed Method is presented. It is called the Block Seed Method and the results there suggest that it can perform better than the Single Seed Method if the right-hand sides are not closely related.

The BlockCG method can be used to accelerate row projection methods like the Cimmino method. Instead of applying the Cimmino approach to single rows of the matrix A, block slices are used which require solving systems with multiple right-hand sides. The resulting block Cimmino method is examined in, e.g. [Lew95; Rui92; Zen13].

6.3.5 Retooled Block CG

In [Dub01] another approach for curing the (nearly) rank deficiency problems discussed in Section 6.3.3 whilst explicitly avoiding deflation in BlockCG is presented. The key idea, common to all methods in [Dub01], is to use factorisations, e.g. QR decompositions, of the search directions block-vector $P^{(k)}$ and/or the residual block-vector $R^{(k)}$. These are applied to enforce full rank of the matrices that have to be inverted in the BlockCG algorithm thereby successfully disguising the potential rank deficiency from the algorithm. The resulting retooled methods have the advantage that there is no need for book keeping of those vectors that became linearly dependent or to work on block-vectors whose number of columns gets reduced during the iteration. Hence, the algorithms in [Dub01] are much simpler and easy to implement than most other methods. On the downside one always invests the equivalent of m matrix-vector multiplications with A per step, even when this would not be necessary due to rank deficiency. Therefore, the desirable side effect of reducing the number of systems, which implies a reduction of the amount of work, that deflation brings with it, cannot be exploited.

In the following we describe two retooled BlockCG variants. We chose to present these two since each of them tries to extend one particular property of the single right-hand side CG algorithm to the block-vectors of the BlockCG algorithm. In [Dub01] more variants are derived and compared to BlockCG.

BlockCG with factorisation of the search direction block-vector

The first algorithm from [Dub01] we want to present is based on an alternative formulation of the BlockCG algorithm. We call this formulation BCGA and we describe the changes that have to be applied to BlockCG leading to BCGA. This version differs from BlockCG in computing the matrices $M^{(k-1)}$ and $N^{(k-1)}$ in an alternate, albeit mathematically equivalent, way. The matrix $M^{(k-1)}$ can be computed differently as

$$\begin{aligned} M^{(k-1)} &= [(P^{(k-1)})^H A P^{(k-1)}]^{-1} (R^{(k-1)})^H R^{(k-1)} \\ &= [(P^{(k-1)})^H A P^{(k-1)}]^{-1} (P^{(k-1)})^H R^{(k-1)}. \end{aligned}$$

This holds, since $P^{(k-1)} = R^{(k-1)} + P^{(k-2)} N^{(k-2)}$ and $(P^{(k-2)})^H R^{(k-1)} = 0$. Assuming $M^{(k-1)}$ to be non-singular, the computation of $N^{(k-1)}$ can be rewritten to

$$\begin{aligned} N^{(k-1)} &= [(R^{(k-1)})^H R^{(k-1)}]^{-1} (R^{(k)})^H R^{(k)} \\ &= [(R^{(k)} + AP^{(k-1)} M^{(k-1)})^H (P^{(k-1)} - P^{(k-2)} N^{(k-2)})]^{-1} \\ &\quad \cdot (R^{(k-1)} - AP^{(k-1)} M^{(k-1)})^H R^{(k)} \\ &= -[(M^{(k-1)})^H (P^{(k-1)})^H A P^{(k-1)}]^{-1} (M^{(k-1)})^H (P^{(k-1)})^H A R^{(k)} \\ &= -[(P^{(k-1)})^H A P^{(k-1)}]^{-1} (P^{(k-1)})^H A R^{(k)} \end{aligned}$$

where we utilised the orthogonality conditions (6.22). This results in the alternate BlockCG version that is displayed in Algorithm 6.8.

Algorithm 6.8: BCGA

```
Input  :  $A \in \mathbb{C}^{n \times n}$, hpd                                   system matrix
          $B = [b_1 | \dots | b_m] \in \mathbb{C}^{n \times m}$                  right-hand side block-vector
          $X^{(0)} = [x_1^{(0)} | \dots | x_m^{(0)}] \in \mathbb{C}^{n \times m}$   initial guess block-vector

Output:   approximate solution $X^{(k)}$ to $AX = B$

1 $R^{(0)} = B - AX^{(0)}$
2 $P^{(0)} = R^{(0)}$
3 for $k = 1, 2, \dots$ until convergence do
4     $Z^{(k-1)} = AP^{(k-1)}$
5     $M^{(k-1)} = [(P^{(k-1)})^H Z^{(k-1)}]^{-1} (P^{(k-1)})^H R^{(k-1)}$
6     $X^{(k)} = X^{(k-1)} + P^{(k-1)} M^{(k-1)}$
7     $R^{(k)} = R^{(k-1)} - Z^{(k-1)} M^{(k-1)}$
8     $N^{(k-1)} = -[(P^{(k-1)})^H Z^{(k-1)}]^{-1} (Z^{(k-1)})^H R^{(k)}$
9     $P^{(k)} = R^{(k)} + P^{(k-1)} N^{(k-1)}$
```

The BCGA algorithm is endowed with a particular property that BlockCG does not have and which we want to exploit. This property is that every factorisation of the block-vector of search directions

$$F^{(k)} C^{(k)} = P^{(k)}$$

with non-singular $C^{(k)} \in \mathbb{C}^{m\times m}$ results in an algorithm in which $C^{(k)}$ does not appear explicitly. In numerical computations, however, it is unlikely that $C^{(k)}$ is exactly singular.

Applying the above factorisation to the according lines from 4 to 9 of the BCGA algorithm yields the following new computations. With $\tilde{M}^{(k-1)}$ we denote the matrix $M^{(k-1)}$ of BCGA which we transform to

$$\begin{aligned}
\tilde{M}^{(k-1)} &= [(P^{(k-1)})^H A P^{(k-1)}]^{-1}(P^{(k-1)})^H R^{(k-1)} \\
&= [(C^{(k-1)})^H (F^{(k-1)})^H A F^{(k-1)} C^{(k-1)}]^{-1} (F^{(k-1)} C^{(k-1)})^H R^{(k-1)} \\
&= (C^{(k-1)})^{-1} [(F^{(k-1)})^H A F^{(k-1)}]^{-1} \\
&\quad \cdot \underbrace{(C^{(k-1)})^{-H} (C^{(k-1)})^H}_{=I} (F^{(k-1)})^H R^{(k-1)} \\
&= (C^{(k-1)})^{-1} [(F^{(k-1)})^H A F^{(k-1)}]^{-1} (F^{(k-1)})^H R^{(k-1)}.
\end{aligned}$$

Now, we define

$$M^{(k-1)} = [(F^{(k-1)})^H A F^{(k-1)}]^{-1} (F^{(k-1)})^H R^{(k-1)}.$$

The updates for the iterate and residual block-vectors become

$$\begin{aligned}
X^{(k)} &= X^{(k-1)} + F^{(k-1)} C^{(k-1)} \tilde{M}^{(k-1)} \\
&= X^{(k-1)} + F^{(k-1)} M^{(k-1)}
\end{aligned}$$

and

$$R^{(k)} = R^{(k-1)} - A F^{(k-1)} M^{(k-1)}.$$

Again, we use $\tilde{N}^{(k-1)}$ to denote the matrix $N^{(k-1)}$ from BCGA and see that

$$\begin{aligned}
\tilde{N}^{(k-1)} &= -[(P^{(k-1)})^H A P^{(k-1)}]^{-1} (A P^{(k-1)})^H R^{(k)} \\
&= -[(F^{(k-1)} C^{(k-1)})^H A F^{(k-1)} C^{(k-1)}]^{-1} (A F^{(k-1)} C^{(k-1)})^H R^{(k)} \\
&= -(C^{(k-1)})^{-1} [(F^{(k-1)})^H A F^{(k-1)}]^{-1} \\
&\quad \cdot \underbrace{(C^{(k-1)})^{-H} (C^{(k-1)})^H}_{=I} (A F^{(k-1)})^H R^{(k)} \\
&= -(C^{(k-1)})^{-1} [(F^{(k-1)})^H A F^{(k-1)}]^{-1} (A F^{(k-1)})^H R^{(k)}
\end{aligned}$$

holds. Then, we define

$$N^{(k-1)} = -[(F^{(k-1)})^H A F^{(k-1)}]^{-1} (A F^{(k-1)})^H R^{(k)}.$$

Algorithm 6.9: BCGAdF

Input :	$A \in \mathbb{C}^{n\times n}$, hpd	system matrix
	$B = [b_1 \mid \dots \mid b_m] \in \mathbb{C}^{n\times m}$	right-hand side block-vector
	$X^{(0)} = [x_1^{(0)} \mid \dots \mid x_m^{(0)}] \in \mathbb{C}^{n\times m}$	initial guess block-vector
Output:	approximate solution $X^{(k)}$ to $AX = B$	

```
1 $R^{(0)} = B - AX^{(0)}$
2 $F^{(0)}C^{(0)} = P^{(0)}$                                         // factorisation of $P^{(0)}$
3 for $k = 1, 2, \dots$ until convergence do
4     $Z^{(k-1)} = AF^{(k-1)}$
5     $M^{(k-1)} = [(F^{(k-1)})^H Z^{(k-1)}]^{-1} (F^{(k-1)})^H R^{(k-1)}$
6     $X^{(k)} = X^{(k-1)} + F^{(k-1)} M^{(k-1)}$
7     $R^{(k)} = R^{(k-1)} - Z^{(k-1)} M^{(k-1)}$
8     $N^{(k-1)} = -[(F^{(k-1)})^H Z^{(k-1)}]^{-1} (Z^{(k-1)})^H R^{(k)}$
9     $F^{(k)}C^{(k)} = R^{(k)} + F^{(k-1)} N^{(k-1)}$                  // factorisation of $P^{(k)}$
```

Finally, the factorisation of the search direction block-vector becomes

$$\begin{aligned} F^{(k)}C^{(k)} &= R^{(k)} + F^{(k-1)}C^{(k-1)}\tilde{N}^{(k-1)} \\ &= R^{(k)} + F^{(k-1)}N^{(k-1)}. \end{aligned}$$

The resulting algorithm BCGAdF is displayed in Algorithm 6.9.

Algorithm 6.9 can be implemented, e.g. with a QR decomposition as factorisation in the lines 2 and 9 and is then referred to as BCGAdQ. In [Dub01] another factorisation is proposed. If the search direction block-vector is factorised to yield A-orthonormal columns of $F^{(k)}$ then the computation of $M^{(k-1)}$ and $N^{(k-1)}$ simplifies, since the inversions vanish. Moreover, the property of A-orthogonal search directions is carried from CG to this version of BlockCG. For computing such a factorisation, a modified Gram-Schmidt method which computes the factor $F^{(k)}$ of $F^{(k)}C^{(k)} = P^{(k)}$ with $(F^{(k)})^H AF^{(k)} = I$ is presented in [Dub01]. Furthermore, the computation of $F^{(k)}$ can be done in-situ and requires no additional multiplications with A. This version of BCGAdF using the A-orthonormalisation is called BCGAdA. However, in [Dub01] erratic behaviour of BCGAdF with the A-orthogonalisation was observed. The proposed solution involves computing a QR decomposition of the search direction block-vector before applying the A-orthogonalisation. In [Dub01] BCGAdA with this additinal decomposition was slower than all other variants which is why we use the BCGAdQ variant from now on.

The computational complexity of Algorithm 6.9 in its BCGAdQ variant is a bit higher than the computational complexity of BlockCG from Proposition 6.12 and we state it in the proposition below.

Proposition 6.15. *Let k denote the number of iteration steps needed to solve the block system* (6.16) *with the BCGAdQ algorithm. The number of operations in Algorithm 6.9 including a QR decomposition of the search direction block-vectors is*

$$o_{\text{bcgadq}} = km(c_A^{\square}(m) + 8m + 4). \tag{6.25}$$

Proof. There are two differences to the BlockCG case which we have to describe. First, the QR decomposition adds an equivalent of $2m^2 + 4m$ vector operations per iteration step if we use the Householder QR decomposition from Proposition 1.5. Second, since BCGAdQ is based on BCGA it uses a different computation of the matrices $M^{(k-1)}$ and $N^{(k-1)}$. In BCGAdQ we can reuse the matrix $[(F^{(k-1)})^H Z^{(k-1)}]^{-1}$ only once whereas we could reuse $(R^{(k)})^H R^{(k)}$ twice in BlockCG. Therefore, we have $3m^2 + 4m$ additional operations as compared to BlockCG leading to o_{bcgadq} in (6.25). □

We see that o_{bcgadq} adds little work as compared to o_{blockcg} only if the cost $c_A^{\square}(m)$ for multiplying with the matrix A in terms of vector operations is large in comparison to m. If, however, $c_A^{\square}(m) \approx m$ then the BCGAdQ algorithm needs significantly more operations than BlockCG. But, one should keep in mind that encountering (nearly) linear dependencies can result in BlockCG taking much more iteration steps or it might not converge at all. This brings us to the comparison of BCGAdQ for solving (6.16) to solving all m systems in (6.1) with CG.

Proposition 6.16. *We assume that the systems in* (6.1) *take an average of $\tilde{k}$ steps when solved with the CG algorithm. The number of vector operations in BCGAdQ is less than the number of vector operations in CG, hinting at BCGAdQ being faster than CG for solving all systems in* (5.1)*, if the inequality*

$$\frac{c_A + 5}{c_A^{\square}(m) + 8m + 4} \geq \frac{k}{\tilde{k}}$$

holds.

Proof. See Proposition 6.13. □

BlockCG with factorisation of the residual block-vector

The second algorithm from [Dub01] that we want to describe in the remainder of this section is called BCGrQ. It is a retooled version of BlockCG and shows the best performance in flop count in the numerical experiments performed in [Dub01]. It carries the property of orthogonal residual vectors from non-block

CG to the block case by applying a QR decomposition to the block-vector of residuals, hence the “rQ” suffix. In contrast to BCGAdF it is not based on the alternate formulation of BlockCG. The potential ill-conditioned or rank deficient matrices do not vanish completely in BCGrQ, but they only occur in non-critical places.

To derive BCGrQ we can start from the BlockCG algorithm (Algorithm 6.7). Therein, we apply a QR decomposition

$$Q^{(k)}C^{(k)} = R^{(k)} \quad \text{with} \quad (Q^{(k)})^H Q^{(k)} = I_m$$

to the block-vector of residuals and we use a transformation

$$D^{(k)}C^{(k)} = P^{(k)}$$

of the search direction block-vector. For the derivation we will assume that $C^{(k)}$ can be close to singular but maintains full rank in exact arithmetic. Then, we define the auxiliary matrix

$$S^{(k)} = C^{(k)}(C^{(k-1)})^{-1}.$$

Now, we can start to substitute in the according lines from 4 to 9 of the BlockCG algorithm. First, we have

$$\begin{aligned}
\tilde{M}^{(k-1)} &= [(P^{(k-1)})^H A P^{(k-1)}]^{-1}(R^{(k-1)})^H R^{(k-1)} \\
&= [(D^{(k-1)}C^{(k-1)})^H A D^{(k-1)}C^{(k-1)}]^{-1}(C^{(k-1)})^H \underbrace{(Q^{(k-1)})^H Q^{(k-1)}}_{=I} C^{(k-1)} \\
&= (C^{(k-1)})^{-1}[(D^{(k-1)})^H A D^{(k-1)}]^{-1} \underbrace{(C^{(k-1)})^{-H}(C^{(k-1)})^H}_{=I} C^{(k-1)} \\
&= (C^{(k-1)})^{-1}[(D^{(k-1)})^H A D^{(k-1)}]^{-1} C^{(k-1)},
\end{aligned}$$

where $\tilde{M}^{(k-1)}$ stems from BlockCG and for BCGrQ we define

$$M^{(k-1)} = ((D^{(k-1)})^H A D^{(k-1)})^{-1}.$$

Now the iterate can be updated via

$$\begin{aligned}
X^{(k)} &= X^{(k-1)} + D^{(k-1)}C^{(k-1)}\tilde{M}^{(k-1)} \\
&= X^{(k-1)} + D^{(k-1)}M^{(k-1)}C^{(k-1)}.
\end{aligned}$$

The residual block-vector can be computed as a QR decomposition as in

$$\begin{aligned}
Q^{(k)}C^{(k)} &= Q^{(k-1)}C^{(k-1)} + AD^{(k-1)}C^{(k-1)}\tilde{M}^{(k-1)} \\
&= Q^{(k-1)}C^{(k-1)} + AD^{(k-1)}M^{(k-1)}C^{(k-1)} \\
\Leftrightarrow \qquad Q^{(k)} \underbrace{C^{(k)}(C^{(k-1)})^{-1}}_{=S^k} &= Q^{(k-1)} + AD^{(k-1)}M^{(k-1)}.
\end{aligned}$$

The search direction block-vector is updated via

$$\begin{aligned} D^{(k)}C^{(k)} &= Q^{(k)}C^{(k)} + P^{(k-1)}[(P^{(k-1)})^H P^{(k-1)}]^{-1}(P^{(k)})^H P^{(k)} \\ &= Q^{(k)}C^{(k)} + D^{(k-1)}(C^{(k-1)})^{-H}(C^{(k)})^H C^{(k)} \\ \Leftrightarrow \qquad D^{(k)} &= Q^{(k)} + D^{(k-1)}(S^{(k)})^H. \end{aligned}$$

And, finally, the matrix $C^{(k)}$, which contains the information on the magnitude of the residuals, can be updated stably as

$$C^{(k)} = S^{(k)}C^{(k-1)}.$$

All this results in the BCGrQ algorithm from [Dub01], which we state in our notation as Algorithm 6.10. Note that the only matrix we need to invert is $(D^{(k)})^H Z^{(k)}$ which has full rank. This follows from $Q^{(k)}$ having full rank stemming from a QR decomposition and the imposed orthogonality conditions (6.22) which together yield

$$(Q^{(k)})^H D^{(k)} = I_m.$$

All in all, this eliminates the need for deflation, because even if some of the columns of the residual block-vector become linearly dependent then the rank deficiency appears in $C^{(k)}$ which is only used to scale the contribution of the search direction block-vector to the solution.

Algorithm 6.10: BCGrQ

Input :	$A \in \mathbb{C}^{n\times n}$, hpd	system matrix
	$B = [b_1 \mid \dots \mid b_m] \in \mathbb{C}^{n\times m}$	right-hand side block-vector
	$X^{(0)} = [x_1^{(0)} \mid \dots \mid x_m^{(0)}] \in \mathbb{C}^{n\times m}$	initial guess block-vector
Output:	approximate solution $X^{(k)}$ to $AX = B$	

```
1 $Q^{(0)}C^{(0)} = R^{(0)} = B - AX^{(0)}$, $D^{(0)} = Q^{(0)}$        // QR decomposition of $R^{(0)}$
2 for $k = 1, 2, \dots$ until convergence do
3     $Z^{(k-1)} = AD^{(k-1)}$
4     $M^{(k-1)} = [(D^{(k-1)})^H Z^{(k-1)}]^{-1}$
5     $X^{(k)} = X^{(k-1)} + D^{(k-1)}M^{(k-1)}C^{(k-1)}$
6     $Q^{(k)}S^{(k)} = Q^{(k-1)} - Z^{(k-1)}M^{(k-1)}$        // QR decomposition of $R^{(k)}$
7     $D^{(k)} = Q^{(k)} + D^{(k-1)}(S^{(k)})^H$
8     $C^{(k)} = S^{(k)}C^{(k-1)}$
```

As for BCGAdQ, the computational complexity of BCGrQ in Algorithm 6.10 is almost similar to the computational complexity of BlockCG.

Proposition 6.17. *Let k denote the number of iteration steps needed to solve the block system* (6.16) *with the BCGrQ algorithm. The number of operations in Algorithm 6.10 is*

$$o_{\mathrm{bcgrq}} = km(c_A^{\square}(m) + 6m + 4). \tag{6.26}$$

Proof. There are two noticeable changes in the number of operations as compared to BlockCG. First, we have m^2 less operations than in BlockCG for only building $M^{(k-1)}$ and not also $N^{(k-1)}$. Second, the QR decomposition in line 6 requires $2m^2 + 4m$ additional vector operations as stated in Proposition 1.5 when Algorithm 1.1 is applied. All in all, this results in (6.26). $\square$

This results in only slightly more work as compared to the BlockCG algorithm and less work than BCGAdQ. As for the previous block methods the following proposition states a comparison of BCGrQ with CG.

Proposition 6.18. *We assume that the systems in* (6.1) *take an average of $\tilde{k}$ steps when solved with the CG algorithm. The number of vector operations in BCGrQ is less than the number of vector operations in CG, hinting at BCGrQ being faster than CG for solving all systems in* (5.1)*, if the inequality*

$$\frac{c_A + 5}{c_A^{\square}(m) + 6m + 4} \geq \frac{k}{\tilde{k}}$$

holds.

Proof. See Proposition 6.13. $\square$

7 Krylov Subspace Methods for Shifted Block Systems

The previous two chapters introduced methods that could either solve shifted systems (5.1) or systems with multiple right-hand sides (6.1). In Chapter 4, however, we presented applications which require the solution of a combination of both. In fact, for these applications we need to solve *families of shifted linear systems with multiple right-hand sides* (1.1) that we repeat here for convenience

$$(A + \sigma_i I)x_{i,j} = b_j, \quad i = 1, \dots, s, \quad j = 1, \dots, m. \tag{1.1}$$

In this chapter we will present methods that are capable of solving (1.1) efficiently. We will focus on the case where $(A + \sigma_i I) \in \mathbb{C}^{n \times n}$ is Hermitian positive definite for every shift $\sigma_i \in \mathbb{C}$ and $x_{i,j}, b_j \in \mathbb{C}^n$.

As we did for unshifted systems with multiple right-hand sides in Chapter 6 we can reformulate equation (1.1). By arranging the m right-hand sides and the corresponding ms solution vectors in the matrices

$$\begin{aligned} B &= [b_1| \cdots |b_m] \quad \text{and} \\ X_j &= [x_{1,j}| \cdots |x_{m,j}] \end{aligned}$$

we can rewrite (1.1) as a *shifted block system*

$$(A + \sigma_j I)X_j = B. \tag{7.1}$$

One could apply either a shifted method from Chapter 5 to every shifted system (5.1) belonging to right-hand side b_j or one could treat every system belonging to a single shift σ_i as a system (6.1) or a block system (6.16) and apply a method from Chapter 6. Both choices are more or less practicable and we will explore them numerically in Chapter 8. Nevertheless, a third alternative is to apply methods that are specifically developed with the family of systems (1.1) in mind.

Methods targeted at solving the systems (1.1) and (7.1) efficiently are relatively rare. Until recently there were, to our knowledge, two methods especially meant for solving these systems. Both can solve (1.1) and (7.1), respectively, for general (not necessarily Hermitian) matrices $(A + \sigma_i I) \in \mathbb{C}^{n \times n}$.

The first method is an extension of GMRES-DR called GMRES-Proj-Sh from [DMW08]. It has to perform restarts in order to limit its memory footprint and computational cost as it is based on GMRES—a method using long recurrences. These restarts can deteriorate the convergence of the method. This is tried to be alleviated by performing *deflated restarts*, which means that eigenspace information is carried from every restart cycle to the next. The eigenvalue computations involved in this process can make the restarts quite costly. Hence, one has to find a tradeoff between the convergence speed and the costs for restarting. Nonetheless, especially for shifted systems with related right-hand sides GMRES-Proj-Sh shows superior convergence behaviour in [DMW08]. GMRES-Proj-Sh is a deflation method as those presented in Section 6.2 extended for multiple shifts. It solves the shifted systems (1.1) for every right-hand side one after the other. Thus, information from solving the systems belonging to the right-hand side b_j can only be shared to the systems belonging to upcoming right-hand sides. A block method approach might be able to improve on that. On the plus side, GMRES-Proj-Sh does not need to check for linearly dependent vectors as a block method would.

The second method is a block-QMR method for multiple shifts from [FFF97] which extends the unshifted block-QMR method from [FM97]. It is based on a block Lanczos-type process that extends the two-sided Lanczos process to multiple starting vectors and includes deflation in the block method sense. We will describe this block Lanczos-type process in detail in Section 7.1.2. Being a QMR method, though, the block-QMR algorithm not only inherits short recurrences but also the need for a multiplication with A and with A^H per step of the iteration. The multiplication with A^H does not contribute to the solution. It is only needed to span an additional space for bi-orthogonalisation. Thus, methods tailored for a Hermitian positive definite matrix A that only need one multiplication with A in every iteration step can be expected to perform better.

Another method for solving (1.1) for Hermitian positive definite matrices appeared recently in [Fut$^+$13]. It is an extension of BCGrQ that we have presented in Section 6.3 to the multiple shifts case. Due to the recent publication of the article in the proceedings of the High Performance Computing for Computational Science conference 2012 we only present the algorithm briefly and apply our cost model in order to compare it to the other methods.

This chapter is organised as follows. In Section 7.1 we first introduce deflated block Krylov subspaces. Afterwards, we present three methods for computing bases for these. In Section 7.2 we develop a CG method called DSBlockCG for multiple right-hand sides and multiple shifts for solving (1.1) that advances the deflated block Krylov subspaces vector-wise. The method developed in Section 7.3, BFDSCG, is a deflated shifted block Krylov subspace method that advances the deflated block Krylov subspace block-wise. Both methods are based on shifted CG from Section 5.1 and the Lanczos-type processes from Section 7.1. Finally, we present the SBCGrQ algorithm from [Fut+13] in Section 7.4.

7.1 Deflated Block Lanczos Processes

This section is dedicated to introduce the block Lanczos-type process and variants thereof that most methods presented in this chapter are based upon. It was presented by Boley [ABH94] and Freund [FF95; FF94] at the same Oberwolfach workshop after being developed independently. The version of the block Lanczos-type process that we present first and want to base deflated block Krylov subspace methods on is explained in detail in [Ali+00]. It has been used, for instance, in the Block-QMR method from [FM97].

7.1.1 Deflated Block Krylov Subspaces

Before we can describe the block Lanczos-type process we need to enhance our Definition 6.7 of block Krylov subspaces. The block Lanczos-type process and the methods based upon it advance the block Krylov subspace not in blocks of m vectors at every step. Instead, it constructs *intermediate subspaces* that result from extending the block Krylov subspaces by a dimension of 1 using one vector a time. This is sometimes called *Ruhe's approach*, since this single vector approach, that we described briefly in Section 6.3.2, was first introduced in [Ruh79]. The following definition extends Definition 6.7 and includes the mentioned intermediate subspaces.

Definition 7.1. *Let $A \in \mathbb{C}^{n\times n}$ and $B \in \mathbb{C}^{n\times m}$. We define by*

$$\begin{aligned}\mathcal{K}^{\square}_{k,j}(A,B) &= \mathcal{K}^{\square}_{k}(A,B) + \operatorname{span}\left\{A^k b_1, \dots, A^k b_j\right\}\\ &= \operatorname{span}\left\{b_1, \dots, b_m, Ab_1, \dots, Ab_m, A^2 b_1, \dots, A^{k-1} b_m, A^k b_1, \dots, A^k b_j\right\}\\ &= \operatorname{colspan}\left\{\boldsymbol{K}_{k,j}(A,B)\right\} \qquad (7.2)\end{aligned}$$

the (k, j)-th block Krylov subspace *generated by the matrix A and block-vector B. Here, we used*

$$\boldsymbol{K}_{k,j}(A,B) = [B|AB|A^2B|\cdots|A^{k-1}B|A^k b_1|\cdots|A^k b_j]$$

what we refer to as the block Krylov matrix.

Note that $\mathscr{K}^{\square}_{k,0}(A,B) = \mathscr{K}^{\square}_{k}(A,B)$ and $\mathscr{K}^{\square}_{k,m}(A,B) = \mathscr{K}^{\square}_{k+1}(A,B)$. Moreover, for $0 < j < m$ the block Krylov subspaces $\mathscr{K}^{\square}_{k,j}(A,B)$ “lie between” $\mathscr{K}^{\square}_{k}(A,B)$ and $\mathscr{K}^{\square}_{k+1}(A,B)$, i.e.

$$\mathscr{K}^{\square}_{k}(A,B) \subseteq \mathscr{K}^{\square}_{k,1}(A,B) \subseteq \cdots \subseteq \mathscr{K}^{\square}_{k,m-1}(A,B) \subseteq \mathscr{K}^{\square}_{k+1}(A,B).$$

When it comes to computing bases for block Krylov subspaces, Definition 7.1 is useful as long as the block Krylov matrix from equation (7.2), whose columns span the block Krylov subspace, does not contain *nearly linearly dependent* vectors. By nearly linearly dependent vectors we refer to vectors that would yield a vector of small norm after being orthogonalised against previous vectors. If it contains exactly linearly dependent vectors then we could use Definition 7.1 for defining deflated block Krylov subspace methods since we can simply span the space $\mathscr{K}^{\square}_{k,j}(A,B)$ with less than $mk + j$ vectors. If, however, the block Krylov matrix $\boldsymbol{K}_{k,j}(A,B)$ contains nearly linearly dependent vectors then we want to be able to remove them for numerical reasons as described in Section 6.3.3. This implies that we are not spanning $\mathscr{K}^{\square}_{k,j}(A,B)$ any more.

So, to arrive at a final definition for deflated block Krylov subspaces and the resulting deflated block Krylov subspace methods we first have to be able to express the matrix whose columns are a basis of the corresponding deflated block Krylov subspace. We define it to be a submatrix of $\boldsymbol{K}_{k,j}(A,B)$ where some of the columns are removed. Clearly, if we have a set of linearly dependent vectors there are multiple choices for removing vectors to end up with a linearly independent set of vectors. The same is true for nearly linearly dependent vectors. But, in view of an implementation we scan the matrix $\boldsymbol{K}_{k,j}(A,B)$ from left to right. Every time the current vector turns out to be nearly linearly dependent on the previous vectors, we remove it. One important observation for this procedure is that if $A^p x$ is linearly dependent on previous vectors then so is $A^q x$ for $q > p$. For nearly linearly dependent $A^p x$ it is not that straightforward. We still remove them but have to discuss the implications later in the section. Hence, if we assume that amongst the starting vectors in $B^{(0)} = B$ there are no (nearly) linearly dependent ones then we can define the deflated block Krylov matrix as

$$\boldsymbol{K}^{\text{defl}}_{k,j}(A,B) := [B^{(0)}|AB^{(1)}|A^2B^{(2)}|\cdots|A^{k-1}B^{(k-1)}|A^kB^{(k)}_{:,1}|\cdots|A^kB^{(k)}_{:,j}], \quad (7.3)$$

where the matrices $B^{(c)}$ are submatrices of $B^{(c-1)}$. These submatrices consist of the columns of the previous matrices except for those columns of $A^cB^{(c-1)}$ that are identified as linearly dependent or nearly linearly dependent. In the following we will refer to this process of removing vectors as exact or inexact deflation and we will simply use *deflation* if we do not need to distinguish between the two. If $A^c B^{(c-1)}_{:,j}$ is (nearly) linearly dependent then $B^{(c-1)}_{:,j}$ does not appear in $B^{(c)}$ any more and we refer to the vector $A^c B^{(c-1)}_{:,j}$ as a deflated vector. Note that if deflation occurred d times until step $(k, 0)$ then the matrix $B^{(k-1)}$ contains $m - d$ columns.

All in all, this finally allows us to define deflated block Krylov subspaces and deflated block Krylov subspace methods. We stress that in some cases it is more convenient to define deflated block Krylov subspaces via their dimension and in other cases via their relation to the matrix $\boldsymbol{K}_{k,j}(A, B)$. Thus, the following definitions contain two flavours of definitions for deflated block Krylov subspaces and deflated block Krylov subspace methods. For counting the dimension we introduce κ as indexing variable.

Definition 7.2. *Let $A \in \mathbb{C}^{n\times n}$, $B \in \mathbb{C}^{n\times m}$ and $\boldsymbol{K}^{\text{defl}}_{k,j}(A, B)$ as in* (7.3)*. We define the (k, j)-th* deflated block Krylov subspace *generated by the matrix A and block-vector B as*

$$\mathcal{K}^{\text{defl}}_{k,j}(A, B) := \operatorname{colspan}\left\{\boldsymbol{K}^{\text{defl}}_{k,j}(A, B)\right\}$$

and the deflated block Krylov subspace *of dimension κ as*

$$\mathcal{K}^{\text{defl}}_{\kappa}(A, B) := \operatorname{colspan}\left\{\boldsymbol{K}^{\text{defl}}_{\kappa}(A, B)\right\}.$$

The matrix $\boldsymbol{K}^{\text{defl}}_{\kappa}(A, B)$ consists of the first κ columns of $\boldsymbol{K}^{\text{defl}}_{g(A,B),0}(A, B)$ using the grade $g(A, B)$ from Definition 6.8.

Before continuing a couple of remarks are in order:

- For the deflated block Krylov subspaces the relation

$$\mathcal{K}^{\text{defl}}_{k,j}(A, B) \subset \mathcal{K}^{\square}_{k,j}(A, B)$$

  holds if and only if no inexact deflation occurred until step (k, j).
- The inequality $\kappa \leq km+j$ holds and we have equality only if no deflation occurred whilst generating the matrix $\boldsymbol{K}^{\text{defl}}_{k,j}(A, B)$.
- We did not precisely state the process of inexact deflation. We will leave this to be defined implicitly by the processes that generate bases for the deflated block Krylov subspaces.
- The two definitions $\mathcal{K}^{\text{defl}}_{k,j}(A, B)$ and $\mathcal{K}^{\text{defl}}_{\kappa}(A, B)$ are, essentially, two different notations for the same spaces. In case of deflation we might have $\mathcal{K}^{\text{defl}}_{\kappa}(A, B) = \mathcal{K}^{\text{defl}}_{k,j}(A, B) = \mathcal{K}^{\text{defl}}_{k,j+1}(A, B)$. We can define a well-defined mapping from the κ-indices to the (k, j)-indices by always mapping κ to the smallest (k, j) assuming a lexicographical ordering. Hence, in the

following we implicitly assume a well-defined mapping from the κ- to the (k, j)-indices. In some places we use $\kappa(k, j)$ instead of κ as index where it is useful and makes referring to the deflated block Krylov subspace less ambiguous. But we have to keep in mind that $\kappa(k, j)$ is non-injective.

Now, we can use Definition 7.2 to define what we understand as a deflated block Krylov subspace method.

Definition 7.3. *Let $X^{(0)} = [x_1^{(0)}| \dots |x_m^{(0)}]$ and $R^{(0)} = B - AX^{(0)}$. We call iterative methods for solving $AX = B$ that generate iterates*

$$x_j^{(k)} \in x_j^{(0)} + \mathscr{K}_{k,0}^{\text{defl}}(A, R^{(0)}) \tag{7.4}$$

or

$$x_j^{(\kappa(k,j))} \in x_j^{(0)} + \mathscr{K}_{\kappa(k,j)}^{\text{defl}}(A, R^{(0)}) \tag{7.5}$$

deflated block Krylov subspace methods.

For future use we introduce the notation $X^{(k)} = [x_1^{(k)}| \dots |x_m^{(k)}]$. The first variant in Definition 7.3 emphasises that some methods always advance blockwise whereas the second variant is used for methods that advance by one vector at a time. Both coincide in some steps and the second one allows for intermediate iterates generated "between" the iterates $X^{(k)}$ and $X^{(k+1)}$.

7.1.2 Two-Sided Deflated Block Lanczos-Type Process

Having defined deflated block Krylov subspaces in Definition 7.2 we can now go ahead and describe the *Lanczos-type method* from [Ali+00]. We will refer to it as the *two-sided deflated block Lanczos-type process* to emphasise that it is not restricted to Hermitian matrices as the deflated block Lanczos-type process that we will introduce later. It can be regarded as a generalisation of the two-sided Lanczos process from Algorithm 2.3 that is capable of handling multiple starting vectors and includes proper deflation. We will not cover every detail of the method from [Ali+00] since we are ultimately interested in using a modification of the process for hpd matrices. Particularly, we omit the *look-ahead* capability of the process and the *cluster-wise bi-orthogonality* that comes along with it. This is only needed for non-Hermitian matrices to prevent serious breakdowns and we refer to [Ali+00] for details thereon. Note that in the following we will mostly use the terms bi-orthogonalisation and orthogonalisation synonymously since from context it is clear which one is meant.

In a nutshell, the two-sided deflated block Lanczos-type process from [Ali$^+$00] computes bi-orthogonal bases of dimension $\kappa = \kappa(k, j) = \tilde{\kappa}(\tilde{k}, \tilde{j})$ for the subspaces $\mathcal{K}^{\text{defl}}_{\kappa(k,j)}(A, R)$ and for $\mathcal{K}^{\text{defl}}_{\tilde{\kappa}(\tilde{k},\tilde{j})}(A^H, L)$. These fulfil a *block Lanczos-type relation* as do the matrices generated from the two-sided Lanczos process (Algorithm 2.3) fulfil (2.13), in the non-block case.

In detail, we want to generate right Lanczos vectors collected as columns of the matrix

$$V^{(\kappa(k,j))} = [v^{(1)}| \cdots |v^{(\kappa(k,j))}]$$

that span the subspace $\mathcal{K}^{\text{defl}}_{\kappa(k,j)}(A, R)$ and left Lanczos vectors

$$W^{(\tilde{\kappa}(\tilde{k},\tilde{j}))} = [w^{(1)}| \cdots |w^{(\tilde{\kappa}(\tilde{k},\tilde{j}))}]$$

that span the subspace $\mathcal{K}^{\text{defl}}_{\tilde{\kappa}(\tilde{k},\tilde{j})}(A^H, L)$. The matrices $R, L \in \mathbb{C}^{n\times m}$ are right and left starting vectors. In principle, one could choose a different number of left and right starting vectors as it is done in [Ali$^+$00], but in view of a Hermitian version of the process we restrict ourself to the same number m for both. Additionally, we want the columns of $V^{(\kappa(k,j))}$ and $W^{(\tilde{\kappa}(\tilde{k},\tilde{j}))}$ to be bi-orthogonal, i.e.

$$(w^{(i)})^H v^{(j)} = \begin{cases} \delta_i \neq 0 & \text{for } i = j \\ 0 & \text{for } i \neq j. \end{cases}$$

Methods for preventing a breakdown caused by $\delta_i = 0$ can be found in [Ali$^+$00]. Depending on the scaling of $w^{(i)}$ and $v^{(j)}$ one could force $\delta_i = 1$, but we choose to normalise $w^{(i)}$ and $v^{(j)}$. In matrix notation we can express this as

$$(W^{(\tilde{\kappa}(\tilde{k},\tilde{j}))})^H V^{(\kappa(k,j))} = \operatorname{diag}(\delta_1, \dots, \delta_\kappa) =: \Delta^{(\kappa)}. \tag{7.6}$$

In the following we will first sketch the two-sided deflated block Lanczos-type process without deflation, then we will discuss the implications of deflating vectors, afterwards we state the algorithms, and finally we will discuss the details that are easier to describe after having stated the algorithm.

Two-sided block Lanczos-type process without deflation

At first, we will describe the whole process for the non-deflated case. We assume that we already have bi-orthogonal bases

$$V^{(\kappa(k,j)-1)} = [v^{(1)}| \cdots |v^{(\kappa(k,j)-1)}] \text{ for } \mathcal{K}^{\square}_{\kappa(k,j)-1}(A, R) \text{ and}$$
$$W^{(\tilde{\kappa}(\tilde{k},\tilde{j})-1)} = [w^{(1)}| \cdots |w^{(\tilde{\kappa}(\tilde{k},\tilde{j})-1)}] \text{ for } \mathcal{K}^{\square}_{\tilde{\kappa}(\tilde{k},\tilde{j})-1}(A^H, L).$$

We will describe the procedure of creating these bases by explaining how we expand the bases of step $\kappa - 1$ to step κ. Since no deflation occurred we have that $\kappa(k, j) - 1 = mk + j - 1 = m\tilde{k} + \tilde{j} - 1 = \tilde{\kappa}(\tilde{k}, \tilde{j}) - 1$ and we simply use κ, k and j for both subspaces.

In step κ the pair of vectors $v^{(\kappa)}$ and $w^{(\kappa)}$ is computed. For $v^{(\kappa)}$, we do so by bi-orthogonalising $Av^{(\kappa(k,j)-m)} = Av^{(m(k-1)+j)}$ against $W^{(\kappa-1)}$ yielding $\boldsymbol{\upsilon}$. That is, we compute

$$\boldsymbol{\upsilon} = Av^{(\kappa-m)} - \sum_{i=\kappa-2m}^{\kappa-1} (w^{(i)})^H Av^{(\kappa-m)} v^{(i)}. \tag{7.7}$$

If $\kappa \leq m$ then we bi-orthogonalise $R_{:,\kappa}$ instead of $Av^{(\kappa-m)}$ against $W^{(\kappa-1)}$, again yielding $\boldsymbol{\upsilon}$. Then we set $v^{(\kappa)} = \boldsymbol{\upsilon} / \|\boldsymbol{\upsilon}\|$. The bi-orthogonalisation of $w^{(\kappa)}$ can be done in the same manner. Note that in equation (7.7) the sum does not need to start at $i = 1$, because for $i < \kappa - 2m$ the inner product $(w^{(i)})^H Av^{(\kappa-m)}$ is zero. This can be seen by observing that for $i < \kappa - 2m$ we have

$$\begin{aligned} (w^{(i)})^H Av^{(\kappa-m)} &= (A^H w^{(i)})^H v^{(\kappa-m)} \\ &= \left(\sum_{c=1}^{i+m} \star w^{(c)} \right)^H v^{(\kappa-m)}, \end{aligned} \tag{7.8}$$

and every vector $w^{(c)}$ in the sum is already orthogonal to $v^{(\kappa-m)}$ by construction (we used $\star$ as dummies for scalars whose values are not important for the observation). This is a property of the bi-orthogonalisation that allows for short recurrences.

We can formulate the bi-orthogonalisation procedure for $\kappa > m$ in matrix form as a *block Lanczos relation*

$$\begin{aligned} AV^{(\kappa-m)} &= V^{(\kappa)} \tilde{T}_v^{(\kappa,\kappa-m)} \\ A^H W^{(\kappa-m)} &= W^{(\kappa)} \tilde{T}_w^{(\kappa,\kappa-m)} \end{aligned} \tag{7.9}$$

where the matrices

$$\begin{aligned} &\tilde{T}_v^{(\kappa,\kappa-m)} \in \mathbb{C}^{\kappa \times \kappa - m} \quad \text{and} \\ &\tilde{T}_w^{(\kappa,\kappa-m)} \in \mathbb{C}^{\kappa \times \kappa - m} \end{aligned}$$

collect the bi-orthogonalisation coefficients. These matrices are banded with upper and lower bandwidth m. In other words, the matrices $\tilde{T}_v^{(\kappa,\kappa-m)}$ and $\tilde{T}_w^{(\kappa,\kappa-m)}$ are block tridiagonal with $m \times m$ blocks and triangular off-diagonal blocks. Figure 7.1 displays the structure of $\tilde{T}_v^{(\kappa,\kappa-m)}$, and $\tilde{T}_w^{(\kappa,\kappa-m)}$ has the same structure.

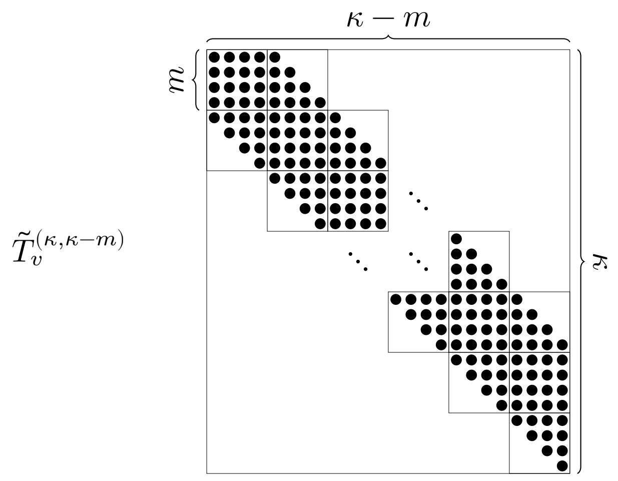


Figure 7.1: Example for the sparsity pattern of the matrix $\tilde{T}_v^{(\kappa,\kappa-m)}$ in case of no deflation for 4 left and right starting vectors.

Two-sided deflated block Lanczos-type process

Deflation makes the whole process more sophisticated since, amongst other reasons, it can happen independently in the two spaces $\mathcal{K}^{\text{defl}}_{\kappa(k,j)}(A, R)$ and $\mathcal{K}^{\text{defl}}_{\tilde{\kappa}(\tilde{k},\tilde{j})}(A^H, L)$. Thus, we need to be able to express this deflation of the left and the right Lanczos vectors. We will describe the process only for the right Lanczos vectors, but for the left Lanczos vectors it can be stated in the same way. Furthermore, we assume that the left and right starting vectors in L and R are free of (nearly) linearly dependent vectors.

Some of the variables in the upcoming algorithm and description have a lower index v or w which indicates that the corresponding variable is used in connection with the right deflated block Krylov subspace and right Lanczos vectors (v) or the left deflated block Krylov subspace and left Lanczos vectors (w). The following explanation mostly covers details related to the right Lanczos vectors. But, unless stated otherwise this can be trivially transferred to the left Lanczos vectors.

The indexing $\kappa(k, j)$ and $\tilde{\kappa}(\tilde{k}, \tilde{j})$ is useful for relating the iteration steps to the block Krylov subspaces. But for the upcoming description of the two-sided deflated block Lanczos-type process it is a bit clunky. Hence, for the moment we drop this notation and use the index μ_v for the Lanczos vector where the next candidate vector originates from as an A-multiple. Moreover, we use the index c for the Lanczos vector we currently want to compute.

The detailed description where the candidate Lanczos vectors are taken from and how c and μ_v are related is split in two parts. First, building upon the previous descriptions we briefly describe their relation until deflation occurs the first time in a paragraph using this new notation. Second, the deflated case is described in more detail afterwards and we give an example later that illustrates the deflation process even more. Since we finally want to build $\mathcal{K}_\kappa^{\text{defl}}(A, R)$, we have $c \leq \kappa$ and $\mu_v < c$ during the process and stop when $c = \kappa$ or $\mu_v = c$.

Before deflation. After an initial bi-orthogonalisation of the starting vectors that yields $v^{(1)}, \dots, v^{(m)}$ and $w^{(1)}, \dots, w^{(m)}$ we proceed by extending the right deflated block Krylov subspace by the A-multiple $Av^{(1)}$. In general, having already created $c-1$ left and right Lanczos vectors we use $Av^{(c-m)} = Av^{(\mu_v)}$ to extend the deflated block Krylov subspace—as long as no deflation was needed. Bi-orthogonalisation of $Av^{(\mu_v)}$ against the matrix $W^{(c-1)}$ leaves us with a vector $\boldsymbol{\upsilon}^{(\mu_v)}$. We call this vector $\boldsymbol{\upsilon}^{(\mu_v)}$ a *candidate Lanczos vector*. This emphasises that we might not use this vector to span $\mathcal{K}_c^{\text{defl}}(A, R)$ if it gets deflated. Note that the index of the candidate Lanczos vector corresponds to the index of the Lanczos vector it is created from and not the index of the Lanczos vector it might end up in.

Deflation indicator. We detect deflation by monitoring the norm $\|\boldsymbol{\upsilon}^{(\mu_v)}\|$ of the current candidate Lanczos vector. Deflation occurs for the first time when the bi-orthogonalisation leaves us with $\boldsymbol{\upsilon}^{(\mu_v)}$ as the zero vector (*exact deflation*) or as a vector with very small, non-zero norm (*inexact deflation*). We use the deflation tolerance tol_{defl} as a threshold and apply inexact deflation if $\varepsilon_{\text{mach}} < \|\boldsymbol{\upsilon}^{(\mu_v)}\| < \text{tol}_{\text{defl}}$. As stated in Section 6.3.3 we treat a vector with norm close to machine precision as a case of exact deflation. Thus, by setting $\text{tol}_{\text{defl}} = \varepsilon_{\text{mach}}$ we only allow exact deflation.

Handling deflation. If $\boldsymbol{\upsilon}^{(\mu_v)}$ is identified as (nearly) linearly dependent then we do not set $v^{(c)} = \boldsymbol{\upsilon}^{(\mu_v)} / \|\boldsymbol{\upsilon}^{(\mu_v)}\|$. Instead, we skip this $\boldsymbol{\upsilon}^{(\mu_v)}$ and call $\boldsymbol{\upsilon}^{(\mu_v)}$ a *deflated candidate vector* and we call $v^{(\mu_v)}$ a *deflated Lanczos vector* since its A-multiple was removed. If it is clear by context we use the term *deflated vector* in both cases. Then, we proceed with orthogonalising the next vector $Av^{(\mu_v+1)}$, and ideally we end up with a vector that does not need to be deflated. Otherwise, we repeat the process until a vector is generated that does not need to be deflated or the right deflated block Krylov subspace is exhausted. The tricky part is, that every time we deflate a vector the recurrence length decreases with profound impact on indexing in successive steps.

This means that we have to keep track which vector has to be multiplied next by A to expand the deflated block Krylov subspace further and which vectors we need to orthogonalise against. We will discuss the details—especially the bookkeeping of deflated vectors—after presenting the algorithm.

Algorithm of the two-sided deflated block Lanczos-type process. Algorithm 7.1 displays the two-sided deflated block Lanczos-type process. It is a stripped down version of the algorithm presented in [Ali⁺00]. We applied the simplifications mentioned so far, i.e. m left and right starting vectors and no look-ahead implying no need for cluster-wise bi-orthogonality. Furthermore, we assume that the starting vectors are not (nearly) linearly dependent starting vectors. This is a weak assumption since the algorithm can be modified easily to seamlessly handle a deflated bi-orthogonalisation of the starting vectors. The implementation that we use for numerical experiments in Chapter 8 includes this modification. We omit this here to simplify the presentation of the algorithm and not clutter it unnecessarily.

Since deflation may occur several times in the left and right spaces, we have to be careful with the indexing to use. In Algorithm 7.1, the index c is used as the counting index for the vectors of the bi-orthogonal bases. This means that after step c of the outermost loop we have created bases for $\mathscr{K}_c^{\mathrm{defl}}(A, R)$ and $\mathscr{K}_c^{\mathrm{defl}}(A^H, L)$. These bases are collected as columns of the matrix $V^{(c)}$ and the matrix $W^{(c)}$, respectively. Hence, the left and right deflated block Krylov subspaces always share the same dimension at the beginning and the end of the outermost loop in line 3.

Besides the index c we need another, already mentioned, index for the left Lanczos vectors and one for the right Lanczos vectors to access specific Lanczos vectors. The index μ_v counts the number of passes through the loop starting in line 15. It is used to keep track of the vector from which we get the next candidate right Lanczos vector $\boldsymbol{\upsilon}^{(\mu_v)}$ after the loop in line 7 by bi-orthogonalising $Av^{(\mu_v)}$.

The algorithm generates orthogonalisation coefficients t_{j,μ_v} and $\tilde{t}_{j,\mu_w}$ in the lines 8 and 22, respectively. These can be collected together with the norms from line 14 and line 28 in matrices

$$(\tilde{T}_v^{(c,\mu_v)})_{p,q} = \begin{cases} t_{p,q} & \text{if } p \in J_v^{(q)} \\ t_{p,q} & \text{if } h_v^{(p)} = q \text{ and } \left\|\boldsymbol{\upsilon}^{(q)}\right\| \geq \mathrm{tol}_{\mathrm{defl}} \\ 0 & \text{else} \end{cases}$$

Algorithm 7.1: Two-sided deflated block Lanczos-type process

```
Input  : $A \in \mathbb{C}^{n \times n}$                      system matrix
         $R \in \mathbb{C}^{n \times m}, L \in \mathbb{C}^{n \times m}$       right and left starting block-vector
         $\mathrm{tol}_{\mathrm{defl}}, \kappa$                      deflation tolerance, number of steps
Output : $V^{(\kappa)} = [v^{(1)}| \dots |v^{(\kappa)}]$        basis of $\mathcal{K}_\kappa^{\mathrm{defl}}(A, R)$, bi-orthogonal to $W^{(\kappa)}$
         $W^{(\kappa)} = [w^{(1)}| \dots |w^{(\kappa)}]$        basis of $\mathcal{K}_\kappa^{\mathrm{defl}}(A^H, L)$, bi-orthogonal to $V^{(\kappa)}$
         $\tilde{T}_v^{(\kappa, h_v^{(\kappa)})}, \tilde{T}_w^{(\kappa, h_v^{(\kappa)})}$       bi-orth. coefficients matrices, s.t. (7.11) holds

1  $\mu_v = \mu_w = 0, I_v = I_w = \emptyset, (h_v^{(1)}, \dots, h_v^{(m)}) = (h_w^{(1)}, \dots, h_w^{(m)}) = (1, \dots, 1)$
2  bi-orthogonalise $R$ and $L$ yielding $v^{(1)}, \dots, v^{(m)}$ and $w^{(1)}, \dots, w^{(m)}$
3  for $c = m+1, m+2, \dots$ do
4      repeat                                                        // build the next right Lanczos vector
5          $\mu_v = \mu_v + 1$; if $\mu_v = c$ then stop                   // right subspace depleted
6          $\boldsymbol{v}^{(\mu_v)} = A v^{(\mu_v)}$
7          for $j \in J_v^{(\mu_v)} = \left\{ h_w^{(\mu_v)}, \dots, c-1 \right\} \cup I_v$ do             // orthogonalisation
8              $t_{j,\mu_v} = (w^{(j)})^H v / \delta^{(j)}$
9              $\boldsymbol{v}^{(\mu_v)} = \boldsymbol{v}^{(\mu_v)} - t_{j,\mu_v} v^{(j)}$
10         if $\|\boldsymbol{v}^{(\mu_v)}\| = 0$ then discard vector $\boldsymbol{v}^{(\mu_v)}$                    // exact deflation
11         else if $\|\boldsymbol{v}^{(\mu_v)}\| < \mathrm{tol}_{\mathrm{defl}}$ then                         // inexact deflation
12             keep $v^{(\mu_v)}$ for later orthogonalisation and set $I_w = I_w \cup \{\mu_v\}$
13         else                                                      // new right Lanczos vector
14             $t_{c,\mu_v} = \|\boldsymbol{v}^{(\mu_v)}\|$
15             $v^{(c)} = \boldsymbol{v}^{(\mu_v)} / t_{c,\mu_v}$
16             $h_v^{(c)} = \mu_v$
17     until non-deflated new vector $v^{(c)}$ computed in line 15
18     repeat                                                        // build the next left Lanczos vector
19         $\mu_w = \mu_w + 1$; if $\mu_w = c$ then stop                   // left subspace depleted
20         $\boldsymbol{w}^{(\mu_w)} = A^H w^{(\mu_w)}$
21         for $j \in J_w^{(\mu_w)} = \left\{ h_v^{(\mu_w)}, \dots, c-1 \right\} \cup I_w$ do             // orthogonalisation
22             $\tilde{t}_{j,\mu_w} = (v^{(j)})^H w / \delta^{(j)}$
23             $\boldsymbol{w}^{(\mu_w)} = \boldsymbol{w}^{(\mu_w)} - \tilde{t}_{j,\mu_w} w^{(j)}$
24         if $\|\boldsymbol{w}^{(\mu_w)}\| = 0$ then discard vector $\boldsymbol{w}^{(\mu_w)}$                    // exact deflation
25         else if $\|\boldsymbol{w}^{(\mu_w)}\| < \mathrm{tol}_{\mathrm{defl}}$ then                         // inexact deflation
26             keep $w^{(\mu_w)}$ for later orthogonalisation and set $I_v = I_v \cup \{\mu_w\}$
27         else                                                      // new left Lanczos vector
28             $\tilde{t}_{c,\mu_w} = \|\boldsymbol{w}^{(\mu_w)}\|$
29             $w^{(c)} = \boldsymbol{w}^{(\mu_w)} / \tilde{t}_{c,\mu_w}$
30             $h_w^{(c)} = \mu_w$
31     until non-deflated new vector $w^{(c)}$ computed in line 29
32     $\delta^{(c)} = (w^{(c)})^H v^{(c)}$
33     if $\delta^{(c)} = 0$ then stop                                 // serious breakdown
```

and

$$(\tilde{T}_w^{(c,\mu_w)})_{p,q} = \begin{cases} \tilde{t}_{p,q} & \text{if } p \in J_w^{(q)} \\ \tilde{t}_{p,q} & \text{if } h_w^{(p)} = q \text{ and } \left\|\boldsymbol{\omega}^{(q)}\right\| \geq \mathrm{tol}_{\mathrm{defl}} \\ 0 & \text{else.} \end{cases}$$

We stress that even the coefficients of deflated candidate vectors are kept in these matrices but not their norms as they are removed and not normalised. After some further description of the algorithm we illustrate the sparsity patterns of these matrices using an example.

Algorithm 7.1 can break down in three places. First, in line 33 the algorithm can stop with $\delta^{(c)} = 0$ with $w^{(c)} \neq 0 \neq v^{(c)}$. This is called a serious breakdown and can be cured in most cases by using look-ahead techniques leading to cluster-wise bi-orthogonality as described in [Ali$^+$00]. Second, if the algorithm stops in line 5 then the right deflated block Krylov subspace is depleted. This indicates that we found a (nearly) A-invariant subspace. A deflated block Krylov subspace method implemented on top of the two-sided deflated block Lanczos-type process should be able to make use of this and have converged to an approximate solution by then. Third, if the algorithm stops in line 19 then the left deflated block Krylov subspace is exhausted and we found a (nearly) A^H-invariant subspace. If the goal is to solve $AX = B$ then one could restart with a new set of left starting vectors spanning a different left deflated block Krylov subspace.

Keeping deflated vectors. The deflated vectors $v^{(\mu_v)}$ whose corresponding candidate vectors $\boldsymbol{\upsilon}^{(\mu_v)} \neq 0$ end up not being added to the basis of the deflated block Krylov subspace have to be saved for later use. We need them for preserving the bi-orthogonality of the bases for the left and right deflated block Krylov subspaces. The indices of the inexactly deflated vectors that are created while building the right deflated block Krylov subspace are kept in the set I_v and we define

$$I_v^{(\mu)} := \{i : i \in I_v \wedge i \leq \mu\}\,.$$

Although not explicitly built by the algorithm, we introduce the matrix $\tilde{V}_{\mathrm{defl}}^{(\mu_v)}$ here. It will turn out to be useful for theoretical considerations later on. The matrix $\tilde{V}_{\mathrm{defl}}^{(\mu_v)}$ has the same dimensions as $V^{(\mu_v)}$ and collects those of the candidate vectors $\boldsymbol{\upsilon}^{(1)}, \ldots, \boldsymbol{\upsilon}^{(\mu_v)}$ that are inexactly deflated. It can be defined via its columns

$$(\tilde{V}_{\mathrm{defl}}^{(\mu_v)})_{:,\mu} = \begin{cases} \boldsymbol{\upsilon}^{(\mu)} & \text{if } \mu \in I_v^{(\mu_v)} \\ 0 & \text{else.} \end{cases} \tag{7.10}$$

Hence, if $\boldsymbol{v}^{(\mu)}$ is inexactly deflated then column μ of $\tilde{V}_{\text{defl}}^{(\mu_v)}$ is set to $\boldsymbol{v}^{(\mu)}$. Note that we do not normalise these deflated vectors, hence they have norm smaller than the deflation tolerance tol_{defl}. All other columns of $\tilde{V}_{\text{defl}}^{(\mu_v)}$ are zero.

History indices. The *history index* $h_v^{(c)}$ is set to the index μ_v belonging to the last run through the loop in line 15 while generating the Lanczos vector $v^{(c)}$. In other words, $h_v^{(c)}$ denotes the index of the vector from which $v^{(c)}$ was computed by orthogonalising $Av^{(h_v^{(c)})}$ yielding $\boldsymbol{v}^{(h_v^{(c)})}$ and normalising $v^{(c)} = \boldsymbol{v}^{(h_v^{(c)})}/\|\boldsymbol{v}^{(h_v^{(c)})}\|$ thereafter. A useful property of the history indices is that two successive history indices can tell the number of deflated vectors between the construction of $v^{(c-1)}$ and $v^{(c)}$ as $h_v^{(c)} - h_v^{(c-1)} - 1$. Using the history indices and the index c we can define the right reduced block size as

$$m_v^{(c)} := c - h_v^{(c)} \leq m$$

and the number

$$d_v^{(c)} := m - m_v^{(c)}$$

of deflated right candidate Lanczos vectors until step c.

The main purpose of the history index, however, is to record against which vectors we still need to orthogonalise. For example, we initialise $h_v^{(1)} = \ldots = h_v^{(m)} = 1$ indicating that initial orthogonalisation up to step m has to start with $v^{(1)}$. By the same argument as in (7.8) we see that in case of no deflation or exact deflation we only need to bi-orthogonalise the candidate vector $Av^{(\mu_v)}$ against $w^{(h_w^{(\mu_v)})}, \ldots, w^{(c-1)}$. We illustrate this in the upcoming example. Unless deflation happened we have $\mu_v = \mu_w = c - m$ and $h_v^{(\mu_v)} = h_w^{(\mu_w)} = c - 2m$. Hence, we always have to orthogonalise against the last $2m$ vectors. If deflation occurred then the right candidate Lanczos vector $\boldsymbol{v}^{(\mu_v)}$ is explicitly orthogonalised against the last

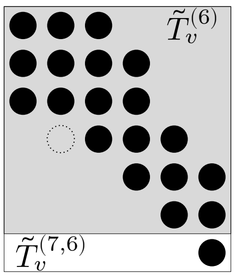
$$2m - d_v^{(c)} - d_w^{(\mu_v)} = m_v^{(c)} + m_w^{(\mu_v)}$$

vectors and the left Lanczos vectors whose A-multiples were inexactly deflated. We find these vectors using the set $I_v^{(\mu_v)}$ even they are not amongst the last $m_v^{(c)} + m_w^{(\mu_v)}$ vectors any more. Knowing that we have to orthogonalise against the last $m_v^{(c)} + m_w^{(\mu_v)}$ Lanczos vectors we can compute the index

$$\begin{aligned} c - (m_v^{(c)} + m_w^{(\mu_v)}) &= h_v^{(c)} - m_w^{(\mu_v)} \\ &= h_v^{(c)} + \mu_v - \mu_v - m_w^{(\mu_v)} \\ &= \underbrace{h_v^{(c)} - \mu_v}_{=0} + h_w^{(\mu_v)} \\ &= h_w^{(\mu_v)} \end{aligned}$$

of the Lanczos vector at which the orthogonalisation has to start in line 7 of Algorithm 7.1.

An example. Since the explanation so far introduced a lot of notation but is crucial for understanding the deflation process we want to elaborate on that using an example. In our example we start with two left starting vectors l_1, l_2 and two right starting vectors r_1, r_2. Inexact deflation occurs while creating the fourth right Lanczos vector by realising after orthogonalisation that $Av^{(2)}$ is nearly linearly dependent on the previous vectors $v^{(1)}, v^{(2)}$ and $v^{(3)}$. Additionally, we want exact deflation to occur while creating the fifth left Lanczos vector. Here and later on we will use the abbreviations $\tilde{T}_v^{(c)} = (\tilde{T}_v^{(\mu,c)})_{1:c,1:c}$ and $\tilde{T}_w^{(c)} = (\tilde{T}_w^{(\mu,c)})_{1:c,1:c}$. In Figure 7.2 we portray the sparsity patterns of $\tilde{T}_v^{(7,6)}$ collecting the orthogonalisation coefficients for the right Lanczos vectors and $\tilde{T}_w^{(7,6)}$ for the left Lanczos vectors, respectively.



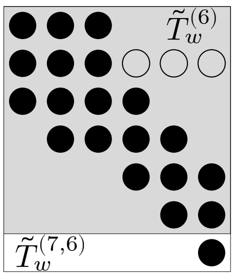


Figure 7.2: Example for the sparsity patterns of the matrices $\tilde{T}_v^{(7,6)}$ and $\tilde{T}_v^{(6)}$ as well as $\tilde{T}_w^{(7,6)}$ and $\tilde{T}_w^{(6)}$ generated by Algorithm 7.1 in case of inexact deflation in step $c = 4$ in the right Lanczos vector sequence and exact deflation in step $c = 5$ in the left Lanczos vector sequence for 2 left and right starting vectors. The circles ● represent non-zero entries. Empty circles ○ represent entries of small magnitude arising due to orthogonalisation against inexactly deflated vectors. Empty dotted circles ◌ represent computed values of small magnitude being the norm of inexactly deflated vectors and are not actually included in the matrices.

Table 7.1 lists the indices, history indices and sets throughout every cycle through the loops in line 17 and line 31 for our example. We display the process until 7 left and right Lanczos vectors have been generated. The same outer index can appear in multiple rows of the table for those steps in which deflation occurs. To emphasise that deflation occurred we display dashes (—) for the values that are only generated as soon as a new Lanczos vector is constructed. We give an example on how to read the table using the second row. It can be read as:

- we want to generate the $(c = 4)$-th Lanczos vector,
- the candidate right Lanczos vector $\boldsymbol{v}^{(2)}$ is computed from $v^{(\mu_v)} = v^{(2)}$,
- we have to orthogonalise $Av^{(2)}$ against the vectors from $v^{(h_w^{(\mu_v)})} = v^{(1)}$ to $v^{(c-1)} = v^{(3)}$,
- we realise that this candidate vector has to be deflated as indicated by an unset value $(-)$ of $h_v^{(c)}$ and
- the deflation is an inexact deflation, since μ_v is added to the set I_w.

In the third row the fourth right Lanczos vector is successfully generated, because the row displays all the information gathered after leaving the loops in line 17 and line 31. In the fourth row of the table the exact deflation in the left Lanczos vector sequence can be found.

#	c	μ_v	$h_w^{(\mu_v)}$	$h_v^{(c)}$	μ_w	$h_v^{(\mu_w)}$	$h_w^{(c)}$	$I_v^{(\mu_w)}$	$I_w^{(\mu_v)}$
1	3	1	1	1	1	1	1	$\emptyset$	$\emptyset$
2	4	2	1	–	–	–	–	$\emptyset$	$\{2\}$
3	4	3	1	3	2	1	2	$\emptyset$	$\{2\}$
4	5	–	–	–	3	1	–	$\emptyset$	$\{2\}$
5	5	4	2	4	4	2	4	$\emptyset$	$\{2\}$
6	6	5	3	5	5	3	5	$\emptyset$	$\{2\}$
7	7	6	4	6	6	4	6	$\emptyset$	$\{2\}$

Table 7.1: Values taken by the index variables and sets of Algorithm 7.1 in case of inexact deflation in step $c = 4$ in the right Lanczos vector sequence and exact deflation in step $c = 5$ in the left Lanczos vector sequence for 2 left and right starting vectors.

We want to make use of the example one last time. Our paramount interest is in keeping all the right Lanczos vectors bi-orthogonal to the left Lanczos vectors. Not adding inexactly deflated vectors to the corresponding basis has the potential to break the bi-orthogonality. We demonstrate how bi-orthogonality is kept by describing a few of the vector relations that can be derived from the matrices $\tilde{T}_v^{(7,6)}$ and $\tilde{T}_w^{(7,6)}$ in our example. In Figure 7.3 we stated all the relations that are represented by these matrices. We use $\star$ as a placeholder for those coefficients that are not important for our descriptions. Furthermore, for the relations describing the computation of $v^{(i)}$ and $w^{(i)}$ we note that the previous Lanczos vectors $v^{(1)}, \dots, v^{(i-1)}$ and $w^{(1)}, \dots, w^{(i-1)}$ are already bi-orthogonal. We want to describe two particular relations from Figure 7.3 in more detail.

$$\boxed{\begin{array}{ll}
\star v^{(3)} = Av^{(1)} - \star v^{(1)} - \star v^{(2)} & \star w^{(3)} = A^H w^{(1)} - \star w^{(1)} - \star w^{(2)} \\
\underbrace{\boldsymbol{v}^{(2)}}_{\approx 0} = Av^{(2)} - \star v^{(1)} - \star v^{(2)} - \star v^{(3)} & \\
\star v^{(4)} = Av^{(3)} - \star v^{(1)} - \star v^{(2)} - \star v^{(3)} & \star w^{(4)} = A^H w^{(2)} - \star w^{(1)} - \star w^{(2)} - \star w^{(3)} \\
 & \underbrace{\boldsymbol{w}^{(3)}}_{=0} = A^H w^{(3)} - \star w^{(1)} - \star w^{(2)} - \star w^{(3)} - \star w^{(4)} \\
\star v^{(5)} = Av^{(4)} - \star v^{(2)} - \star v^{(3)} - \star v^{(4)} & \star w^{(5)} = A^H w^{(4)} - \star w^{(2)} - \star w^{(3)} - \star w^{(4)} \\
\star v^{(6)} = Av^{(5)} - \star v^{(4)} - \star v^{(5)} & \star w^{(6)} = A^H w^{(5)} - \alpha w^{(2)} - \star w^{(4)} - \star w^{(5)} \\
\star v^{(7)} = Av^{(6)} - \star v^{(5)} - \star v^{(6)} & \star w^{(7)} = A^H w^{(6)} - \star w^{(2)} - \star w^{(5)} - \star w^{(6)}
\end{array}}$$

Figure 7.3: Example of recurrence relations in the two-sided deflated block Lanczos-type process for 2 left and right starting vectors. Inexact deflation occurs in step $c = 4$ in the right Lanczos vector sequence and exact deflation in step $c = 5$ in the left Lanczos vector sequence. We display the left Lanczos vectors $v^{(3)}, \dots, v^{(7)}$ and the right Lanczos vectors $w^{(3)}, \dots, w^{(7)}$ including the deflated candidate Lanczos vectors $\boldsymbol{v}^{(2)}$ and $\boldsymbol{w}^{(3)}$. We use $\star$ as placeholder for scalars that we do not need to specify for this example.

The first relation we want to discuss is

$$\star v^{(6)} = Av^{(5)} - \star v^{(4)} - \star v^{(5)}.$$

We can check that $v^{(6)}$ is already orthogonal to $w^{(3)}$. Even if we would assume that the orthogonalisation would involve the Lanczos vectors $v^{(1)}, v^{(2)}$ and $v^{(3)}$ we see that

$$\begin{aligned}
\star (w^{(3)})^H v^{(6)} &= (w^{(3)})^H (Av^{(5)} - \star v^{(1)} - \star v^{(2)} - \alpha v^{(3)} - \star v^{(4)} - \star v^{(5)}) \\
&= (w^{(3)})^H A v^{(5)} - \alpha (w^{(3)})^H v^{(3)} \\
&= (A^H w^{(3)})^H v^{(5)} - \alpha (w^{(3)})^H v^{(3)} \\
&= (\star w^{(1)} + \star w^{(2)} + \star w^{(3)} + \star w^{(4)})^H v^{(5)} - \alpha (w^{(3)})^H v^{(3)} \\
&= -\alpha (w^{(3)})^H v^{(3)}.
\end{aligned}$$

The next to last equality holds by using the relation for the candidate Lanczos vector $\boldsymbol{w}^{(3)}$ as stated in Figure 7.3 and the knowledge that $\boldsymbol{w}^{(3)}$ was removed due to exact deflation, which means that $\boldsymbol{w}^{(3)} = 0$. Hence, $\alpha = 0$ ensures orthogonality of $w^{(3)}$ and $v^{(6)}$ and explicit orthogonalisation against $v^{(1)}, v^{(2)}$ and $v^{(3)}$ is not necessary.

The second relation we want to discuss is

$$\star w^{(6)} = A^H w^{(5)} - \alpha w^{(2)} - \star w^{(4)} - \star w^{(5)}$$

as an example how inexact deflation in one Lanczos vector sequence influences the orthogonalisation process in the other sequence. We do not have to orthogonalise against $v^{(3)}$ involving $w^{(3)}$ explicitly for the same reason as before. But to keep orthogonality against those vectors whose A-multiple got inexactly deflated, we can utilise the inexactly deflated candidate vectors to obtain orthogonalisation coefficients. In our example this is the case for $v^{(2)}$ and we have

$$\begin{aligned}
\star(v^{(2)})^H w^{(6)} &= (v^{(2)})^H (A^H w^{(5)} - \alpha w^{(2)} - \star w^{(4)} - \star w^{(5)}) \\
&= (v^{(2)})^H A^H w^{(5)} - \alpha (v^{(2)})^H w^{(2)} \\
&= (A v^{(2)})^H w^{(5)} - \alpha (v^{(2)})^H w^{(2)} \\
&= (\boldsymbol{\upsilon}^{(2)} + \star v^{(1)} + \star v^{(2)} + \star v^{(3)})^H w^{(5)} - \alpha (v^{(2)})^H w^{(2)} \\
&= (\boldsymbol{\upsilon}^{(2)})^H w^{(5)} - \alpha (v^{(2)})^H w^{(2)}.
\end{aligned}$$

Hence, computing α as

$$\alpha = \frac{(v^{(2)})^H A^H w^{(5)}}{(v^{(2)})^H w^{(2)}} = \frac{(\boldsymbol{\upsilon}^{(2)})^H w^{(5)}}{(v^{(2)})^H w^{(2)}} = \frac{(\boldsymbol{\upsilon}^{(2)})^H w^{(5)}}{\delta^{(2)}}$$

results in $(v^{(2)})^H w^{(6)} = 0$ thereby ensuring bi-orthogonality. This is the rationale behind keeping the deflated vectors like $v^{(2)}$ and their indices in the sets I_v and I_w. Note that in the algorithm we compute α using $(v^{(2)})^H A w^{(5)}$. For theoretical considerations, however, the value $(\boldsymbol{\upsilon}^{(2)})^H w^{(5)}$ is important since its norm is bound by $\|(\boldsymbol{\upsilon}^{(2)})^H w^{(5)}\| \leq \mathrm{tol}_{\mathrm{defl}}$.

A deflated block Lanczos-type relation. Inexactly deflating candidate vectors invalidates the relations (7.9). But we can state a *deflated block Lanczos-type relation* that the matrices generated by Algorithm 7.1 fulfil

$$\begin{aligned}
AV^{(\mu_v)} &= V^{(c)} \tilde{T}_v^{(c,\mu_v)} + \tilde{V}_{\mathrm{defl}}^{(\mu_v)} \\
AW^{(\mu_w)} &= W^{(c)} \tilde{T}_w^{(c,\mu_w)} + \tilde{W}_{\mathrm{defl}}^{(\mu_w)}.
\end{aligned} \tag{7.11}$$

Here, we used the matrix $\tilde{V}_{\mathrm{defl}}^{(\mu_v)}$ as it is defined in (7.10).

If we want to base deflated block Krylov subspace methods on the two-sided deflated block Lanczos-type process then we need an equivalent formulation of (7.11) in which we write the matrix $\tilde{T}_v^{(c,\mu_v)}$ as the sum

$$\tilde{T}_v^{(c,\mu_v)} = T_v^{(c,\mu_v)} + S_v^{(c,\mu_v)}.$$

The matrix $S_v^{(c,\mu_v)}$ contains all the entries of $T_v^{(c,\mu_v)}$ that arise while orthogonalising against inexactly deflated vectors (empty solid circles $\bigcirc$ in Figure 7.2) from the left Lanczos vector sequence. The matrix $T_v^{(c,\mu_v)}$, on the other hand, contains the remaining entries (only the filled circles ● in Figure 7.2) of $\tilde{T}_v^{(c,\mu_v)}$. Note that the empty dotted circles (◌) in Figure 7.2) are not actually present entries in $\tilde{T}_v^{(c,\mu_v)}$ but are there to indicate where deflation occurred. Now, we can rewrite (7.11) as

$$\begin{aligned} AV^{(\mu_v)} &= V^{(c)}\tilde{T}_v^{(c,\mu_v)} + \tilde{V}_{\text{defl}}^{(\mu_v)} \\ &= V^{(c)}(T_v^{(c,\mu_v)} + S_v^{(c,\mu_v)}) + \tilde{V}_{\text{defl}}^{(\mu_v)} \\ &= V^{(c)}T_v^{(c,\mu_v)} + V^{(c)}S_v^{(c,\mu_v)} + \tilde{V}_{\text{defl}}^{(\mu_v)} \\ &= V^{(c)}T_v^{(c,\mu_v)} + V_{\text{defl}}^{(\mu_v)} \end{aligned} \tag{7.12}$$

with $V_{\text{defl}}^{(\mu_v)} = V^{(c)}S_v^{(c,\mu_v)} + \tilde{V}_{\text{defl}}^{(\mu_v)}$.

The following proposition quantifies how much of an error inexact deflation introduces.

Proposition 7.4. *Let all the matrices and indices be defined as before. Then for the matrices $\tilde{V}_{\text{defl}}^{(\mu_v)}$ and $V_{\text{defl}}^{(\mu_v)}$ the following bounds hold*

$$\left\|\tilde{V}_{\text{defl}}^{(\mu_v)}\right\|_F \leq \text{tol}_{\text{defl}}\sqrt{d_v^{(\mu_v)}} \tag{7.13}$$

$$\left\|V_{\text{defl}}^{(\mu_v)}\right\|_F \leq \text{tol}_{\text{defl}}\sqrt{d_v^{(\mu_v)} + \frac{\mu_v}{\left|\min_{i\in\{1,\ldots,\mu_v\}}\{\delta^{(i)}\}\right|^2}d_w^{(\mu_v)}}. \tag{7.14}$$

Proof. We note that $\tilde{V}_{\text{defl}}^{(\mu_v)}$ consists of at most $d_v^{(\mu_v)}$ non-zero columns having each norm less or equal to tol_{defl} and this shows that (7.13) holds. In addition to the bound (7.13) that can be found in [Ali⁺00] we added (7.14) which we will prove next.

The matrix $S_v^{(c,\mu_v)}$ contains the orthogonalisation coefficients from orthogonalising against inexactly deflated vectors in the left Lanczos vector sequence. Therefore, at most $d_w^{(\mu_v)}$ rows in $S_v^{(c,\mu_v)}$ contain non-zero entries of magnitude not larger than

$$\left|\frac{\text{tol}_{\text{defl}}}{\min_{i\in\{1,\ldots,\mu_v\}}\{\delta^{(i)}\}}\right|.$$

Using $\tilde{\delta} = \min_{i\in\{1,\ldots,\mu_v\}}\{\delta^{(i)}\}$ we can show that the inequality (7.13) holds since

$$\left\|V_{\text{defl}}^{(\mu_v)}\right\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{\mu_v}\left|(V^{(c)}S_v^{(c,\mu_v)} + \tilde{V}_{\text{defl}}^{(\mu_v)})_{i,j}\right|^2}$$

$$
\begin{aligned}
&\le \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{\mu_v}\left|(V^{(c)}S_v^{(c,\mu_v)})_{i,j}\right|^2 + d_v^{(\mu_v)}\mathrm{tol}_{\mathrm{defl}}^2} \\
&= \sqrt{\sum_{i\in I_w^{(\mu_v)}}\sum_{j=1}^{\mu_v}\|v^{(i)}\|^2\left|(S_v^{(c,\mu_v)})_{i,j}\right|^2 + d_v^{(\mu_v)}\mathrm{tol}_{\mathrm{defl}}^2} \\
&\le \sqrt{d_w^{(\mu_v)}\mu_v\frac{\mathrm{tol}_{\mathrm{defl}}^2}{\left|\tilde{\delta}\right|^2} + d_v^{(\mu_v)}\mathrm{tol}_{\mathrm{defl}}^2} \\
&= \mathrm{tol}_{\mathrm{defl}}\sqrt{d_w^{(\mu_v)}\frac{\mu_v}{\left|\tilde{\delta}\right|^2} + d_v^{(\mu_v)}}. \qquad \square
\end{aligned}
$$

In the light of Proposition 7.4 we see that small values $\delta^{(i)}$ can result in a large norm $\left\|V_{\mathrm{defl}}^{(\mu_v)}\right\|_F$. We will see that the deflated block Lanczos-type process for Hermitian matrices in Section 7.1.3 does not suffer from this but for the two-sided version it is a reason to implement cluster-wise bi-orthogonality as in [Ali$^+$00].

With (7.12) we arrived at our final version of a deflated block Lanczos-type relation. Ultimately, we are only interested in this relation for the right Lanczos vectors, but could, of course, apply the same considerations to the left Lanczos vectors resulting in

$$A^H W^{(\mu_w)} = W^{(c)}T_w^{(c,\mu_w)} + W_{\mathrm{defl}}^{(\mu_w)}.$$

Finally, we remark that the following relations hold

$$
\begin{aligned}
(W^{(\mu)})^H A V^{(\mu)} &= \Delta^{(\mu)}T_v^{(\mu)} + (W^{(\mu)})^H V_{\mathrm{defl}}^{(\mu)} \\
(V^{(\mu)})^H A^H W^{(\mu)} &= \Delta^{(\mu)}T_w^{(\mu)} + (V^{(\mu)})^H W_{\mathrm{defl}}^{(\mu)}
\end{aligned}
$$

with $\Delta^{(i)} = \mathrm{diag}\left(\delta^{(1)},\dots,\delta^{(i)}\right)$ as in (7.6).

7.1.3 Deflated Block Lanczos-Type Process for Hermitian Matrices

In this subsection we develop a deflated block Lanczos-type process for Hermitian matrices stemming from the two-sided deflated block Lanczos-type process that we have presented in Subsection 7.1.2 as a new contribution to deflated block Lanczos-type processes. A less detailed description has been published before in [BF14].

Several simplifications apply if the two-sided deflated block Lanczos-type process from Algorithm 7.1 is specialised for Hermitian matrices. Like before, we will first roughly describe the process we want to develop, then we state the algorithm and discuss the details of the process before ending the subsection with an example.

The notation in this subsection can be simpler than that of the previous subsection. For instance, we do not have to indicate to which subspace an index or vector belongs since it is sufficient to span only one deflated block Krylov subspace and still benefit from short recurrences. In the following we will describe the process and introduce the nomenclature used in the Hermitian case very briefly since most of it is directly descendant from the process introduced in Section 7.1.2. We focus on the changes that are more than just a substitution of w by v or the like. In a few places, however, we give a more formal definition of the entities used in the resulting algorithm.

We adopt the indexing from the two-sided process introduced before. The index c is used for the new Lanczos vector which is to be computed in the current iteration step. The index μ indicates the Lanczos vector of which the next candidate Lanczos vector is an A-multiple.

Overview of the process. The deflated block Krylov subspace that we want to compute is built from a block-vector B consisting of the m starting vectors $b_1, \dots, b_m$. In contrast to the two-sided deflated block Lanczos-type process we want to include the deflation and orthogonalisation of the starting vectors seamlessly which we briefly mentioned for Algorithm 7.1. Thus, the orthonormalisation of the starting vectors, that we start the deflated block Lanczos-type process for Hermitian matrices with, yields the first $m_0 \leq m$ Lanczos vectors $v^{(1)}, \dots, v^{(m_0)}$. From there we proceed by extending the deflated block Krylov subspace by $Av^{(1)}$. As before, in general we orthonormalise the vector $Av^{(\mu)}$, which unless deflations happened in a previous step is the vector $Av^{(c-m)}$, against $V^{(c-1)} = [v^{(1)}| \cdots |v^{(c-1)}]$ to build the candidate Lanczos vector $\boldsymbol{\upsilon}^{(\mu)}$. We check the norm of $\boldsymbol{\upsilon}^{(\mu)}$ to decide if deflation needs to be applied, and if we decide that the candidate Lanczos vector has to be removed then we continue with computing the next candidate Lanczos vector. From there on $\mu > c - m$ and we repeat this until we obtain a candidate Lanczos vector that does not need to be deflated. We then normalise this vector, call it $v^{(c)}$ and build the matrix

$$V^{(c)} = [v^{(1)}| \cdots |v^{(c)}]. \tag{7.15}$$

This whole process can be repeated until $\mu = c$, and in this case we reached a (nearly) A-invariant subspace $\mathscr{K}_c^{\text{defl}}(A, B)$.

Algorithm 7.2: Deflated block Lanczos-type process

Input :	$A \in \mathbb{C}^{n\times n}$	system matrix, Hermitian
	$B = [b_1, \dots, b_m] \in \mathbb{C}^{n\times m}$	starting block-vector
	$\mathrm{tol}_{\mathrm{defl}}$	deflation tolerance
	κ	number of steps to perform
Output:	$V^{(\kappa)} = [v^{(1)}\mid \dots \mid v^{(\kappa)}]$	orthonormal basis of $\mathcal{K}^{\mathrm{defl}}_{\kappa}(A, B)$
	$\tilde{T}^{(\kappa, h^{(\kappa)})}$	matrix containing orthogonalisation coefficients such that (7.19) holds

```
1  $\mu = -m, I = \emptyset, (h^{(-m+1)}, \dots, h^{(0)}) = (1, \dots, 1)$
2  for $c = 1, 2, \dots$ do
3      repeat                                                         // build the next Lanczos vector
4          $\mu = \mu + 1$; if $\mu = c$ then stop                    // block Krylov subspace depleted
5          if $\mu \leq 0$ then
6              $\boldsymbol{v}^{(\mu)} = b_{m+\mu}$
7          else
8              $\boldsymbol{v}^{(\mu)} = Av^{(\mu)}$
9          for $j \in J^{(\mu)} = \{h^{(\mu)}, \dots, c-1\} \cup I$ do   // orthogonalisation
10             $t_{j,\mu} = (v^{(j)})^H v$
11             $\boldsymbol{v}^{(\mu)} = \boldsymbol{v}^{(\mu)} - t_{j,\mu} v^{(j)}$
12         if $\|\boldsymbol{v}^{(\mu)}\| = 0$ then discard vector $\boldsymbol{v}^{(\mu)}$   // exact deflation
13         else if $\|\boldsymbol{v}^{(\mu)}\| < \mathrm{tol}_{\mathrm{defl}}$ and $\mu > 0$ then   // inexact deflation
14             keep $v^{(\mu)}$ for later orthogonalisation and set $I = I \cup \{\mu\}$
15         else                                                       // new Lanczos vector
16             $t_{c,\mu} = \|\boldsymbol{v}^{(\mu)}\|$
17             $v^{(c)} = \boldsymbol{v}^{(\mu)}/t_{c,\mu}$
18             $h^{(c)} = \max\{1, \mu\}$
19     until non-deflated new vector $v^{(c)}$ computed in line 17
```

The algorithm. Algorithm 7.2 displays the deflated block Lanczos-type process for Hermitian matrices. It can be regarded as a specialised version of Algorithm 7.1 which originates from [Ali+00]. A notable difference to the two-sided version of the algorithm is that we only have one index μ instead of μ_v and μ_w. The index μ counts every generated candidate Lanczos vector which allows us to use μ to unambiguously describe the progress of Algorithm 7.2. Therefore, from now on we primarily use μ as the iteration index and $c^{(\mu)}$ is the corresponding index of the newly computed Lanczos vector in iteration step μ. The deflated orthogonalisation of the starting vectors is implemented by starting with $\mu = -m$, $c = 1$ and some minor special treatments for the indices $\mu \leq 0$.

History indices. The *history index* $h^{(c^{(\mu)})}$ is set to contain the index μ belonging to the last run through the loop in line 19 whilst generating the Lanczos vector $v^{(c^{(\mu)})}$. More precisely, we set

$$h^{(c^{(\mu)})} := \max\left\{\tilde{\mu} : c^{(\tilde{\mu})} = c^{(\mu)}\right\},$$

except for the special case of the starting vectors where we set $h^{(c^{(\mu)})} = 1$. In the algorithm we initialise $h^{(-m+1)} = \ldots = h^{(0)} = 1$ indicating that initial orthogonalisation has to start with $v^{(1)}$. The *reduced block size* is defined as

$$m^{(c^{(\mu)})} := c^{(\mu)} - h^{(c^{(\mu)})} \leq m$$

and

$$d^{(c^{(\mu)})} := m - m^{(c^{(\mu)})}$$

is the number of deflated candidate Lanczos vectors up to step $c^{(\mu)}$. As for the two-sided algorithm we can obtain the index of the Lanczos vector at which orthogonalisation has to start with in line 9 of Algorithm 7.2 as

$$\begin{aligned} c - (m^{(c^{(\mu)})} + m^{(\mu)}) &= h^{(c^{(\mu)})} - m^{(\mu)} \\ &= h^{(c^{(\mu)})} + \mu - \mu - m^{(\mu)} \\ &= \underbrace{h^{(c^{(\mu)})} - \mu}_{=0} + h^{(\mu)} \\ &= h^{(\mu)}. \end{aligned}$$

A deflated block Lanczos-type relation. On our way towards a deflated block Lanczos-type relation for the deflated block Lanczos-type process of Algorithm 7.2 we need a few more definitions. We define the matrix $\tilde{V}_{\text{defl}}^{(\mu)} \in \mathbb{C}^{n\times\mu}$ via its columns

$$(\tilde{V}_{\text{defl}}^{(\mu)})_{:,i} = \begin{cases} \boldsymbol{\upsilon}^{(i)} & \text{if } i \in I^{(\mu)} \\ 0 & \text{else} \end{cases} \tag{7.16}$$

where the set $I^{(\mu)}$ contains the indices of the inexactly deflated vectors up to step μ, i.e.

$$I^{(\mu)} := \{i : i \in I \wedge i \leq \mu\}.$$

In words, the matrix $\tilde{V}_{\text{defl}}^{(\mu)}$ collects all the inexactly deflated candidate Lanczos vectors up to step μ and in column $i \in I^{(\mu)}$ we find the candidate Lanczos vector $\boldsymbol{\upsilon}^{(i)}$.

The orthogonalisation coefficients $t_{j,\mu}$ that the algorithm generates in line 10 as well as the norms in line 16 are collected in the matrix $\tilde{T}^{(c,\mu)}$ of recurrence coefficients with

$$(\tilde{T}^{(c^{(\mu)},\mu)})_{p,q} = \begin{cases} t_{p,q} & \text{if } p \in J^{(q)} \\ t_{p,q} & \text{if } h^{(p)} = q \text{ and } q \notin I^{(\mu)} \\ 0 & \text{else.} \end{cases} \tag{7.17}$$

Note that $\tilde{T}^{(c,\mu)}$ does not contain the values $t_{p,q}$ with $q \leq 0$.

Using the matrices $V^{(c^{(\mu)})}$ from (7.15), $\tilde{V}_{\text{defl}}^{(\mu)}$ from (7.16), and $\tilde{T}^{(c^{(\mu)},\mu)}$ from (7.17) we can state a deflated block Lanczos-type relation as

$$AV^{(\mu)} = V^{(c^{(\mu)})}\tilde{T}^{(c^{(\mu)},\mu)} + \tilde{V}_{\text{defl}}^{(\mu)}. \tag{7.18}$$

Relation (7.18), however, is not the final relation that we want to use as a foundation for later developed deflated block Krylov subspace methods, because the matrix $\tilde{T}^{(\mu)} = (\tilde{T}^{(c,\mu)})_{1:\mu,1:\mu}$ lacks two useful properties. First, the matrix $\tilde{T}^{(\mu)}$ not necessarily has a small bandwidth. In the possible case of inexact deflation in step $\mu = 1$ the complete first row consists of non-zeros. Second, $\tilde{T}^{(\mu)}$ is not Hermitian. Orthogonalisation coefficients belonging to inexactly deflated vectors only appear above the diagonal of $\tilde{T}^{(\mu)}$.

Like for the two-sided algorithm we can split the matrix

$$\tilde{T}^{(c^{(\mu)},\mu)} = T^{(c^{(\mu)},\mu)} + S^{(c^{(\mu)},\mu)}$$

where

$$(T^{(c^{(\mu)},\mu)})_{p,q} = \begin{cases} t_{p,q} & \text{if } p \in \{j \in J^{(q)} : j \geq h^{(q)}\} \\ t_{p,q} & \text{if } h^{(p)} = q \text{ and } q \notin I^{(\mu)} \\ 0 & \text{else} \end{cases}$$

and

$$(S^{(c^{(\mu)},\mu)})_{p,q} = \begin{cases} t_{p,q} & \text{if } p \in \{j \in J^{(q)} : j < h^{(q)}\} \\ 0 & \text{else.} \end{cases}$$

In words, the matrix $S^{(c^{(\mu)},\mu)}$ contains those entries of $\tilde{T}^{(c^{(\mu)},\mu)}$ that arise from orthogonalisations against deflated Lanczos vectors that are not in the range of the according history index anymore. Now, we can transform (7.18) to

$$\begin{aligned} AV^{(\mu)} &= V^{(c^{(\mu)})}\tilde{T}^{(c^{(\mu)},\mu)} + \tilde{V}_{\text{defl}}^{(\mu)} \\ &= V^{(c^{(\mu)})}(T^{(c^{(\mu)},\mu)} + S^{(c^{(\mu)},\mu)}) + \tilde{V}_{\text{defl}}^{(\mu)} \\ &= V^{(c^{(\mu)})}T^{(c^{(\mu)},\mu)} + \underbrace{V^{(c^{(\mu)})}S^{(c^{(\mu)},\mu)} + \tilde{V}_{\text{defl}}^{(\mu)}}_{=:V_{\text{defl}}^{(\mu)}} \end{aligned}$$

which yields the deflated block Lanczos-type relation

$$AV^{(\mu)} = V^{(c^{(\mu)})}T^{(c^{(\mu)},\mu)} + V_{\text{defl}}^{(\mu)}. \tag{7.19}$$

The following proposition specialises Proposition 7.4 to the Hermitian case.

Proposition 7.5. *Let all the matrices and indices be defined as before. Then the norms of the matrices* $\tilde{V}_{\text{defl}}^{(\mu)}$ *and* $V_{\text{defl}}^{(\mu)}$ *from* (7.18) *and* (7.19)*, respectively, collecting orthogonalisation coefficients related to deflated candidate Lanczos vectors can be bound by*

$$\left\|\tilde{V}_{\text{defl}}^{(\mu)}\right\|_F \leq \text{tol}_{\text{defl}}\sqrt{d^{(\mu)}} \tag{7.20}$$

$$\left\|V_{\text{defl}}^{(\mu)}\right\|_F \leq \text{tol}_{\text{defl}}\sqrt{d^{(\mu)}\mu} \tag{7.21}$$

Proof. We note that $\tilde{V}_{\text{defl}}^{(\mu)}$ consists of at most $d^{(\mu)}$ non-zero columns having each norm less or equal to tol_{defl}. This shows that (7.20) holds. The matrix $S^{(c^{(\mu)},\mu)}$ contains the orthogonalisation coefficients from orthogonalising against inexactly deflated vectors. Therefore, at most $d^{(\mu)}$ rows in $S^{(c^{(\mu)},\mu)}$ contain non-zero entries of magnitude not larger than tol_{defl}. Moreover, the non-zeros in $S_v^{(c^{(\mu)},\mu)}$ are all located above the diagonal which makes them at most $\mu - 1$ per row as a rough bound. Using this we can show that the inequality (7.20) holds since

$$\begin{aligned}
\left\|V_{\text{defl}}^{(\mu)}\right\|_F &= \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{\mu}\left|(V^{(c^{(\mu)})}S^{(c^{(\mu)},\mu)} + \tilde{V}_{\text{defl}}^{(\mu)})_{i,j}\right|^2} \\
&\leq \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{\mu}\left|(V^{(c^{(\mu)})}S^{(c^{(\mu)},\mu)})_{i,j}\right|^2 + d^{(\mu)}\text{tol}_{\text{defl}}^2} \\
&= \sqrt{\sum_{i\in I^{(\mu)}}\sum_{j=1}^{\mu}\|v^{(i)}\|^2\left|(S^{(c^{(\mu)},\mu)})_{i,j}\right|^2 + d^{(\mu)}\text{tol}_{\text{defl}}^2} \\
&\leq \sqrt{d^{(\mu)}(\mu-1)\text{tol}_{\text{defl}}^2 + d^{(\mu)}\text{tol}_{\text{defl}}^2} \\
&= \text{tol}_{\text{defl}}\sqrt{d^{(\mu)}\mu}.
\end{aligned}$$

□

As compared to Proposition 7.4 the bound for $\left\|V_{\text{defl}}^{(\mu)}\right\|_F$ in (7.21) can be much better than (7.14) for $\left\|V_{\text{defl}}^{(\mu_v)}\right\|_F$, because $(v^{(i)})^H v^{(i)} = 1$. The bound in (7.21) only depends on the deflation tolerance tol_{defl}, the number of deflated vectors $d^{(\mu)}$ and the number of steps μ of the deflated block Lanczos-type process.

A sharper bound could be formulated if the steps in which inexact deflation occurs were taken into account. But for a moderate number of steps $\left\|V_{\text{defl}}^{(\mu)}\right\|_F$ is quite small. The deflated block Lanczos-type relation (7.19) can be used to obtain

$$\begin{aligned}(V^{(\mu)})^H A V^{(\mu)} &= T^{(\mu)} + (V^{(\mu)})^H V_{\text{defl}}^{(\mu)} \\ &= T^{(\mu)} + S^{(\mu)} + (V^{(\mu)})^H \tilde{V}_{\text{defl}}^{(\mu)} \\ &= T^{(\mu)} + \underbrace{S^{(\mu)} + (S^{(\mu)})^H}_{=:\tilde{S}^{(\mu)}}\end{aligned} \tag{7.22}$$

and we see that $T^{(\mu)}$ is Hermitian. Like before we can bound the norm of $\tilde{S}^{(\mu)}$ in the following proposition.

Proposition 7.6. *The inequality*

$$\left\|\tilde{S}^{(\mu)}\right\|_F \leq \text{tol}_{\text{defl}} \sqrt{2\mu d^{(\mu)}} \tag{7.23}$$

holds using the definitions from before.

Proof. We have

$$\begin{aligned}\left\|\tilde{S}^{(\mu)}\right\|_F = \left\|S^{(\mu)} + (S^{(\mu)})^H\right\|_F &= \sqrt{\sum_{i=1}^{\mu}\sum_{j=1}^{\mu} \left|(S^{(\mu)} + (S^{(\mu)})^H)_{i,j}\right|^2} \\ &\leq \sqrt{2\mu d^{(\mu)} \text{tol}_{\text{defl}}^2} = \text{tol}_{\text{defl}} \sqrt{2\mu d^{(\mu)}}.\end{aligned}$$

The inequality holds since $S^{(\mu)}$ consists of at most $d^{(\mu)}$ rows with entries of magnitude not larger than tol_{defl}. All remaining entries are zero. □

Implementation details. Before proceeding to an example a couple of remarks on the deflated block Lanczos-type process and Algorithm 7.2 are in order:

- In line 10 the scalar $t_{j,\mu}$ has to be computed only for $j \geq \mu$ and for the orthogonalisation against inexactly deflated vectors. The entries above the diagonal of $T^{(\mu)}$ are already known due to symmetry as indicated by (7.22).
- The check for $\|v\| = 0$ in line 12 should be implemented by comparing to some small threshold.
- The algorithm can be implemented efficiently with storing not more than $2m + 1$ vectors at a time. When deflation occurs, the number of stored vectors decreases by one since after some steps we need to orthogonalise against 2 less last vectors but need to continue to orthogonalise against the deflated one.

An example. We finish the discussion of the deflated block Lanczos-type process with an example that is depicted in Figure 7.4. We start with 4 starting vectors $b_1, \dots, b_4$ and inexact deflation occurs twice—once while creating the 8-th Lanczos vector in step $\mu = 4$ and the second time in step $\mu = 9$. The table in Figure 7.4a lists the indices, history indices and sets throughout every cycle through the first $\mu = 10$ steps of Algorithm 7.2 for our example. Until step $\mu = 10$ there are $c^{(\mu)} = 12$ Lanczos vectors created including the orthogonalised 4 starting vectors. The graphical representation in Figure 7.4b shows the sparsity pattern of the matrices $\tilde{T}^{(c^{(\mu)},\mu)}$ and $\tilde{T}^{(\mu)}$. The sparsity pattern of $T^{(\mu)}$ is represented by the entries marked with filled circles ● only. The reduced block size $m^{(\nu)}$ can be found in $\tilde{T}^{(c^{(\mu)},\mu)}$ as the number of non-zeros above and left of entry (ν, ν) and the history index $h^{(\nu)}$ is the row-number of the first non-zero entry in column ν of the matrix $\tilde{T}^{(c^{(\mu)},\mu)}$ whilst ignoring entries of small magnitude (○ and ◌) in both cases.

Figure 7.4b finally closes the gap to our initial definition of the deflated block Krylov subspaces in Definition 7.2 and the deflated block Krylov matrix from (7.3). The rectangles grouping some of the entries in Figure 7.4b display what would be computed in one step if the algorithm would proceed block-wise instead of using Ruhe's approach of generating a single vector per step. After $\nu = \kappa(k, j)$ steps of Algorithm 7.2 we have spanned the deflated block Krylov subspace $\mathcal{K}^{\text{defl}}_{\kappa(k,j)}(A, B)$. The reduced block size $m^{(\nu)}$ in step ν corresponds to the number of columns of the matrix $B^{(k)}$ in (7.3). In our example $B^{(0)}$ has four columns, $B^{(1)}$ and $B^{(2)}$ have three, and the block-vectors starting from $B^{(3)}$ have two columns.

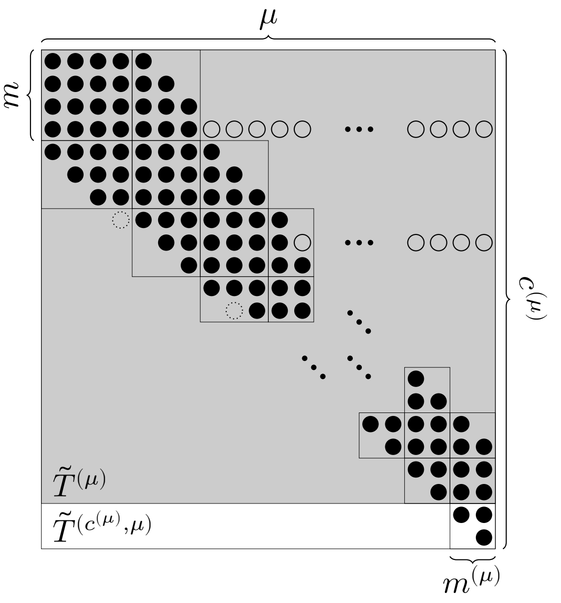
7.1.4 Block-Featured Deflated Lanczos-Type Process for Hermitian Matrices

The two-sided deflated block Lanczos-type process and the deflated block Lanczos-type process presented before whilst implementing proper deflation pursue Ruhe's approach of expanding the deflated block Krylov subspaces by one vector at a time. But, working on block-vectors instead of single vectors has the potential to speed up computations in an actual implementation. This is caused by making use of BLAS level 3 functions which apply techniques like blocking resulting in a higher performance.

In this section we develop a novel process as contribution to deflated block Lanczos-type processes which we call the *block-featured deflated Lanczos-type process*. We do so by adding proper deflation to the concept of expanding the deflated block Krylov subspace block-wise.

μ	$c^{(\mu)}$	$h^{(c^{(\mu)})}$	$h^{(\mu)}$	$I^{(\mu)}$
1	5	1	1	$\emptyset$
2	6	2	1	$\emptyset$
3	7	3	1	$\emptyset$
4	8	–	1	$\{4\}$
5	8	5	1	$\{4\}$
6	9	6	2	$\{4\}$
7	10	7	3	$\{4\}$
8	11	8	5	$\{4\}$
9	12	–	6	$\{4, 9\}$
10	12	10	7	$\{4, 9\}$

(a) The relation of the iteration counters μ and $c^{(\mu)}$, the history indices $h^{(c^{(\mu)})}$ and $h^{(\mu)}$ as well as the set of inexactly deflated candidate vector indices $I^{(\mu)}$ for positive μ.



(b) Sparsity patterns of the two matrices $\tilde{T}^{(c^{(\mu)},\mu)}$ and $\tilde{T}^{(\mu)}$ that are defined in (7.17). The filled circles ● represent non-zero entries. Empty circles ○ represent entries of small magnitude arising due to orthogonalisation against inexactly deflated vectors. Empty dotted circles ◌ represent computed values of small magnitude being the norm of inexactly deflated vectors and are not actually included in the matrices.

Figure 7.4: Example demonstrating the progression of Algorithm 7.2 in case of deflation in steps $\mu = 4$ and $\mu = 9$ for 4 starting vectors.

Our starting point is the block Lanczos process from Algorithm 6.6. The process performs a QR decomposition of the newly created block-vector. Its purpose is to ensure that the block-vector consists of m orthonormal Lanczos vectors. This, however, works only as long as no (nearly) linearly dependent vectors arise in the process. Here, we enhance the block Lanczos process to mimic the behaviour of the deflated block Lanczos-type process in that it removes columns of the generated block-vectors that are (nearly) linearly dependent and maintains orthogonality of all the generated Lanczos vectors.

The lack of handling (nearly) linearly dependent vectors with the QR decomposition is what we want to tackle for two reasons. First, we always have a block size of m. If we were able to decrease the block size in case of (near) linear dependency we could generate a deflated block Krylov subspace of the same dimension with less computational work. Second, using a QR decomposition can result in a rank-deficient or ill-conditioned factor R. In Algorithm 6.6 the R-factor enters the matrix $\boldsymbol{T}^{(c+1,c)}$ as $T_{c+1,c}$ and might lead to numerical instabilities when building a block Krylov subspace method on top of the block Lanczos process.

As in the previous sections we first give an overview of the block-featured deflated Lanczos-type process, then we state the algorithm and lastly discuss some details of the process.

Overview of the process. Like before, we want to construct an orthonormal basis of the deflated block Krylov subspace. This time, however, we want to progress block-wise which means that we extend a basis of $\mathscr{K}^{\text{defl}}_{k-1,0}(A, B)$ to $\mathscr{K}^{\text{defl}}_{k,0}(A, B)$ in step k. The deflated basis is kept in the columns of a matrix $\boldsymbol{V}^{(\kappa(k,0))}$. To include deflation we have to check all of the up to m Lanczos vectors created in the current step k for (near) linear dependency. A QR decomposition could be applied to spot (near) linear dependency in the block-vector $V^{(k)} = Q^{(k)}R^{(k)}$ by monitoring the magnitude of the diagonal entries in the matrix $R^{(k)}$. This would be the exact analogue of the approach in the two-sided deflated block Lanczos-type process and the deflated block Lanczos-type process. However, in the case of (nearly) linearly dependent columns in $V^{(k)}$ it might not be possible to select a strict subset of the columns of $Q^{(k)}$ whose span (approximately) equals range$(V^{(k)})$ [GL96, Section 5.4]. Hence, an alternative to the QR decomposition has to be found.

In our algorithm we replace the QR decomposition by a *rank-revealing QR decomposition.* The term rank-revealing QR decomposition was coined in [Cha87]. The basic idea is to compute a factorisation

$$\mathcal{V}^{(k)}\Pi^{(k)} = \tilde{V}^{(k)}R^{(k)}$$

for the column-permuted matrix $\mathcal{V}^{(k)} \in \mathbb{C}^{n\times m^{(k-1)}}$. Here, we used the reduced block size $m^{(k-1)}$ in step $k-1$ which together with the number of deflated vectors $d^{(k-1)}$ until step $k-1$ satisfies $m = m^{(k-1)} + d^{(k-1)}$. The matrix $\Pi^{(k)} \in \mathbb{R}^{m^{(k-1)}\times m^{(k-1)}}$ represents a permutation, $\tilde{V}^{(k)} \in \mathbb{C}^{n\times m^{(k-1)}}$ has orthonormal columns and $R^{(k)} \in \mathbb{C}^{m^{(k-1)}\times m^{(k-1)}}$ is upper triangular. The notation $\mathcal{V}$ emphasises that we still might remove columns from the *candidate Lanczos block-vector* $\mathcal{V}$ to obtain the next block-vector of Lanczos vectors. Now, the rank-revealing enters the stage by choosing the permutation $\Pi^{(k)}$ in such a way that if $R^{(k)}$ is partitioned as

$$R^{(k)} = \begin{bmatrix} R^{(k)}_{11} & R^{(k)}_{12} \\ 0 & R^{(k)}_{22} \end{bmatrix}$$

then the upper triangular block $R_{11}^{(k)} \in \mathbb{C}^{m^{(k)} \times m^{(k)}}$ is non-singular and the norm $\|R_{22}^{(k)}\|_2$ of $R_{22}^{(k)} \in \mathbb{C}^{(m^{(k-1)}-m^{(k)}) \times (m^{(k-1)}-m^{(k)})}$ is small. One can show that $\sigma_{m^{(k-1)}+1}(\mathcal{V}^{(k)}) \leq \|R_{22}^{(k)}\|_2$ [Cha87] where σ_i denotes the i-th singular value of $\mathcal{V}^{(k)}$ with $\sigma_1 \geq \ldots \geq \sigma_{m^{(k-1)}}$. This implies that if $\|R_{22}^{(k)}\|_2$ is small then $\mathcal{V}^{(k)}$ is nearly rank deficient and we could deflate $m^{(k-1)} - m^{(k)}$ vectors. If we remove these vectors from $\tilde{V}^{(k)}$ we end up with $V^{(k)} \in \mathbb{C}^{n \times m^{(k)}}$ containing linearly independent columns.

Different rank-revealing QR decompositions differ in how they compute the permutation $\Pi^{(k)}$. The most prominent rank-revealing QR decomposition is the column pivoting Householder QR method from [BG65]. It is presented in [GL96, Algorithm 5.4.1] and we have described its additions to the Householder QR decomposition in Section 1.3. Its key idea is to build the factorisation successively and choose the permutation such that out of all remaining vectors always the vector with largest norm after already orthogonalising against the already computed columns of Q is selected for being orthogonalised next. This implies that the diagonal entries of the R-factor are decreasing in magnitude and that entries right of the diagonal are smaller in magnitude than the corresponding diagonal entry. However, it does not always guarantee a small $\|R_{22}^{(k)}\|_2$ for (nearly) rank deficient matrices [Cha87]. Then again, in practise it is rarely seen that the method fails to identify (near) rank deficiency. In [Cha87] a rank-revealing QR decomposition method RRQR is presented that is more costly than the column pivoting Householder-QR method but guarantees a small norm of the $R_{22}^{(k)}$-block for rank-1 deficient matrices. Some improvements on the rank-revealing QR decomposition from [Cha87] can be found in [CH94; GE96; HP92].

Besides the rank-revealing QR decomposition which allows us to decide if and which columns of the block-vector can be deflated we have to keep the deflated Lanczos vectors like in the two-sided deflated block Lanczos-type process and the deflated block Lanczos-type process. That way we can ensure that $\boldsymbol{V}^{(\kappa(k,0))}$ still consists of orthonormal columns even after deflation occurred.

The algorithm. Algorithm 7.3 displays the block-featured deflated Lanczos-type process for Hermitian matrices that incorporates the rank-revealing QR decomposition described before. Up to line 6 of the algorithm the initialisation is completed and the starting block-vector B is orthogonalised yielding $V^{(1)}$. Some columns might have been deflated, thus the reduced block size $m^{(1)}$ might be smaller than m. From line 8 to line 11 the orthogonalisation against the last two blocks and the previously deflated vectors takes place. The rank-revealing QR decompositions in line 2 and line 12 can be any rank-revealing

Algorithm 7.3: Block-featured deflated Lanczos-type process

Input :	$A \in \mathbb{C}^{n\times n}$	system matrix, Hermitian
	$B \in \mathbb{C}^{n\times m}$	starting block-vector
	$\mathrm{tol}_{\mathrm{defl}}$	deflation tolerance
	k	number of steps to perform
Output:	$\boldsymbol{V}^{(k+1)} = [V^{(1)}\vert\cdots\vert V^{(k+1)}]$	the columns of $\boldsymbol{V}^{(k+1)}$ form an orthonormal basis of $\mathscr{K}^{\mathrm{defl}}_{k+1,0}(A,B)$
	$\boldsymbol{T}^{(k+1,k)}$	matrix containing orthogonalisation coefficient blocks such that (7.24) holds

```
1  $V^{(0)} = 0$
2  $\mathfrak{V}^{(1)}\mathfrak{T}^{(1,0)} = B\Pi^{(1)}$        // rank-revealing QR decomposition of $B$
3  check $\mathfrak{T}^{(1,0)}$ for deflation yielding $m^{(1)}$
4  $V^{(1)} = \mathfrak{V}^{(1)}_{:,1:m^{(1)}}$
5  $V^{(1)}_{\mathrm{keep}} = (B\Pi^{(1)})_{:,m^{(1)}+1:m}$        // keep deflated vectors
6  $T_{1,0} = (\mathfrak{T}^{(1,0)}(\Pi^{(1)})^{-1})_{1:m^{(1)},:}$
7  for $c = 1,2,\dots,k$ do
8      $\tilde{V} = AV^{(c)} - V^{(c-1)}T_{c,c-1}$
9      $T_{c,c} = (V^{(c)})^H\tilde{V}$
10     $\tilde{V} = \tilde{V} - V^{(c)}T_{c,c}$
11     $\tilde{V} = \tilde{V} - V^{(c)}_{\mathrm{keep}}((V^{(c)}_{\mathrm{keep}})^H\tilde{V})$        // orthogonalisation against previously deflated vectors
12     $\mathfrak{V}^{(c+1)}\mathfrak{T}^{(c+1,c)} = \tilde{V}\Pi^{(c+1)}$        // rank-revealing QR decomposition of $\tilde{V}$
13     check $\mathfrak{T}^{(c+1,c)}$ for deflation yielding $m^{(c+1)}$
14     $V^{(c+1)} = \mathfrak{V}^{(c+1)}_{:,1:m^{(c+1)}}$
15     $V^{(c+1)}_{\mathrm{keep}} = [V^{(c)}_{\mathrm{keep}} \,\vert\, (V^{(c)}\Pi^{(c+1)})_{:,m^{(c+1)}+1:m^{(c)}}]$        // keep deflated vectors
16     $T_{c+1,c} = (\mathfrak{T}^{(c+1,c)}(\Pi^{(c+1)})^{-1})_{1:m^{(c+1)},:}$
17     $T_{c,c+1} = (T_{c+1,c})^H$
```

QR decomposition. In our implementation we use the column-pivoting Householder QR decomposition from [BG65] since it is cheaper to compute than the other mentioned rank-revealing QR decompositions. The check for need of deflation in the lines 3 and 13 is done by searching for the smallest ν such that the diagonal entry $(\mathfrak{T}^{(c+1,c)})_{\nu,\nu}$ is smaller than $\mathrm{tol}_{\mathrm{defl}}$ in magnitude. We then set $m^{(c+1)} = \nu$.

The Lanczos vectors that the deflated vectors originate from are kept in the matrix $V^{(c+1)}_{\mathrm{keep}}$ in the case of inexact deflation. If $(\mathfrak{T}^{(c+1,c)})_{\mu,\mu} \approx 0$ for $\mu > \nu$ then exact deflation is applied and there is no need to keep the corresponding Lanczos vector. The inverse of the permutation that is computed during the rank-revealing QR decomposition is applied to the matrix $\mathfrak{T}^{(c+1,c)}$ in the lines 6 and 16 and the last $m^{(c+1)}$ rows thereof form $T_{c+1,c}$.

We note that the block-featured deflated Lanczos-type process not necessarily spans the same deflated block Krylov subspace as the deflated block Lanczos-type process since the involved rank-revealing QR decompositions can lead to a different selection of deflated vectors. In actual computations we can see that deflation occurs in different steps of both processes which can be seen in Figure 8.24 of Section 8.4.

A deflated block Lanczos-type relation. The matrices

$$\boldsymbol{V}^{(k+1)} = [V^{(1)}| \cdots |V^{(k+1)}]$$

and

$$\boldsymbol{T}^{(k+1,k)} = \begin{bmatrix} T_{1,1} & T_{1,2} & & \\ T_{2,1} & T_{2,2} & \ddots & \\ & \ddots & \ddots & T_{k-1,k} \\ & & T_{k,k-1} & T_{k,k} \\ & & & T_{k+1,k} \end{bmatrix}$$

are generated by the block-featured deflated Lanczos-type process in Algorithm 7.3. Using these we can state a deflated block Lanczos-type relation

$$A\boldsymbol{V}^{(k)} = \boldsymbol{V}^{(k+1)}\boldsymbol{T}^{(k+1,k)} + \boldsymbol{V}_{\text{defl}}^{(k)} \tag{7.24}$$

that the matrices $\boldsymbol{V}^{(k+1)}$ and $\boldsymbol{T}^{(k+1,k)}$ fulfil. The matrix $\boldsymbol{V}_{\text{defl}}^{(k)}$ amounts to the deflated candidate Lanczos vectors $\mathcal{V}^{(c+1)}_{:,(m^{(c+1)}+1):m^{(c)}}$ in the columns of $\mathcal{V}^{(c+1)}$ and the orthogonalisation against vectors in $V_{\text{keep}}^{(c)}$. We described how the matrix $\boldsymbol{V}_{\text{defl}}^{(k)}$ is composed in detail in Section 7.1.3 and refer to the description of the deflated block Lanczos-type process for all the details thereon.

Analogously to Proposition 7.5 and Proposition 7.6 that hold for the deflated block Lanczos-type process we can derive similar bounds in the following proposition. But first, we note that equation (7.22) translates to

$$(\boldsymbol{V}^{(k)})^H A\boldsymbol{V}^{(k)} = \boldsymbol{T}^{(k)} + (\boldsymbol{V}^{(k)})^H \boldsymbol{V}_{\text{defl}}^{(k)}. \tag{7.25}$$

Proposition 7.7. *Using the previous notation the following bounds hold*

$$\begin{aligned} \left\|\boldsymbol{V}_{\text{defl}}^{(k)}\right\|_F &\leq \text{tol}_{\text{defl}}\sqrt{d^{(k)}\kappa(k,0)} \quad \textit{and} \\ \left\|(\boldsymbol{V}^{(k)})^H\boldsymbol{V}_{\text{defl}}^{(k)}\right\|_F &\leq \text{tol}_{\text{defl}}\sqrt{2mkd^{(k)}}. \end{aligned} \tag{7.26}$$

Proof. See Proposition 7.5 and Proposition 7.6. □

An example. We want to clarify the sparsity pattern of the matrix $\boldsymbol{T}^{(k+1,k)}$ using an example. In Figure 7.5 we display a similar example as for the deflated block Lanczos-type process. Initially, we have 4 starting vectors $b_1, \dots, b_4$ and inexact deflation occurs twice—once while creating the $V^{(2)}$ and the second time while creating $V^{(4)}$. The graphical representation shows the sparsity pattern of the matrices $\boldsymbol{T}^{(k+1,k)}$ and $\boldsymbol{T}^{(k)}$ denoted by filled circles ●. The reduced block size $m^{(c+1)}$ can be found as the number of rows in the block $T_{c+1,c}$. The dotted circles ◌ indicate entries of small magnitude in the matrix $(\boldsymbol{V}^{(k)})^H \boldsymbol{V}^{(k)}_{\text{defl}}$ that arise due to orthogonalisation against deflated Lanczos vectors but are not included in $\boldsymbol{T}^{(k+1,k)}$. The most prominent difference to the structure of $T^{(\mu)}$ in Figure 7.4b is that the off-diagonal blocks are no necessarily triangular. This, however, is not required for Algorithm 7.3 since we only need the block-tridiagonal structure of $\boldsymbol{T}^{(k+1,k)}$.

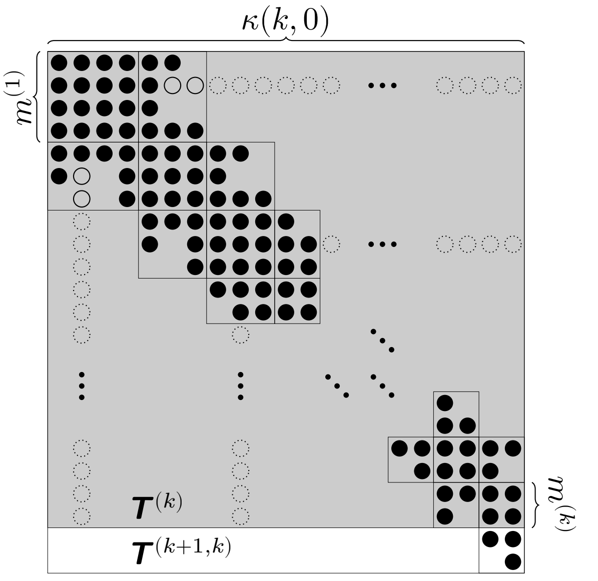


Figure 7.5: Example showing the sparsity pattern of matrix $\boldsymbol{T}^{(k+1,k)}$ generated by Algorithm 7.3 in case of deflation in steps $c = 1$ and $c = 3$ for 4 starting vectors. The filled circles ● represent non-zero entries. Empty circles ○ represent entries of small magnitude arising due to orthogonalisation against inexactly deflated vectors that are actually kept in the blocks of $\boldsymbol{T}^{(k+1,k)}$. Empty dotted circles ◌ represent values of small magnitude that are not actually included in $\boldsymbol{T}^{(k+1,k)}$.

7.2 DSBlockCG

In this section we want to develop a deflated shifted block Krylov subspace method for solving (1.1) for a Hermitian positive definite matrix A which we call *Deflated Shifted Block CG (DSBlockCG)*. It is based on shifted CG and the deflated block Lanczos-type process from Algorithm 7.2 introduced in Section 7.1.3. As it advances the basis of the current deflated block Krylov subspace by one vector at every iteration step, the method we present here will generate iterates for every shift after every newly computed Lanczos vector. Hence, it generates iterates of the kind (7.5).

We split the derivation in two parts. First, we consider the unshifted block system $AX = B$ and derive a block CG method in a similar manner as CG can be derived from the Lanczos process. We used this kind of derivation already for the shifted CG method in Section 5.1. As we base the method on the deflated block Lanczos-type process we have to figure out how deflation influences the computation of iterates. Second, we will discuss how the method then can be extended to handle multiple shifts.

7.2.1 A Deflated Block CG Method

In the following we will use the indexing using μ that we introduced in Section 7.1.3 and used especially for Algorithm 7.2. This emphasises that our index counts single vector-wise and not block-wise. Our goal will be to generate iterates $X^{(\mu)}$ for $\mu > 0$ fulfilling the Galerkin condition

$$\begin{aligned} &X^{(\mu)} \text{ with } x_j^{(\mu)} \in x_j^{(0)} + \mathcal{K}_\mu^{\text{defl}}(A, R^{(0)}) \text{ and} \\ &R^{(\mu)} \text{ with } r_j^{(\mu)} = r_j^{(0)} - Ax_j^{(\mu)} \perp \mathcal{K}_\mu^{\text{defl}}(A, R^{(0)}). \end{aligned}$$

Building upon this and using the orthogonal basis $V^{(\mu)}$ and the Hermitian matrix $T^{(\mu)}$ created by the deflated block Lanczos-type process we want to obtain iterates $X^{(\mu)}$ as

$$\begin{aligned} X^{(\mu)} &:= X^{(0)} + V^{(\mu)}(T^{(\mu)})^{-1}(V^{(\mu)})^H R^{(0)} \\ &\approx X^{(0)} + V^{(\mu)}((V^{(\mu)})^H A V^{(\mu)})^{-1}(V^{(\mu)})^H R^{(0)}. \end{aligned} \tag{7.27}$$

We assume here that according to (7.22) and (7.23) the entries of $\tilde{S}^{(\mu)}$ belonging to $(V^{(\mu)})^H A V^{(\mu)} = T^{(\mu)} + \tilde{S}^{(\mu)}$ that arise due to orthogonalisation against deflated vectors have no substantial influence and using $T^{(\mu)}$ instead is justified. Therefore, with deflation the constructed method will not have a finite termination property even in exact arithmetic, and we must expect the attainable accuracy of the iterates not to exceed the deflation tolerance tol_{defl}.

According to Algorithm 7.2 the index μ is increased every time a new candidate Lanczos vector has been created. This implies that the matrix $\widetilde{T}^{(c^{(\mu)},\mu)}$ grows a column and if no deflation occurs it also grows a row. This means that $T^{(\mu)}$ always grows a column and row. Hence, we can generate a new iterate as in (7.27) even though, occasionally, the candidate vector gets deflated and no new Lanczos vector is obtained. In this sense, deflation has no effect on the generation of the iterates.

What we still need to obtain is an update formulation for the iterates which uses short recurrences. The band structure of $T^{(\mu)}$ can be exploited for this. The following derivation is in most parts analogous to the derivation of shifted CG in Section 5.1. Let

$$L^{(\mu)}D^{(\mu)}(L^{(\mu)})^H := T^{(\mu)}$$

be the root-free Cholesky decomposition of $T^{(\mu)}$ where $L^{(\mu)}$ is a lower triangular matrix of unit diagonal with the same sparsity pattern as the lower triangular part of $T^{(\mu)}$ and $D^{(\mu)}$ is a diagonal matrix $D^{(\mu)} = \operatorname{diag}\left(d^{(1)}, d^{(2)}, \dots, d^{(\mu)}\right)$.

The only difference between $T^{(\mu)}$ and $T^{(\mu-1)}$ is the newly appended last column and row. So, writing

$$T^{(\mu)} = \begin{bmatrix} T^{(\mu-1)} & (t^{(\mu)})^H \\ t^{(\mu)} & t^{(\mu,\mu)} \end{bmatrix}, L^{(\mu)} = \begin{bmatrix} L^{(\mu-1)} & 0 \\ \ell^{(\mu)} & 1 \end{bmatrix} \text{ and } D^{(\mu)} = \begin{bmatrix} D^{(\mu-1)} & \\ & d^{(\mu)} \end{bmatrix}$$

yields

$$\begin{aligned} T^{(\mu)} &= \begin{bmatrix} T^{(\mu-1)} & (t^{(\mu)})^H \\ t^{(\mu)} & \alpha^{(\mu)} \end{bmatrix} \\ &= L^{(\mu)}D^{(\mu)}(L^{(\mu)})^H \\ &= \begin{bmatrix} L^{(\mu-1)} & \\ \ell^{(\mu)} & 1 \end{bmatrix} \begin{bmatrix} D^{(\mu-1)} & \\ & d^{(\mu)} \end{bmatrix} \begin{bmatrix} (L^{(\mu-1)})^H & (\ell^{(\mu)})^H \\ & 1 \end{bmatrix} \\ &= \begin{bmatrix} L^{(\mu-1)}D^{(\mu-1)}(L^{(\mu-1)})^H & L^{(\mu-1)}D^{(\mu-1)}(\ell^{(\mu)})^H \\ \ell^{(\mu)}D^{(\mu-1)}(L^{(\mu-1)})^H & \ell^{(\mu)}D^{(\mu-1)}(\ell^{(\mu)})^H + d^{(\mu)} \end{bmatrix}. \end{aligned}$$

We obtain updates for $L^{(\mu-1)}$ to $L^{(\mu)}$ and $D^{(\mu-1)}$ to $D^{(\mu)}$ by computing

$$\begin{aligned} \ell^{(\mu)} &= t^{(\mu)}(L^{(\mu-1)})^{-H}(D^{(\mu-1)})^{-1} \quad \text{and} \\ d^{(\mu)} &= t^{(\mu,\mu)} - \ell^{(\mu)}D^{(\mu-1)}(\ell^{(\mu)})^H. \end{aligned} \tag{7.28}$$

Note that in our notation $\ell^{(\mu)}$ and $t^{(\mu)}$ are row vectors and that they are of length $\mu - 1$. The sparsity pattern of $T^{(\mu)}$ implies that at most the last m

entries in each of the vectors $t^{(\mu)}$ as well as $\ell^{(\mu)}$ are non-zero. More precisely, $t^{(\mu)}_{h^{(\mu)}}, \dots, t^{(\mu)}_{\mu-1}$ are the only non-zeros in $t^{(\mu)}$.

We can use the updates for $L^{(\mu)}$ and $D^{(\mu)}$ in (7.28) to formulate updates for the iterates in (7.27) from $X^{(\mu-1)}$ to $X^{(\mu)}$. Using (7.27) we get

$$\begin{aligned}
X^{(\mu)} &= X^{(0)} + V^{(\mu)}(L^{(\mu)}D^{(\mu)}(L^{(\mu)})^H)^{-1}(V^{(\mu)})^H R^{(0)} \\
&= X^{(0)} + \begin{bmatrix} V^{(\mu-1)} & v^{(\mu)} \end{bmatrix} (L^{(\mu)})^{-H}(D^{(\mu)})^{-1}(L^{(\mu)})^{-1} \begin{bmatrix} V^{(\mu-1)} & v^{(\mu)} \end{bmatrix}^H R^{(0)} \\
&= X^{(0)} + \begin{bmatrix} P^{(\mu-1)} & p^{(\mu)} \end{bmatrix} (D^{(\mu)})^{-1} \begin{bmatrix} U^{(\mu-1)} \\ u^{(\mu)} \end{bmatrix} \\
&= X^{(\mu-1)} + \frac{1}{d^{(\mu)}} p^{(\mu)} u^{(\mu)}. \qquad (7.29)
\end{aligned}$$

Here we used

$$P^{(\mu)} := \begin{bmatrix} P^{(\mu-1)} & p^{(\mu)} \end{bmatrix} = \begin{bmatrix} V^{(\mu-1)} & v^{(\mu)} \end{bmatrix} \begin{bmatrix} L^{(\mu-1)} & 0 \\ \ell^{(\mu)} & 1 \end{bmatrix}^{-H}$$

and

$$U^{(\mu)} := \begin{bmatrix} U^{(\mu-1)} \\ u^{(\mu)} \end{bmatrix} = \begin{bmatrix} L^{(\mu-1)} & 0 \\ \ell^{(\mu)} & 1 \end{bmatrix}^{-1} \begin{bmatrix} (V^{(\mu-1)})^H \\ (v^{(\mu)})^H \end{bmatrix} R^{(0)}.$$

The advantage of introducing the matrices $P^{(\mu)}$ and $U^{(\mu)}$ is that for the cost of having to store them we have the benefit of being able to do the inversion of $L^{(\mu)}$ implicitly and have simple updates for

$$\begin{aligned}
p^{(\mu)} &= v^{(\mu)} - P^{(\mu-1)}(\ell^{(\mu)})^H \text{ and} \\
u^{(\mu)} &= (v^{(\mu)})^H R^{(0)} - \ell^{(\mu)} U^{(\mu-1)}.
\end{aligned} \qquad (7.30)$$

One must expect that the vectors $v^{(\mu)}$ tend to lose orthogonality rather quickly in numerical computation. However, $(v^{(\mu)})^H R^{(0)}$ in (7.30) should always be taken as zero for $\mu > m$ even though it is not in actual computation since it stabilises the computation of the new iterates. As described before, $L^{(\mu)}$ inherits the banded structure of $T^{(\mu)}$. This implies that the updates in (7.30) need at most the last m columns of $P^{(\mu-1)}$ and, accordingly, the last m rows of $U^{(\mu-1)}$.

7.2.2 Stopping Criterion for the Deflated Block CG Method

In order to be able to stop the iteration we should be able to compute the norms of the residuals $r_j^{(\mu)}, 1 \leq j \leq m$. Fortunately, although the residuals are not directly available, their norms can be computed at very low additional cost. The iterates $X^{(\mu)}$ are generated in such a way that the residuals $R^{(\mu)}$ are orthogonal to the Lanczos vectors $v^{(j)}$ for $j \leq \mu$. Therefore, they may be written as

$$R^{(\mu)} = W^{(\mu)} S^{(\mu)}$$

where $S^{(\mu)} = [s_1^{(\mu)}|\cdots|s_m^{(\mu)}] \in \mathbb{C}^{m^{(\mu)} \times m}$ and $W^{(\mu)} \in \mathbb{C}^{n \times m^{(\mu)}}$ consists of the last $m^{(\mu)}$ columns of $V^{(c^{(\mu)})}$, i.e. $W^{(\mu)} = [v^{(\mu+1)}|\dots|v^{(c^{(\mu)})}]$. For $\nu > m$ this can be seen by using relation (7.19) which yields

$$\begin{aligned}
(v^{(\nu)})^H R^{(\mu)} &= (v^{(\nu)})^H (B - AX^{(\mu)}) \\
&= \underbrace{(v^{(\nu)})^H (R^{(0)}}_{=0} - AV^{(\mu)} (T^{(\mu)})^{-1} (V^{(\mu)})^H R^{(0)}) \\
&= (v^{(\nu)})^H V^{(c^{(\mu)})} T^{(c^{(\mu)},\mu)} (T^{(\mu)})^{-1} (V^{(\mu)})^H R^{(0)}.
\end{aligned}$$

Hence, the residuals $R^{(\mu)}$ only depend on the last few Lanczos vectors. Moreover, by exploiting the property that $V^{(\mu)}$ and thus $W^{(\mu)}$ has orthonormal columns, we get

$$\left\| r_j^{(\mu)} \right\| = \left\| W^{(\mu)} s_j^{(\mu)} \right\| = \left\| s_j^{(\mu)} \right\|.$$

Therefore, if we are just interested in the norm of each of the residuals we only have to compute the norm of the columns of $S^{(\mu)} \in \mathbb{C}^{m^{(\mu)} \times m}$. Computing $S^{(\mu)}$ is actually quite simple since

$$\begin{aligned}
S^{(\mu)} &= (W^{(\mu)})^H R^{(\mu)} \\
&= (W^{(\mu)})^H (B - AX^{(\mu)}) \\
&= (W^{(\mu)})^H (R^{(0)} - AP^{(\mu)} (D^{(\mu)})^{-1} U^{(\mu)}) \\
&= (W^{(\mu)})^H (R^{(0)} - AV^{(\mu)} (L^{(\mu)})^{-H} (D^{(\mu)})^{-1} U^{(\mu)}) \\
&= \underbrace{(W^{(\mu)})^H (R^{(0)}}_{=0 \text{ for } \mu > m} - V^{(c^{(\mu)})} T^{(c^{(\mu)},\mu)} \underbrace{(L^{(\mu)})^{-H} (D^{(\mu)})^{-1} U^{(\mu)}}_{=:\tilde{S}^{(\mu)}}) \\
&= - \begin{bmatrix} 0 & I_{m^{(\mu)}} \end{bmatrix} T^{(c^{(\mu)},\mu)} \tilde{S}^{(\mu)}. \qquad (7.31)
\end{aligned}$$

The computation of $S^{(\mu)}$ thus essentially only involves one matrix-matrix product of small matrices of dimension $m^{(\mu)}$ and one inversion of the small triangular matrix $(L^{(\mu)})^H$. So obtaining $S^{(\mu)}$ is computationally cheap.

7.2.3 The Deflated Shifted Block CG Method

The block method developed so far can be extended to handle shifted systems and multiple right-hand sides at the same time. The only restriction is on the choice of the starting vectors, because we need to have collinear initial residuals for all shifts of a given right-hand side as for shifted CG in Section 5.1. This constraint can obviously be fulfilled by choosing 0 as the starting vector for all shifts. For ease of notation we focus on a situation where we have just one additional shift σ and use the notation $A_\sigma := A + \sigma I$. Matrices and vectors belonging to the shifted system will also be denoted by the index σ.

Krylov subspaces are shift-invariant as discussed in Chapter 5. Moreover, starting with the initial vector b, the Lanczos process produces exactly the same vectors, whether we take A or A_σ for computing a basis of $\mathcal{K}_k(A, b) = \mathcal{K}_k(A_\sigma, b)$ [PPV95]. This property immediately carries over to deflated block Krylov subspaces and to the block Lanczos-type process as well as the deflated block Lanczos-type process. Moreover, the deflated block Lanczos-type relation (7.19) turns into

$$A_\sigma V^{(\mu)} = V^{(c^{(\mu)})} T_\sigma^{(c^{(\mu)},\mu)} + V_{\text{defl}}^{(\mu)}$$

with

$$T_\sigma^{(c^{(\mu)},\mu)} = T^{(c^{(\mu)},\mu)} + \sigma I.$$

In particular, we have

$$(V^{(\mu)})^H (A + \sigma I) V^{(\mu)} = T^{(\mu)} + \sigma I =: T_\sigma^{(\mu)},$$

so that we can obtain the iterates for the shifted system in exactly the same manner as described in Section 7.2.1, with $T^{(\mu)}$ replaced by $T_\sigma^{(\mu)}$. This saves us the cost of creating $V^{(\mu)}$ and $T^{(\mu)}$ for each shifted system individually which is expected to be the most expensive part since it involves multiplications with the matrix A_σ. However, for every shift, the Cholesky factorisation $L_\sigma^{(\mu)}$ and $D_\sigma^{(\mu)}$ and the matrices $P_\sigma^{(\mu)}$ and $U_\sigma^{(\mu)}$ needed for generating the iterates, have to be computed and stored. We will give a detailed analysis of the memory requirements in the next section.

7.2.4 The DSBlockCG Algorithm

Summarising the previous sections and using Algorithm 7.2 we now formulate the DSBlockCG algorithm as Algorithm 7.4. We have previously published this algorithm in [BF14]. The loop in line 22 means that if for a fixed shift σ_i all the iterates $x_{i,j}$ with $1 \leq j \leq m$ are converged to the requested tolerance, the matrices $L_i^{(\mu)}, D_i^{(\mu)}, P_i^{(\mu)}, U_i^{(\mu)}$ and $S_i^{(\mu)}$ do not have to be updated any more. Additionally, only those columns of $S_i^{(\mu)}$ have to be computed that belong to

Algorithm 7.4: DSBlockCG

Input :	$A \in \mathbb{C}^{n\times n}$	system matrix
	$B = [b_1 \vert \cdots \vert b_m] \in \mathbb{C}^{n\times m}$	right-hand side block-vector
	$\sigma_1, \dots, \sigma_s$	shifts with $\sigma_i \in \mathbb{R}$, s.t. $A + \sigma_i I$ is hpd
	$\mathrm{tol}_{\mathrm{defl}}$	deflation tolerance

Output: approximate solutions $x_{i,j}^{(k)}$ to $(A + \sigma_i I)x_{i,j} = b_j$

```
1  $\mu = -m, I = \emptyset, (h^{(-m+1)}, \dots, h^{(0)}) = (1, \dots, 1)$
2  $X^{(0)} = 0, P^{(0)} = 0, U^{(0)} = 0$
3  for $c = 1, 2, \dots$ until convergence do
4      repeat                                                   // build the next Lanczos vector
5          $\mu = \mu + 1$
6          if $\mu = c$ then stop                                   // block Krylov subspace depleted
7          if $\mu \le 0$ then
8              $\boldsymbol{v}^{(\mu)} = b_{m+\mu}$
9          else
10             $\boldsymbol{v}^{(\mu)} = Av^{(\mu)}$
11         for $j \in J^{(\mu)} = \{h^{(\mu)}, \dots, c-1\} \cup I$ do             // orthogonalisation
12             $t_{j,\mu} = (v^{(j)})^H v$
13             $\boldsymbol{v}^{(\mu)} = \boldsymbol{v}^{(\mu)} - t_{j,\mu} v^{(j)}$
14         if $\|\boldsymbol{v}^{(\mu)}\| = 0$ then discard vector $\boldsymbol{v}^{(\mu)}$                // exact deflation
15         else if $\|\boldsymbol{v}^{(\mu)}\| < \mathrm{tol}_{\mathrm{defl}}$ and $\mu > 0$ then                // inexact deflation
16             keep $v^{(\mu)}$ for later orthogonalisation and set $I = I \cup \{\mu\}$
17         else                                                     // new Lanczos vector
18             $t_{c,\mu} = \|\boldsymbol{v}^{(\mu)}\|$
19             $v^{(c)} = \boldsymbol{v}^{(\mu)} / t_{c,\mu}$
20             $h^{(c)} = \max\{1, \mu\}$
21         if $\mu > 0$ then                                        // compute the iterates
22             forall the shifts $\sigma_i$ with unconverged systems do
                   // update the Cholesky decomposition, cf. (7.28)
23                 if $\mu = 1$ then $\ell_i^{(\mu)} = 0$
24                 else $\ell_i^{(\mu)} = t^{(\mu)} (L_i^{(\mu-1)})^{-H} (D_i^{(\mu-1)})^{-1}$
25                 $d_i^{(\mu)} = (t^{(\mu,\mu)} + \sigma_i) - \ell_i^{(\mu)} D_i^{(\mu-1)} (\ell_i^{(\mu)})^H$
                   // update the matrices $P$ and $U$, cf. (7.30)
26                 $p_i^{(\mu)} = v^{(\mu)} - P_i^{(\mu-1)} (\ell_i^{(\mu)})^H$
27                 $u_i^{(\mu)} = (v^{(\mu)})^H B - \ell^{(\mu)} U^{(\mu-1)}$
                   // update the non-converged columns of the iterate, cf. (7.29)
28                 $X_i^{(\mu)} = X_i^{(\mu-1)} + \frac{1}{d_i^{(\mu)}} p_i^{(\mu)} u_i^{(\mu)}$
                   // check convergence, cf. (7.31)
29                 $S_i^{(\mu)} = -\begin{bmatrix} 0 & I_{m^{(\mu)}} \end{bmatrix} T^{(c^{(\mu)},\mu)} (L^{(\mu)})^{-H} (D^{(\mu)})^{-1} U^{(\mu)}$
30     until non-deflated new vector $v^{(c)}$ computed in line 19
```

non-converged systems for shift σ_i. Note that we allow for an arbitrary number of shifts $\sigma_1, \dots, \sigma_s$.

We state two remarks before we discuss the memory requirements and computational complexity. First, in contrast to the version of the algorithm that we presented in [BF14] we have included in Algorithm 7.4 the initial orthonormalisation of the vectors b_i. This is done by starting with a negative index μ as we have described for the deflated block Lanczos-type process (Algorithm 7.2) and has the beneficial side-effect that linear dependency in the vectors b_i can be recognised and treated with deflation. Second, as discussed for equation (7.30) in an actual implementation $(v^{(\mu)})^H B$ in line 27 should not be computed for $\mu > 0$ and should be assumed as being zero instead.

The following proposition summarises the memory requirements of Algorithm 7.4.

Proposition 7.8. *The DSBlockCG algorithm requires the storage of*

$$(sm + 2m + 1)n + 4m^2 + 2sm^2 + \mathcal{O}(m) \tag{7.32}$$

floating point numbers in addition to the storage for the matrix A, the right-hand sides B and the result vectors X_i.

Proof. The memory footprint of the DSBlockCG algorithm in a memory efficient implementation consists of $2m + 1$ vectors of length n for storing the matrix V, and a temporary vector v. Additionally, the search direction matrix $P_i^{(\mu)}$ has to be stored for every shift which amounts to additional sm vectors of length n. The matrix $T^{(\mu)}$ can be stored using an economy version matrix $\widehat{T}^{(\mu)} \in \mathbb{C}^{(2m+1)\times(m+1)}$. We then only keep as much entries as needed in $\widehat{T}$. The entry $(T^{(\mu)})_{p,q}$ can be found as $(\widehat{T}^{(\mu)})_{1+\tilde{p},1+\tilde{q}}$ where $\tilde{p} = p \mod (2m+1)$ and $\tilde{q} = q \mod (m+1)$. Since for every shifted system the shift only has to be applied to the diagonal of $T^{(\mu)}$, there is no need for storing more than one copy of this matrix. Also independent from the number of shifts is the storage for $S_i^{(\mu)}$ and a temporary matrix. However, for every shift we need to store the matrices $U_i^{(\mu)}$ and $L_i^{(\mu)}$ where the diagonal of $L_i^{(\mu)}$ contains $D_i^{(\mu)}$. All these matrices have a size of at most $m \times m$. Overall we end up with the number of floating point numbers as stated in (7.32). □

For the computational complexity we assume that $n \gg m$ and $n \gg s$ so that $\mathcal{O}(m^3)$ and $\mathcal{O}(ms)$ operations are negligible compared to the vector operations of length n. For simplicity, we only analyse the non-deflation case summarised in the following result.

Proposition 7.9. *We assume that no deflation occurs and that all systems from* (7.1) *belonging to the same shift σ_i are solved after k_i steps. Let $k = \max_{i=1,\ldots,s}\{k_i\}$. The total number of operations in Algorithm 7.4 for solving* (7.1) *is*

$$o_{\text{dsblockcg}} = k(c_A + 3m + 1) + 2m \sum_{i=1}^{s} k_i. \tag{7.33}$$

Proof. The computational cost for one step in the DSBlockCG algorithm consists of

- a multiplication with the matrix A,
- the orthogonalisation against $2m$ previous vectors,
- the normalisation of the new vector that extends the Krylov subspace,
- the update of the search direction matrices $P_i^{(\mu)}$ and the iterates $X_i^{(\mu)}$ for every system belonging to a non-converged shift σ_i.

The orthogonalisation needs $2m$ vector additions but just m inner products by exploiting the symmetry of $T^{(\mu)}$. Normalising the resulting vector after orthogonalisation accounts for one vector operation. The updates of the search directions and the iterates account for $2m$ vector operations but are only performed for not yet converged systems. Overall we end up with a computational complexity as stated in (7.33). □

Concluding this section we discuss how DSBlockCG performs against shifted CG and BCGrQ depending on the number of right-hand sides and shifts as parameters using our cost model from Section 1.3. First, we compare DSBlockCG to shifted CG from Section 5.1 which is a more fine-grained version of the result in [BF14, Section 5].

Proposition 7.10. *We assume that no deflation occurs and that all systems from* (1.1) *belonging to the same shift σ_i are solved after step k_i in the DSBlockCG algorithm and after $\tilde{k}_i$ steps of shifted CG when shifted CG is applied to each right-hand side individually. Let*

$$k = \max_{i=1,\ldots,s}\{k_i\} \text{ and } \tilde{k} = \max_{i=1,\ldots,s}\left\{\tilde{k}_i\right\}.$$

Further, assume that

$$2\sum_{i=1}^{s} k_i = \tau k \text{ and } 2\sum_{i=1}^{s} \tilde{k}_i = \tau \tilde{k}$$

for the same $\tau \geq 2$. The number of vector operations in DSBlockCG is less than the number of vector operations in shifted CG, hinting at DSBlockCG *being faster than shifted CG for solving all systems in* (1.1), *if*

$$\frac{m(c_A + 4 + \tau)}{c_A + 3m + 1 + m\tau} \geq \frac{k}{\tilde{k}} \tag{7.34}$$

holds.

Proof. Comparing the number of operations of DSBlockCG from Proposition 7.9 and shifted CG from Proposition 5.1 yields

$$\begin{aligned} & & m o_{\text{scg}} &\geq o_{\text{dsblockcg}} \\ \Leftrightarrow & & m\left(\tilde{k}(c_A + 4) + 2\sum_{i=1}^{s} \tilde{k}_i\right) &\geq k(c_A + 3m + 1) + 2m\sum_{i=1}^{s} k_i \\ \Leftrightarrow & & \tilde{k} m(c_A + 4 + \tau) &\geq k(c_A + 3m + 1 + m\tau) \end{aligned}$$

□

Proposition 7.10 shows that DSBlockCG as compared to shifted CG benefits most if c_A dominates the terms in the left-hand side of (7.34). Thus, DSBlockCG can be expected to perform better if the matrix-vector multiplication with A is costly and τ is reasonably small. The latter is true for either a small number of shifts or if the shifts are distributed so that only a few of the shifted systems are hard to solve.

Estimating the number of steps of shifted CG and DSBlockCG a priori is not possible in most practical situations. But there is some useful information that we can retrieve from (7.34) instead of regarding Proposition 7.10 only as a hint or an a posteriori explanation for time measurements. We know that $k \in \{\tilde{k}, \dots, m\tilde{k}\}$ where the extreme cases are that all but one right-hand side are deflated immediately on the $\tilde{k}$ end and that there is no benefit from the block approach at all on the $m\tilde{k}$ end. If we assume that $k = \alpha m\tilde{k}$ for $\alpha \in [1/m, 1]$ then we get

$$\begin{aligned} & & \frac{m(c_A + 4 + \tau)}{c_A + 3m + 1 + m\tau} &\geq \frac{k}{\tilde{k}} \\ \Leftrightarrow & & \frac{(c_A + 4 + \tau)}{c_A + 3m + 1 + m\tau} &\geq \alpha \\ \Leftrightarrow & & c_A + 4 + \tau &\geq \alpha(c_A + 3m + 1 + m\tau) \\ \Leftrightarrow & & (1 - \alpha)c_A &\geq \alpha(3m + 1 + m\tau) - 4 - \tau. \end{aligned}$$

Hence, for $\alpha = 1$ we can expect no gain in computational time when using DSBlockCG instead of shifted CG applied to all right-hand sides separately.

But, if we save some iteration steps with DSBlockCG compared to shifted CG and c_A is sufficiently large then we can expect some savings in computational time. Especially when deflation occurs, the number of iteration steps can decrease significantly, thus leading to $\alpha \ll 1$.

In addition to the comparison of DSBlockCG and shifted CG that was already published in [BF14] we now want to compare DSBlockCG to BCGrQ which is applied to every shift separately. We make the simplifying assumption that all shifts are equally hard to solve, i.e. DSBlockCG computes iterates for every system until the last step resulting in

$$o_{\text{dsblockcg}} = k(c_A + 3m + 1) + 2m \sum_{i=1}^{s} k_i = k(c_A + 3m + 1 + 2sm).$$

Proposition 7.11. *Assume that BCGrQ applied to any of the shifts in* (1.1) *separately solves* $(A + \sigma_i I)X = B$ *in* $\tilde{k}$ *steps and let* mk *be the number of steps that DSBlockCG needs to solve* (1.1). *The number of vector operations in DSBlockCG is less than the number of vector operations in BCGrQ, hinting at DSBlockCG being able to outperform BCGrQ for solving all systems in* (1.1), *if*

$$\frac{s(c_A^{\square}(m) + 6m + 4)}{c_A + 3m + 1 + 2sm} \geq \frac{k}{\tilde{k}}$$

holds.

Proof. Comparing the number of operations of DSBlockCG from Proposition 7.9 and BCGrQ from Proposition 6.17 yields

$$\begin{aligned} & s o_{\text{bcgrq}} \geq o_{\text{dsblockcg}} \\ \Leftrightarrow \qquad & s\tilde{k}(mc_A^{\square}(m) + 6m^2 + 4m) \geq k(c_A + 3m + 1 + 2sm) \end{aligned}$$ □

The iteration numbers k and $\tilde{k}$ in Proposition 7.11 can be expected to be almost the same. Hence, only for one shift or if the number of shifts is small and $c_A^{\square}(m) \ll c_A$ then BCGrQ might be faster than DSBlockCG. This is in line with the comparison of shifted CG and CG in Proposition 5.2 and confirms that shifted methods should be preferred since the solutions for the shifted systems essentially come for free.

7.3 BFDSCG

In the following we develop a deflated shifted block Krylov subspace methods for solving (1.1) for a Hermitian positive definite matrix A which we will call

Block-Featured Deflated Shifted CG (BFDSCG). It is closely related to the DSBlockCG method of Section 7.2. In fact, the derivation follows the same idea of merging shifted CG with a deflated block Krylov subspace but instead of the deflated block Lanczos-type process we now base the algorithm on the block-featured deflated Lanczos-type process. Hence, we do not progress in a single vector manner but block-vector-wise and generate iterates of the kind (7.4). Before we consider shifted systems, we develop a block CG method for solving the unshifted block system $AX = B$. Since the following steps of the derivation are analogous to the those in Section 7.2, we will keep the description brief in most cases to reduce redundancy.

7.3.1 A Block-Featured Deflated CG Method

Since we base the deflated shifted block Krylov subspace method of this section on the block-featured deflated Lanczos-type process of Algorithm 7.3, we will adopt the indexing using the index k from there. This emphasises that we progress in a block-wise manner. Our goal will be to generate iterates $X^{(k)}$ fulfilling the Galerkin condition

$$\begin{aligned} X^{(k)} &\text{ with } x_j^{(k)} \in x_j^{(0)} + \mathcal{K}_{k,0}^{\text{defl}}(A, R^{(0)}) \text{ and} \\ R^{(k)} &\text{ with } r_j^{(k)} = r_j^{(0)} - Ax_j^{(k)} \perp \mathcal{K}_{k,0}^{\text{defl}}(A, R^{(0)}). \end{aligned}$$

In particular, we want to use the orthogonal basis $\boldsymbol{V}^{(k)}$ and the Hermitian matrix $\boldsymbol{T}^{(k)}$ created by the block-featured deflated Lanczos-type process to define iterates $X^{(k)}$ as

$$\begin{aligned} X^{(k)} &:= X^{(0)} + \boldsymbol{V}^{(k)}(\boldsymbol{T}^{(k)})^{-1}(\boldsymbol{V}^{(k)})^H R^{(0)} \\ &\approx X^{(0)} + \boldsymbol{V}^{(k)}((\boldsymbol{V}^{(k)})^H A \boldsymbol{V}^{(k)})^{-1}(\boldsymbol{V}^{(k)})^H R^{(0)}. \end{aligned} \tag{7.35}$$

We assume here that according to (7.25) and (7.26) the entries of $(\boldsymbol{V}^{(k)})^H A \boldsymbol{V}^{(k)}$ that arise due to orthogonalisation against deflated vectors and are not included in $\boldsymbol{T}^{(k)}$ have no substantial influence. In other words, we assume that

$$A\boldsymbol{V}^{(k)} \approx \boldsymbol{V}^{(k+1)}\boldsymbol{T}^{(k+1,k)}. \tag{7.36}$$

In step k of Algorithm 7.3 a block $V^{(k+1)}$ of $m^{(k+1)}$ Lanczos vectors and candidate Lanczos vectors has been created. The matrix $\boldsymbol{T}^{(k+1,k)}$ as compared to $\boldsymbol{T}^{(k,k-1)}$ is extended by $m^{(k)}$ columns and $m^{(k+1)}$ rows or more specifically by three blocks $T_{k-1,k} \in \mathbb{C}^{m^{(k-1)} \times m^{(k)}}$, $T_{k,k} \in \mathbb{C}^{m^{(k)} \times m^{(k)}}$, and $T_{k+1,k} \in \mathbb{C}^{m^{(k+1)} \times m^{(k)}}$. This implies that $\boldsymbol{T}^{(k)}$ grows by $m^{(k)}$ columns and rows consisting of the blocks

$T_{k,k-1}$, $T_{k-1,k} = (T_{k,k-1})^H$, and $T_{k,k}$ as compared to $\boldsymbol{T}^{(k-1)}$. So, deflation happening in step k results in $m^{(k+1)} < m^{(k)}$ and thereby influences the computation of iterates (7.35) in steps after k. We come back to this in Section 7.3.3.

In order to achieve short recurrences we need to obtain an update formulation of the iterates in (7.35) from step $k-1$ to step k. For this we exploit the block tri-diagonal structure of $\boldsymbol{T}^{(k)}$ and the way $\boldsymbol{T}^{(k-1)}$ is updated to $\boldsymbol{T}^{(k)}$. Let

$$\boldsymbol{L}^{(k)}\boldsymbol{D}^{(k)}(\boldsymbol{L}^{(k)})^H := \boldsymbol{T}^{(k)}$$

be the root-free Cholesky decomposition of $\boldsymbol{T}^{(k)}$. The matrix $\boldsymbol{L}^{(k)}$ is a lower block-triangular matrix with lower triangular diagonal blocks that have a unit diagonal and $\boldsymbol{L}^{(k)}$ has the same block structure as the lower block-triangular part of $\boldsymbol{T}^{(k)}$. The matrix $\boldsymbol{D}^{(k)} = \operatorname{diag}\left(D^{(1)}, D^{(2)}, \dots, D^{(k)}\right)$ is a diagonal matrix where all the blocks are diagonal matrices themselves.

Now, we have all the notation at hand to retrieve an update for the root-free Cholesky decomposition of $\boldsymbol{T}^{(k)}$. We do so by writing

$$\boldsymbol{T}^{(k)} = \begin{bmatrix} \boldsymbol{T}^{(k-1)} & (T^{(k)})^H \\ T^{(k)} & T^{(k,k)} \end{bmatrix}, \boldsymbol{L}^{(k)} = \begin{bmatrix} \boldsymbol{L}^{(k-1)} & 0 \\ L^{(k)} & L^{(k,k)} \end{bmatrix}, \text{ and } \boldsymbol{D}^{(k)} = \begin{bmatrix} \boldsymbol{D}^{(k-1)} & \\ & D^{(k)} \end{bmatrix}.$$

The matrix $T^{(k)} \in \mathbb{C}^{m^{(k)} \times \kappa(k,0)}$ consists of zeros except for the rightmost $m^{(k-1)}$ columns which contain the matrix $T_{k,k-1}$ of the block-featured deflated Lanczos-type process. Moreover, we have $T^{(k,k)} = T_{k,k}$. The matrix $L^{(k)}$ has the same block structure as $T^{(k)}$ in that all but the last $m^{(k-1)}$ columns are zeros and $D^{(k)}$ is a diagonal matrix. This yields

$$\begin{aligned}\boldsymbol{T}^{(k)} &= \begin{bmatrix} \boldsymbol{L}^{(k-1)} & \\ L^{(k)} & L^{(k,k)} \end{bmatrix} \begin{bmatrix} \boldsymbol{D}^{(k-1)} & \\ & D^{(k)} \end{bmatrix} \begin{bmatrix} (\boldsymbol{L}^{(k-1)})^H & (L^{(k)})^H \\ & (L^{(k,k)})^H \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{L}^{(k-1)}\boldsymbol{D}^{(k-1)}(\boldsymbol{L}^{(k-1)})^H & \boldsymbol{L}^{(k-1)}\boldsymbol{D}^{(k-1)}(L^{(k)})^H \\ L^{(k)}\boldsymbol{D}^{(k-1)}(\boldsymbol{L}^{(k-1)})^H & L^{(k)}\boldsymbol{D}^{(k-1)}(L^{(k)})^H + L^{(k,k)}D^{(k)}(L^{(k,k)})^H \end{bmatrix},\end{aligned}$$

from which we obtain updates for $\boldsymbol{L}^{(k-1)}$ to $\boldsymbol{L}^{(k)}$ by computing

$$L^{(k)} = T^{(k)}(\boldsymbol{L}^{(k-1)})^{-H}(\boldsymbol{D}^{(k-1)})^{-1}. \tag{7.37}$$

From the structure of $T^{(k)}$ we can deduce that $L^{(k)}$ only has non-zeros in the last $m^{(k-1)}$ columns and we combine these to a block $L^{(k,k-1)}$. Then the update from (7.37) reads

$$L^{(k,k-1)} = T^{(k,k-1)}(L^{(k-1,k-1)})^{-H}(D^{(k-1)})^{-1}. \tag{7.38}$$

A notable difference to DSBlockCG is that we not only need to compute an update for the new diagonal entry $D^{(k)}$ but also for the new block-diagonal entry $L^{(k,k)}$ of $\boldsymbol{L}^{(k)}$. This is done via a root-free Cholesky decomposition of the $m^{(k)} \times m^{(k)}$-matrix $T^{(k,k)} - L^{(k)}\boldsymbol{D}^{(k-1)}(L^{(k)})^H$ yielding

$$\begin{aligned} L^{(k,k)}D^{(k)}(L^{(k,k)})^H &:= T^{(k,k)} - L^{(k)}\boldsymbol{D}^{(k-1)}(L^{(k)})^H \\ &= T^{(k,k)} - L^{(k,k-1)}D^{(k-1)}(L^{(k,k-1)})^H. \end{aligned} \tag{7.39}$$

We can use the updates in (7.37), (7.38), and (7.39) for the blocks of the Cholesky decomposition to formulate updates for the iterates in (7.35) from $X^{(k-1)}$ to $X^{(k)}$. Using (7.35) we get

$$\begin{aligned} X^{(k)} &= X^{(0)} + \boldsymbol{V}^{(k)}(\boldsymbol{L}^{(k)}\boldsymbol{D}^{(k)}(\boldsymbol{L}^{(k)})^H)^{-1}(\boldsymbol{V}^{(k)})^H R^{(0)} \\ &= X^{(0)} + \begin{bmatrix} \boldsymbol{V}^{(k-1)} & V^{(k)} \end{bmatrix} (\boldsymbol{L}^{(k)})^{-H}(\boldsymbol{D}^{(k)})^{-1}(\boldsymbol{L}^{(k)})^{-1} \begin{bmatrix} \boldsymbol{V}^{(k-1)} & V^{(k)} \end{bmatrix}^H R^{(0)} \\ &= X^{(0)} + \begin{bmatrix} \boldsymbol{P}^{(k-1)} & P^{(k)} \end{bmatrix} (\boldsymbol{D}^{(k)})^{-1} \begin{bmatrix} \boldsymbol{U}^{(k-1)} \\ U^{(k)} \end{bmatrix} \\ &= X^{(k-1)} + P^{(k)}(D^{(k)})^{-1}U^{(k)}. \end{aligned} \tag{7.40}$$

Here we used

$$\boldsymbol{P}^{(k)} := \begin{bmatrix} \boldsymbol{P}^{(k-1)} & P^{(k)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{V}^{(k-1)} & V^{(k)} \end{bmatrix} \begin{bmatrix} \boldsymbol{L}^{(k-1)} & 0 \\ L^{(k)} & L^{(k,k)} \end{bmatrix}^{-H}$$

and

$$\boldsymbol{U}^{(k)} := \begin{bmatrix} \boldsymbol{U}^{(k-1)} \\ U^{(k)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{L}^{(k-1)} & 0 \\ L^{(k)} & L^{(k,k)} \end{bmatrix}^{-1} \begin{bmatrix} (\boldsymbol{V}^{(k-1)})^H \\ (V^{(k)})^H \end{bmatrix} R^{(0)}.$$

Using the auxiliary matrices $\boldsymbol{P}^{(k)}$ and $\boldsymbol{U}^{(k)}$ allows us to obtain a simple update for $X^{(k)}$ based on the updates

$$\begin{aligned} P^{(k)} &= \left(V^{(k)} - \boldsymbol{P}^{(k-1)}(L^{(k)})^H\right)(L^{(k,k)})^{-H} \\ &= \left(V^{(k)} - P^{(k-1)}(L^{(k,k-1)})^H\right)(L^{(k,k)})^{-H} \text{ and} \\ U^{(k)} &= (L^{(k,k)})^{-1}\left((V^{(k)})^H R^{(0)} - L^{(k)}\boldsymbol{U}^{(k-1)}\right) \\ &= (L^{(k,k)})^{-1}\left((V^{(k)})^H R^{(0)} - L^{(k,k-1)}U^{(k-1)}\right). \end{aligned} \tag{7.41}$$

Although the block-vectors $V^{(k)}$ tend to loose orthogonality rather quickly in numerical computation, the block inner product $(V^{(k)})^H R^{(0)}$ in (7.41) should be taken as zero for $k > 1$ even though it is not in actual computation since it stabilises the computation of the new iterates.

7.3.2 Stopping Criterion

As for the deflated block CG method we are interested in being able to compute the norms of the residuals $r_j^{(k)}, 1 \leq j \leq m$ for stopping the iteration. Since we know that the residual $R^{(k)}$ is orthogonal to $\boldsymbol{V}^{(k)}$ it can be written as

$$R^{(k)} = V^{(k+1)} S^{(k)}$$

where $S^{(k)} = [s_1^{(k)} | \cdots | s_m^{(k)}] \in \mathbb{C}^{m^{(k+1)} \times m}$ and can be computed as

$$\begin{aligned}
S^{(k)} &= (V^{(k+1)})^H R^{(k)} \\
&= (V^{(k+1)})^H (B - AX^{(k)}) \\
&= (V^{(k+1)})^H (R^{(0)} - A\boldsymbol{V}^{(k)} (\boldsymbol{L}^{(k)})^{-H} (\boldsymbol{D}^{(k)})^{-1} \boldsymbol{U}^{(k)}) \\
&= \underbrace{(V^{(k+1)})^H (R^{(0)}}_{=0} - \boldsymbol{V}^{(k+1)} \boldsymbol{T}^{(k+1,k)} (\boldsymbol{L}^{(k)})^{-H} (\boldsymbol{D}^{(k)})^{-1} \boldsymbol{U}^{(k)}) \\
&= -\begin{bmatrix} 0 & I_{m^{(k+1)}} \end{bmatrix} \boldsymbol{T}^{(k+1,k)} (\boldsymbol{L}^{(k)})^{-H} (\boldsymbol{D}^{(k)})^{-1} \boldsymbol{U}^{(k)} \\
&= -T^{(k+1,k)} (L^{(k,k-1)})^{-H} (D^{(k)})^{-1} U^{(k)} \qquad (7.42)
\end{aligned}$$

using (7.36). Computing $S^{(k)}$ essentially involves only one matrix-matrix product of matrices of dimension $m^{(k+1)}$ and one inversion of the small triangular matrix $(L^{(k)})^H$ which makes obtaining $S^{(k)}$ computationally cheap. For monitoring the norm of individual columns of $R^{(k)}$ it is sufficient to compute $S^{(k)}$ since we can exploit that $V^{(k+1)}$ has orthonormal columns yielding

$$\left\| r_j^{(k)} \right\| = \left\| V^{(k+1)} s_j^{(k)} \right\| = \left\| s_j^{(k)} \right\|.$$

This makes for a feasible stopping criterion.

7.3.3 Deflation in the Block-Featured Deflated CG Method

Until now we have not explicitly taken into account what happens when deflation occurs. As a result of deflation the block-featured deflated Lanczos-type process that we base our method on removes columns in the Lanczos block-vectors $V^{(k)}$. Moreover, the matrix $T^{(k,k-1)} \in \mathbb{C}^{m^{(k)} \times m^{(k-1)}}$ can be rectangular with a smaller number of rows than columns and $T^{(k,k)}$ might have less rows and columns than $T^{(k-1,k-1)}$.

A close look on the derivation in Section 7.3.1 and 7.3.2 reveals that all the relations we formulated stay true when deflation occurs. In fact, the computation of $L^{(k,k-1)}$ for the update of the root-free Cholesky decomposition of $\boldsymbol{T}^{(k)}$ in (7.38) yields an $m^{(k)} \times m^{(k-1)}$ matrix. The decomposition in (7.39) reduces the size from $m^{(k-1)}$ to $m^{(k)}$, too. Finally, in (7.41) the rectangular matrix $L^{(k,k-1)}$ reduces the $m^{(k-1)}$ columns of $P^{(k-1)}$ to $m^{(k)}$ columns in $P^{(k-1)}(L^{(k,k-1)})^H$. The number of rows in $U^{(k)}$ is affected in the same way. An actual implementation needs to be able to adjust the sizes of the involved matrices.

All in all, the relations defining the updates of the iterates are robust and capable of handling a reduction in the block size. Furthermore, our method updates the iterates for all systems, even those belonging to deflated right-hand sides, since $U^{(k)}$ retains m columns. Thus, we do not have to treat deflated systems specially and retrieve their solutions when the non-deflated systems have converged like it is the case in some other methods.

As for the update of the iterates, the computation of $S^{(k)}$ only involves matrices of matching dimensions. Hence, equation (7.42) also yields a feasible stopping criterion in case of deflation.

7.3.4 The Block-Featured Deflated Shifted CG Method

Now we want to extend the method for being able to handle multiple shifts. Once again, we have to add a constraint on the choice of the starting block-vector which have to yield collinear initial residuals for all shifts of a given right-hand side. We describe the idea for one additional shift σ and use the notation $A_\sigma := A + \sigma I$. The final algorithm will be stated for an arbitrary number of shifts.

Because of the shift invariance of deflated block Krylov subspaces the block-featured deflated Lanczos-type process generates the same Lanczos vectors if it is applied to A or A_σ and the same stating block-vector. Then the relation (7.24) yields

$$A_\sigma \boldsymbol{V}^{(k)} = \boldsymbol{V}^{(k+1)} \boldsymbol{T}_\sigma^{(k+1,k)} + \boldsymbol{V}_{\text{defl}}^{(k+1)}$$

with

$$\boldsymbol{T}_\sigma^{(k+1,k)} = \boldsymbol{T}^{(k+1,k)} + \sigma I.$$

Hence, we have

$$(\boldsymbol{V}^{(k)})^H (A + \sigma I) \boldsymbol{V}^{(k)} = \boldsymbol{T}^{(k)} + \sigma I =: \boldsymbol{T}_\sigma^{(k)},$$

and can obtain the iterates for the shifted system in the same manner as for the unshifted system except for $\boldsymbol{T}^{(k)}$ being replaced by $\boldsymbol{T}_\sigma^{(k)}$. The expensive part of creating $\boldsymbol{V}^{(k)}$ and $\boldsymbol{T}^{(k)}$ can be done just once and is then recycled for all shifted systems. It remains to compute and store the Cholesky factorisation $\boldsymbol{L}_\sigma^{(k)}$ and $\boldsymbol{D}_\sigma^{(k)}$ as well as the matrices $\boldsymbol{P}_\sigma^{(k)}$ and $\boldsymbol{U}_\sigma^{(k)}$ for all shifts individually.

7.3.5 The BFDSCG Algorithm

Now we can put together all parts and state the BFDSCG algorithm in Algorithm 7.5.

Before discussing the memory requirements and computational complexity we want to elaborate on computing iterates if a subset of the right-hand sides has converged. In line 28 we independently check the convergence for all the right-hand sides belonging to the system with shift σ_i via the norm of the columns of $S_i^{(c)}$. In principle, we could stop updating column j of the iterate X_i and skip the computation of the corresponding column of $P_i^{(c)}$ whilst keeping its original size if system j has converged. Furthermore, we would even be able to delete this column from $P_i^{(c)}$. Skipping columns in X_i or $P_i^{(c)}$ leads to introducing additional logic in line 26 or line 27 preventing us from using block-vector operations. If we choose to remove column j from $P_i^{(c)}$ then subsequently the number of columns in $V^{(c+1)}$ and $P_i^{(c)}$ differs and we cannot user block-vector operations there. Thus, we found that it is best to just keep computing iterates even for the converged systems until all the systems belonging to one shift have converged.

In the following proposition we summarise the memory requirements of Algorithm 7.5.

Proposition 7.12. *The BFDSCG algorithm requires the storage of*

$$(s+3)mn + 4sm^2 + sm + \mathcal{O}(m) \tag{7.43}$$

floating point numbers in addition to the storage for the matrix A, the right-hand side block-vector B, and the result block-vectors X_i.

Proof. Algorithm 7.5 needs to store $3+s$ block-vectors—2 of them containing Lanczos vectors, one temporary block-vector, and one search direction block-vector per shift. Additional memory for V_{keep} is actually not needed since every time a vector is deflated and appended to the matrix V_{keep} the block size of the block-vector for storing the Lanczos vectors can be reduced. Besides this, we need to store 4 matrices of size $m \times m$ for $U^{(c)}$, $L^{(c,c)}$, $L^{(c+1,c)}$, and $S^{(c)}$ for every shift. What remains is the storage for s diagonal matrices $D^{(c)}$ accounting for sm floating point numbers and some $\mathcal{O}(m)$ storage including, for instance, the permutations $\Pi^{(c+1)}$. Overall we end up with the number of floating point numbers as stated in (7.43). □

As for DSBlockCG we only analyse the non-deflation case and assume that $n \gg m$ and $n \gg s$ so that $\mathcal{O}(m^3)$ and $\mathcal{O}(ms)$ operations are negligible compared to the vector operations.

Algorithm 7.5: BFDSCG

```
Input  :  $A \in \mathbb{C}^{n\times n}$                      system matrix
          $B = [b_1|\cdots|b_m] \in \mathbb{C}^{n\times m}$    right-hand side block-vector
          $\sigma_1, \dots, \sigma_s$                          shifts with $\sigma_i \in \mathbb{R}$, s.t. $A + \sigma_i I$ is hpd
          $\mathrm{tol}_{\mathrm{defl}}$                       deflation tolerance
Output:   approximate solutions $x_{i,j}^{(k)}$ to $(A + \sigma_i I)x_{i,j} = b_j$
1  $V^{(0)} = 0, X^{(0)} = 0, P^{(0)} = 0, U^{(0)} = 0$
2  $\mathfrak{U}^{(1)}\mathfrak{T}^{(1,0)} = B\Pi^{(1)}$        // rank-revealing QR decomposition of $B$
3  check $\mathfrak{T}^{(1,0)}$ for deflation yielding $m^{(1)}$
4  $V^{(1)} = \mathfrak{U}^{(1)}_{:,1:m^{(1)}}$
5  $V^{(1)}_{\mathrm{keep}} = (B\Pi^{(1)})_{:,m^{(1)}+1:m}$        // keep deflated vectors
6  $T_{1,0} = (\mathfrak{T}^{(1,0)}(\Pi^{(1)})^{-1})_{1:m^{(1)},:}$
7  for $c = 1, 2, \dots$ until convergence do
       // block-featured deflated Lanczos-type process part
8      $\tilde{V} = AV^{(c)} - V^{(c-1)}T_{c,c-1}$
9      $T_{c,c} = (V^{(c)})^H\tilde{V}$
10     $\tilde{V} = \tilde{V} - V^{(c)}T_{c,c}$
11     $\tilde{V} = \tilde{V} - V^{(c)}_{\mathrm{keep}}((V^{(c)}_{\mathrm{keep}})^H\tilde{V})$     // orthogonalisation against previously deflated vectors
12     $\mathfrak{U}^{(c+1)}\mathfrak{T}^{(c+1,c)} = \tilde{V}\Pi^{(c+1)}$        // rank-revealing QR decomposition of $\tilde{V}$
13     check $\mathfrak{T}^{(c+1,c)}$ for deflation yielding $m^{(c+1)}$
14     $V^{(c+1)} = \mathfrak{U}^{(c+1)}_{:,1:m^{(c+1)}}$
15     $V^{(c+1)}_{\mathrm{keep}} = [V^{(c)}_{\mathrm{keep}} \,|\, (V^{(c)}\Pi^{(c+1)})_{:,m^{(c+1)}+1:m^{(c)}}]$        // keep deflated vectors
16     $T_{c+1,c} = (\mathfrak{T}^{(c+1,c)}(\Pi^{(c+1)})^{-1})_{1:m^{(c+1)},:}$
17     $T_{c,c+1} = (T_{c+1,c})^H$
       // block-featured deflated shifted CG part
18     forall the shifts $\sigma_i$ with unconverged systems do
19         if $c == 1$ then $L_i^{(1,1)}D_i^{(1)}(L_i^{(1,1)})^H = T^{(1,1)} + \sigma_i I$        // cf. (7.39)
20         else
21             $L_i^{(c,c-1)} = T_i^{(c,c-1)}(L_i^{(c-1,c-1)})^{-H}(D_i^{(c-1)})^{-1}$        // cf. (7.37)
22             $L_i^{(c,c)}D_i^{(c)}(L_i^{(c,c)})^H = T^{(c,c)} + \sigma_i I - L_i^{(c,c-1)}D_i^{(c-1)}(L_i^{(c,c-1)})^H$        // cf. (7.39)
23         if $c == 1$ then
24             $P_i^{(1)} = V^{(1)}(L_i^{(1,1)})^{-H}$
               $U_i^{(1)} = (L_i^{(1,1)})^{-1}(V^{(1)})^HB$        // cf. (7.41)
25         else
26             $P_i^{(c)} = \left(V^{(c)} - P_i^{(c-1)}(L_i^{(c,c-1)})^H\right)(L_i^{(c,c)})^{-H}$
               $U_i^{(c)} = (L_i^{(c,c)})^{-1}\left(-L_i^{(c,c-1)}U_i^{(c-1)}\right)$        // cf. (7.41)
27         $X_i^{(c)} = X_i^{(c-1)} + P_i^{(c)}(D_i^{(c)})^{-1}U_i^{(c)}$        // update iterates, cf. (7.40)
28         $S_i^{(c)} = -T_i^{(c+1,c)}(L_i^{(c,c-1)})^{-H}(D_i^{(c)})^{-1}U_i^{(c)}$        // check convergence, cf. (7.42)
```

Proposition 7.13. *We assume that no deflation occurs and that all systems from* (7.1) *belonging to the same shift σ_i are solved after k_i steps. Let $k = \max_{i=1,\ldots,s}\{k_i\}$ denote the number of iteration steps of BFDSCG applied to solve* (7.1). *The total number of operations in Algorithm 7.5 for solving* (7.1) *is*

$$o_{\text{bfdscg}} = km(c_A^{\square}(m) + 5m + 6) + 2m^2 \sum_{i=1}^{s} k_i. \tag{7.44}$$

Proof. The computational cost for one step in the BFDSCG algorithm consists of

- a matrix-block-vector product with the matrix A,
- the orthogonalisation against 2 previous block-vectors,
- the rank-revealing QR decomposition of the new block-vector whose span extends the Krylov subspace,
- the update of the search direction matrices $P_i^{(c)}$ and the iterates $X_i^{(c)}$ for every system belonging to a non-converged shift σ_i.

The orthogonalisation needs 2 block-vector additions but just 1 block-vector inner product. Normalising the resulting block-vector via a column-pivoting Householder rank-revealing QR decomposition takes the equivalent of $2m^2 + 6m$ vector operations as stated in Proposition 1.6. The updates of the search directions and the iterates account for $2m^2$ vector operations but are only performed for not yet converged systems. Overall we end up with a computational complexity as stated in (7.44). $\square$

Concluding this section the following proposition quantises how BFDSCG in Algorithm 7.5 performs against DSBlockCG from Algorithm 7.4.

Proposition 7.14. *Assume that no deflation occurs and that DSBlockCG solves the systems belonging to shift σ_i in* (7.1) *in $k_i m$ steps and BFDSCG solves the same systems in k_i steps. The number of vector operations in BFDSCG is less than the number of vector operations in DSBlockCG, hinting at BFDSCG being faster than DSBlockCG for solving all systems in* (7.1), *if*

$$2m + 5 \leq c_A - c_A^{\square}(m)$$

holds.

Proof. Let $k = \max_{i\in\{1,\ldots,s\}}\{k_i\}$. Comparing $o_{\text{dsblockcg}}$ and o_{bfdscg} under the stated assumptions yields

$$\begin{aligned} & o_{\text{bfdscg}} \le o_{\text{dsblockcg}} \\ \Leftrightarrow\quad & km(c_A^{\square}(m) + 5m + 6) + 2m^2 \sum_{i=1}^{s} k_i \le km(c_A + 3m + 1) + 2m \sum_{i=1}^{s} k_i m \\ \Leftrightarrow\quad & c_A^{\square}(m) + 5m + 6 \le c_A + 3m + 1. \end{aligned}$$

□

Proposition 7.14 shows that if the blocked nature of the performed operations leads to some improvements in computational time for the matrix-block-vector multiplications expressed by $c_A^{\square}(m) < c_A$ then BFDSCG can be faster than DSBlockCG. In Chapter 8 we will see that $c_A^{\square}(m) \ll c_A$ can result in a significant difference in computational time of both algorithms.

7.4 Shifted BCGrQ

In this section we briefly introduce the SBCGrQ algorithm from [Fut+13]. We were made aware of this method by coincidence via a talk given by Prof. Sakurai. The algorithm itself is published in volume 7851 of "Lecture Notes in Computer Science" with a title unrelated to shifted block methods which made it unlikely to be found.

We omit the details of the derivation of SBCGrQ and state it as Algorithm 7.6. The outer loop and lines 4 to 8 are equivalent to the BCGrQ algorithm in a slightly different notation. The loop starting in line 9 extends BCGrQ for solving $(A + \sigma_i I)X_i = B$. Note that only the updates of $X_i^{(c+1)}$ and $P_i^{(c+1)}$ as well as some operations on small matrices depend on the number of shifts. Most importantly, the QR decomposition is computed just once per step and not for every shift.

The memory requirements of Algorithm 7.6 are summarised in the following proposition.

Proposition 7.15. *The SBCGrQ algorithm requires the storage of*

$$(s+3)mn + (7+4s)m^2 + \mathcal{O}(m)$$

floating point numbers in addition to the storage for the matrix A, the right-hand side block-vector B, and the result block-vectors X_i.

Proof. Algorithm 7.6 needs to store $3 + s$ block-vectors—the block-vectors $Q^{(c+1)}$, $P^{(c+1)}$, a temporary block-vector, and for every shift $P_i^{(c+1)}$. The remaining storage is of order $\mathcal{O}(m^2)$ and stems from the following matrices. The

Algorithm 7.6: SBCGrQ

```
Input  :  $A \in \mathbb{C}^{n\times n}$                      system matrix
          $B = [b_1|\cdots|b_m] \in \mathbb{C}^{n\times m}$   right-hand side block-vector
          $\sigma_1, \dots, \sigma_s$                          shifts with $\sigma_i \in \mathbb{R}$, s.t. $A+\sigma_i I$ is hpd
Output:   approximate solutions $x_{i,j}^{(k)}$ to $(A+\sigma_i I)x_{i,j} = b_j$
1  $X^{(0)} = 0, \xi_i^{(-1)} = \alpha^{(-1)} = I_m$
2  $Q^{(0)}\rho^{(0)} = B$                                      // QR decomposition
3  for $c = 0,1,2,\dots$ until convergence do
4      $\alpha^{(c)} = ((P^{(c)})^H A P^{(c)})^{-1}$
5      $X^{(c+1)} = X^{(c)} + P^{(c)}\alpha^{(c)}\Delta^{(c)}$
6      $Q^{(c+1)}\rho^{(c+1)} = Q^{(c)} - AP^{(c)}\alpha^{(c)}$     // QR decomposition
7      $\Delta^{(c+1)} = \rho^{(c+1)}\Delta^{(c)}$
8      $P^{(c+1)} = Q^{(c+1)} + P^{(c)}(\rho^{(c+1)})^H$
9      forall the shifts $\sigma_i$ with unconverged systems do
10         $\tilde{\xi} = I_m + \sigma^{(i)}\alpha^{(c)} + \left[\rho^{(c)} - \xi_i^{(c)}(\xi_i^{(c-1)})^{-1}\right](\alpha^{(c-1)})^{-1}(\rho^{(c)})^H\alpha^{(c)}$
11         $\xi_i^{(c+1)} = \rho^{(c+1)}\tilde{\xi}^{-1}\xi_i^{(c)}$
12         $\alpha_i^{(c)} = \alpha^{(c)}(\rho^{(c+1)})^{-1}\xi_i^{(c+1)}$
13         $\beta_i^{(c)} = \alpha_i^{(c)}(\xi_i^{(c)})^{-1}(\alpha^{(c)})^{-1}(\rho^{(c+1)})^H$
14         $X_i^{(c+1)} = X_i^{(c)} + P_i^{(c)}\alpha_i^{(k)}$
15         $P_i^{(c+1)} = Q^{(c+1)} + P_i^{(c)}\beta_i^{(k)}$
```

matrices $\alpha^{(c)}, \alpha^{(c-1)}, \rho^{(c)}, \rho^{(c+1)}$, and $\Delta^{(c+1)}$ are independent from the number of shifts. Additionally, we need one temporary matrix and $\beta_i^{(c)}$ which can be shared for every shift. Finally, we need to store $\xi_i^{(c+1)}, \xi_i^{(c)}, \xi_i^{(c-1)}$, and $\alpha_i^{(c)}$ for every shift separately. □

For comparing the computational cost of SBCGrQ with BFDSCG we count the number of vector operations in the following proposition.

Proposition 7.16. *We assume that all systems from* (7.1) *belonging to the same shift σ_i are solved after k_i steps. Let $k = \max_{i=1,\dots,s}\{k_i\}$ denote the number of iteration steps of SBCGrQ applied to solve* (7.1)*. The total number of operations in Algorithm 7.6 for solving* (7.1) *is*

$$o_{\text{sbcgrq}} = km(c_A^{\square}(m) + 5m + 4) + 2m^2\sum_{i=1}^{s} k_i. \tag{7.45}$$

Proof. The computational cost for one step in the SBCGrQ algorithm in terms of vector operations consists of

- a matrix-block-vector product with the matrix A,
- computing $\alpha^{(c)}$,
- the QR decomposition of the residual block-vector,
- the updates of the iterate and search direction block-vectors of the unshifted system, and
- the updates of the search direction and iterate block-vectors for every system belonging to a non-converged shift.

The QR decomposition takes an equivalent of $2m^2 + 4m$ vector operations. Every other operation in the list requires m^2 vector operations. Overall we end up with a computational complexity as stated in (7.45). $\square$

Comparing the computational cost o_{sbcgrq} from (7.45) with o_{bfdscg} from (7.44) we can conclude that SBCGrQ is likely to be faster than BFDSCG. Especially, if we consider that the algorithm can be modified to solve one of the shifted systems instead of the unshifted systems as a seed system then the computational costs would even be smaller than estimated in (7.45). However, the number of operations involving small matrices of size m^2 is larger in SBCGrQ which is not represented in our cost model. Moreover, SBCGrQ cannot benefit from deflation. In such circumstances, the BFDSCG algorithm might be faster than SBCGrQ.

8 Numerical Results

In this chapter we compare the performance of the presented methods from chapters 5, 6 and 7 for different test settings. In particular, the contestant algorithms are

- shifted CG (sCG) from Section 5.1,
- restarted shifted CG (rsCG) from Section 5.2,
- the Single Seed Method (SSM) and Seed-CG from Section 6.1,
- incremental eigCG (iEigCG) from Section 6.2,
- BlockCG, BCGAdQ and BCGrQ from Section 6.3 and
- DSBlockCG as well as BFDSCG from Section 7.2 and Section 7.3.

Our main goal will be to analyse if and for which kind of application and setting our newly presented methods restarted shifted CG, DSBlockCG and BFDSCG can be used as an alternative or even an improvement to existing methods.

This chapter is organised as follows. First, we describe the test problems and some implementation details which influence the numerical tests we perform. Afterwards, some tests with the deflated block Lanczos-type process and the block-featured deflated Lanczos-type process as a basis for the algorithms of the methods from Chapter 7 are conducted. Finally, we display and discuss results for applying the implemented algorithms to particular applications.

Some of the results for the DSBlockCG algorithm were already published in [BF14]. There, however, we only compared time measurements for sparse matrices.

8.1 Implementation Details and Test Problems

Often, comparing Krylov subspace methods is done by only counting the number of matrix-vector multiplications. Especially in the case of block methods this can be quite deceptive. In every step of the iteration, besides the multiplication with a matrix for expanding the Krylov subspace, additional work has to be performed. In block Krylov subspace methods for solving m right-hand sides this usually involves $\mathcal{O}(m^2)$ vector operations. This is reflected in simply counting the iteration steps or matrix-vector multiplications, since block methods tend to reduce the number of matrix-vector multiplications whilst increasing the amount of additional work. Hence, measuring the runtime of the algorithms can be a more accurate indicator for their applicability for certain problems.

Runtime measurements, however, give rise to a set of different problems. An implementation in, for instance, Matlab grants only limited control over the amount of parallelism in each performed operation. If, for example, block-vector operations are parallelised and vector operations are not, then time measurements are biased in favour of methods that involve block-vector operations. On the other hand, if different algorithms exhibit a different amount of code optimisation then comparisons of highly optimised production-grade code can result in skewed results, too. Therefore, we chose to compare non-parallelised `C++`-implementations of the presented algorithms yielding fairer time measurements. We use the *Eigen library* [G$^+$10] in version 3.2.2 for sparse and dense BLAS operations. The tests were performed on an Intel Core i7-4770 CPU running at 3.40 GHz with 32 GiB memory and 8 MiB L3 cache. As compiler we used `clang` in version 3.3 with optimisation flag `-O3`.

Throughout this chapter x denotes a vector $x \in \mathbb{K}^n$ and we use $X \in \mathbb{K}^{n\times m}$ as a generic block-vector where in both cases n is clear by context. Here and there we will use $X_m \in \mathbb{K}^{n\times m}$ to refer to a block-vector having a particular number of columns.

For the applications that we have presented in Chapter 4 we need to solve linear systems that involve an operator A^HA. Hence, we will perform the time measurements in this section with A^HA although the methods presented in chapters 5, 6, and 7 can also be applied to A if A is hpd.

Runtime comparisons for Krylov subspace methods and block Krylov subspace methods building subspaces with respect to A^HA hinge—at least in part—on how fast A^HAx and A^HAX for a vector x and a block-vector X, respectively, can be computed. Moreover, according to our cost model from Section 1.3 we need to quantify how the time for computing A^HAx compares to a vector operation. Thus, in the following we describe some details of our implementation for those exemplary applications that we will use in the remainder of this chapter.

8.1.1 Sparse Matrices

For storing general sparse matrices we use the *compressed column storage (CCS)* format of the sparse matrix implementation of the Eigen library.

We will use 6 sparse matrices in our experiments given in Table 8.1. The

Table 8.1: Sparse test matrices used throughout this chapter. The first four of which are from the Florida sparse matrix collection [DH11] the latter two are Wilson-Dirac matrices as introduced in Chapter 4.

matrix	name	dimension	non-zeros	average non-zeros per column
A_{msc}	`msc04515`	4515	97707	21.6
A_{pois}	`Pres_Poisson`	14822	715804	48.3
A_{smt}	`smt`	25710	3749582	145.8
A_{nd12k}	`nd12k`	36000	14220946	395.0
D_{W8}	`WDSparse8`	49152	2408448	49.0
D_{W16}	`WDSparse16`	786432	38535168	49.0

matrices D_{W8} and D_{W16} are sparse matrix representations of the Wilson-Dirac operator from Section 4.1.1. These correspond to a typical configuration in lattice QCD with parameters $\beta = 5.6$ and $\kappa = 0.15825$ for both matrices taken from run A_3 in [Del$^+$07a; Del$^+$07b]. The matrix D_{W8} represents an 8^4 lattice whereas D_{W16} represents a 16^4 lattice, respectively.

As it is explained in [Fro$^+$13a, Section 2.2] the Wilson-Dirac operator can be represented in matrix terms as

$$\begin{aligned}(\tilde{D}_W\psi)(x) =& \frac{m_0+4}{a} I_{12}\psi(x)\\ &- \frac{1}{2a}\sum_{i=0}^{3}\left((I_4-\gamma_i)\otimes U_{\mu_i}^{x}\right)\psi(x+\mu_i)\\ &- \frac{1}{2a}\sum_{i=0}^{3}\left((I_4+\gamma_i)\otimes (U_{\mu_i}^{x-\mu_i})^H\right)\psi(x-\mu_i).\end{aligned} \tag{8.1}$$

This representation also clarifies the 49 non-zeros per row/column for the Wilson-Dirac operator. We have three entries from the gauge links $U_{\mu_i}^x$ for every one of the 8 space-/time-directions which doubles because of $(I_4 - \gamma_i)$ having 2 entries per row plus the diagonal entry from I_{12} which all in all sums up to 49.

Important for our experiments is how a multiplication A^HAx—computed as $A^H(Ax)$—compares to a vector operation of corresponding size vectors for A

being a matrix from Table 8.1. Additionally, we need to compare the time for computing $A^HAX = A^H(AX)$ for $X \in \mathbb{K}^{n\times m}$ with the time for computing m matrix-vector multiplications A^HAx. For this we conducted a couple of tests whose results we present in Table 8.2, Figure 8.1 and Table 8.3.

Table 8.2: Time measurements for the sparse matrices of Table 8.1. We measured the time in seconds for an axpy operation of vectors whose size corresponds to A, a matrix-vector multiplication A^HAx and matrix-block-vector multiplications A^HAX for X having $m = 1$, 10 and 100 columns. All times were averaged over 100 measurements.

matrix	axpy	A^HAx	A^HAX_1	A^HAX_{10}	A^HAX_{100}
A_{msc}	5.49×10^{-6}	2.97×10^{-4}	3.01×10^{-4}	2.82×10^{-3}	2.48×10^{-2}
A_{pois}	1.83×10^{-5}	2.35×10^{-3}	2.38×10^{-3}	2.09×10^{-2}	1.75×10^{-1}
A_{smt}	3.16×10^{-5}	1.17×10^{-2}	1.19×10^{-2}	1.08×10^{-1}	8.94×10^{-1}
A_{nd12k}	4.44×10^{-5}	4.52×10^{-2}	4.47×10^{-2}	4.15×10^{-1}	3.41×10^{0}
D_{W8}	6.35×10^{-5}	8.14×10^{-3}	8.14×10^{-3}	7.21×10^{-2}	6.14×10^{-1}
D_{W16}	1.90×10^{-3}	1.42×10^{-1}	1.44×10^{-1}	1.17×10^{0}	9.72×10^{0}

Table 8.2 shows that depending on the matrix properties the matrix-vector multiplication A^HAx can be much more expensive than an axpy operation and we elaborate on this in a moment. The second observation is that computing with block-vectors instead of vectors introduces an overhead that results in a small time penalty for a small number m of columns which becomes clearer in Figure 8.1. This can be seen by comparing the column labelled A^HAx with the one labelled A^HAX_1. This, however, is only true, if the compiler can perform optimisations by knowing the number of columns at compile-time. If we would store X_1 in an array whose number of columns is not known at compile-time we would end up with an inner loop causing time(A^HAX_1) to be significantly larger than time(A^HAx) even though the floating-point computations are essentially the same. Only with an increasing number of columns of the block-vector X the computation of A^HAX can benefit from applying the same matrix A^HA—still computed as $A^H(AX)$—to the m columns of the block-vector which can be seen in the columns labelled A^HAX_{10} and A^HAX_{100} of Table 8.2 as well as in Figure 8.1. In the latter we plotted the number m against the time measurements for computing m matrix-vector multiplications A^HAx divided by the time for computing A^HAX_m. Starting from 2 right-hand sides we switch from a vector representation of $x = X_1$ to X_m for which, for

a small number of right-hand sides, $A^H AX_m$ is slower than m matrix-vector multiplications $A^H Ax$. This explains the spike at 2 right-hand sides in Figure 8.1.

Finally, Table 8.2 and Figure 8.1 show that the matrix-block-vector multiplication at one point catches up with computing $A^H AX$ as m separate matrix-vector multiplications and can be faster for larger m. Comparing Table 8.2 with Figure 8.1 we see that the factor $\mathrm{time}(A^H Ax)m/\mathrm{time}(A^H AX)$ stays approximately the same from $m = 30$ to 100.

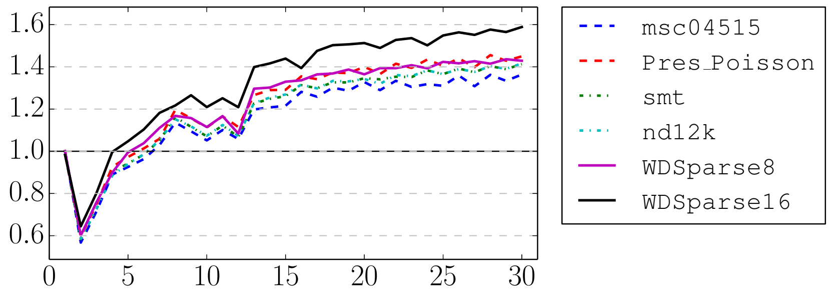


Figure 8.1: Time measurements for computing $A^H AX$ for the sparse matrices of Table 8.1 depending on the number m of columns of X. We measured the time for computing $A^H AX_m$ and plotted m against $\mathrm{time}(A^H Ax)m/\mathrm{time}(A^H AX)$. Values larger than 1 indicate that computing $A^H AX_m$ is faster than computing m matrix-vector multiplications $A^H Ax$.

Table 8.3 displays the cost of a matrix-vector multiplication $A^H Ax$ as compared to an axpy operation for the matrices introduced in Table 8.1 which is important for our cost model from Section 1.3. We can make use of this measurement for defining c_A instead of assuming that $A^H Ax$ costs the equivalent of $2nnz/n$ vector operations if $x \in \mathbb{K}^n$. Hence, in the following we use the numbers from Table 8.3 as c_A.

The drop in the quotient $\mathrm{time}(A^H Ax)/\,\mathrm{time(axpy)}$ from D_{W8} to D_{W16} is caused by hitting the cache limit with the size of the involved vectors. The dimension of D_{W16} is $n_{16} = 12 \times 16^4$ which is 16 times the dimension $n_8 = 12 \times 8^4$ of D_{W8} and the time for computing $A^H Ax$ for `WDSparse16` is roughly 16 times the time for the same computation in case of `WDSparse8`. However, the time for an axpy operation for a vector $x \in \mathbb{C}^{n_{16}}$ does not scale by a factor of 16 compared to the time for an axpy operation of dimension n_8—the factor is close to 2×16. This is because storing a vector $x \in \mathbb{C}^n$ of dimension $n = 12 \times 16^4 = 786432$ requires $n * 16 = 12582912$ byte, i.e. 12 MiB, whereas our system has 8 MiB L3 cache. Hence, somewhere around a vector size of

Table 8.3: Comparison of the time for an axpy operation and the time for computing $A^H Ax$ for the sparse matrices of Table 8.1.

matrix	axpy	$A^H Ax$	$A^H Ax/\text{axpy}$	nnz/row
A_{msc}	5.49×10^{-6}	2.97×10^{-4}	54.2	21.6
A_{pois}	1.83×10^{-5}	2.35×10^{-3}	128.5	48.3
A_{smt}	3.16×10^{-5}	1.17×10^{-2}	368.9	145.8
A_{nd12k}	4.44×10^{-5}	4.52×10^{-2}	1017.1	395.0
D_{W8}	6.35×10^{-5}	8.14×10^{-3}	128.2	49.0
D_{W16}	1.90×10^{-3}	1.42×10^{-1}	74.8	49.0

5×10^5 we can expect a drop in performance for computing axpy operations. We performed time measurements for computing axpy operations for complex- and real-valued vectors of various sizes which support this claim and the results are displayed in Figure 8.2.

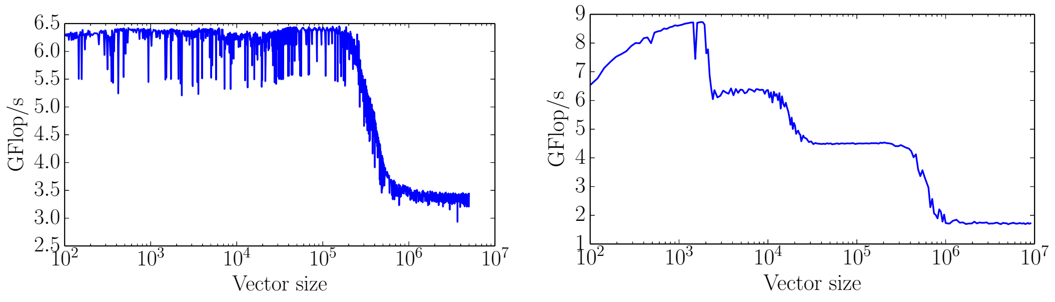


Figure 8.2: Performance of axpy operations averaged over 5000 operations depending on the vector size showing a drop in performance for larger vector sizes. We used complex-valued vectors on the left and real-valued vectors on the right.

8.1.2 Wilson-Dirac Operator Implementation

Representing the Wilson-Dirac operator from Section 4.1.1 as a sparse matrix in actual computations is possible, albeit not advisable. It introduces a lot of redundant storage and computations. Moreover, a tailored implementation is able to make use of symmetries that the operator is endowed with. For example, in the matrices $(I_4+\gamma_i)$ and $(I_4-\gamma_i)$ two of the columns are trivially linearly dependent on the other two. This can be exploited for reducing the cost of computing $(\tilde{D}_W\psi)(x)$ by about a factor of 2 as compared to a naïve implementation.

Independent of the representation as a stencil or a sparse matrix we can exploit the γ_5-symmetry (4.3) when computing $D_W^H D_W x$, because

$$D_W^H D_W x = \Gamma_5 D_W(\Gamma_5 D_W x).$$

This means that no special representation for D_W^H is needed.

We implemented the Wilson-Dirac operator $\tilde{D}_W$ given in equation (8.1) as a stencil operator. For this we store the configuration $\mathcal{U}$ from equation (4.1) via its gauge links U_μ^x as a list of $SU(3)$-matrices. As example operators we use $\tilde{D}_{W8}$ called `WDStencil8` and $\tilde{D}_{W16}$ called `WDStencil16` which are exactly the same operators as $\tilde{D}_{W8}$ and $\tilde{D}_{W16}$, respectively, in an efficient stencil implementation.

As for the sparse matrix representations we performed time measurements for applications of the operators $\tilde{D}_{W8}$ and $\tilde{D}_{W16}$ displayed in Table 8.4, Table 8.5 and Figure 8.3. Again, we use A to denote either $\tilde{D}_{W8}$ or $\tilde{D}_{W16}$.

Table 8.4: Time measurements for the Wilson-Dirac operators `WDStencil8` and `WDStencil16`. The measurements were performed in the same manner as described in Table 8.2.

matrix	axpy	A^HAx	A^HAX_1	A^HAX_{10}	A^HAX_{100}
$\tilde{D}_{W8}$	6.38×10^{-5}	3.03×10^{-3}	3.09×10^{-3}	2.80×10^{-2}	2.56×10^{-1}
$\tilde{D}_{W16}$	1.97×10^{-3}	5.30×10^{-2}	5.44×10^{-2}	4.93×10^{-1}	4.22×10^{0}

Overall, the comparisons yield similar results as those for our sparse matrix implementation. It is noteworthy, though, that computing A^HAX for X having one column is only slightly slower than computing A^HAx. This means that the efficiency of block methods for a small number of right-hand sides is increased. Furthermore, multiplications with $\tilde{D}_{W8}$ are 2 to 3 times faster than multiplications with D_{W8} for our implementations and the same holds for $\tilde{D}_{W16}$. Lastly, the break even point where multiplying with a block-vector X_m is faster than m matrix-vector multiplications with x is reached for a smaller number of columns m than for sparse matrices.

Table 8.5: Comparison of the time for an axpy operation and the time for computing $A^H Ax$ for the Wilson-Dirac operators `WDStencil8` and `WDStencil16`.

matrix	axpy	$A^H Ax$	$A^H Ax/\text{axpy}$
$\tilde{D}_{W8}$	6.38×10^{-5}	3.03×10^{-3}	47.6
$\tilde{D}_{W16}$	1.97×10^{-3}	5.30×10^{-2}	26.9

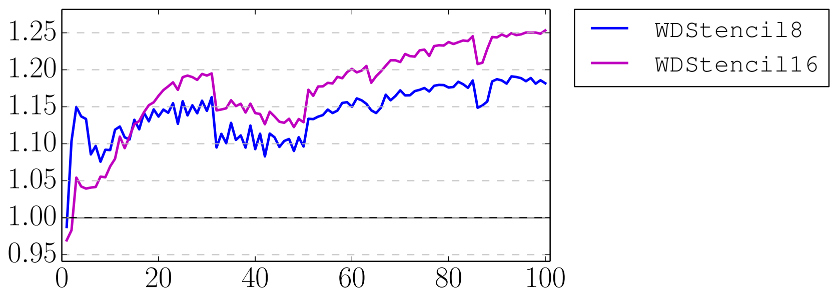


Figure 8.3: Time measurements for computing $A^H AX$ for the Wilson-Dirac operators $\tilde{D}_{W8}$ and $\tilde{D}_{W16}$ depending on the number m of columns of X. The measurements were performed in the same manner as described in Table 8.1.

8.1.3 Image Convolution Operator Implementation

In Section 4.2.2 we introduced image deconvolution and described the kernel for Gaussian blur (4.15) as a particular choice of a point spread function. We implemented the convolution operator from (4.16) using the Gaussian blur kernel as a stencil and applying reflective boundary conditions. For our experiments we chose the parameters $m_k = n_k = 13$, $n_c = m_c = 7$ and $\sigma = 3$. This represents a moderate blurring for the image sizes that we use as examples. As example operators we use A_{psf200} called `PSF200` and A_{psf400} called `PSF400` which implement the described convolution with Gaussian blur for images of size 200×200 and 400×400, respectively.

As for the operators that we have introduced before we performed time measurements for applications of the operators A_{psf200} and A_{psf400} displayed in Table 8.6, Table 8.7 and Figure 8.4.

We use A to denote either A_{psf200} or A_{psf400}. The tables and the figure reveal that an application of one of the operators A_{psf200} and A_{psf400} is more

Table 8.6: Time measurements for the operators `PSF200` and `PSF400`. The measurements were performed in the same manner as described in Table 8.2.

matrix	axpy	$A^H Ax$	$A^H AX_1$	$A^H AX_{10}$	$A^H AX_{100}$
A_{psf200}	1.74×10^{-5}	2.35×10^{-2}	2.36×10^{-2}	7.10×10^{-2}	6.57×10^{-1}
A_{psf400}	6.89×10^{-5}	9.53×10^{-2}	9.68×10^{-2}	2.81×10^{-1}	2.71×10^{0}

Table 8.7: Comparison of time for an axpy operation and for computing Ax for the operators `PSF200` and `PSF400`.

matrix	axpy	$A^H Ax$	$A^H Ax/\text{axpy}$
A_{psf200}	1.74×10^{-5}	2.35×10^{-2}	1347.8
A_{psf400}	6.89×10^{-5}	9.53×10^{-2}	1384.5

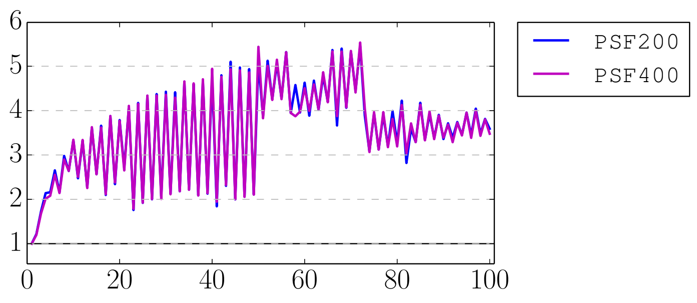


Figure 8.4: Time measurements for computing $A^H AX$ for the operators A_{psf200} and A_{psf400} depending on the number m of columns of X. The measurements were performed in the same manner as described in Figure 8.1.

costly compared to an axpy operation than for any operator introduced so far. Moreover, as for the Wilson-Dirac operator stencil implementations we have an early break even point for the block-vector multiplication. As we can see, even for small m computing $A^H AX_m$ is at least twice as fast as m multiplications $A^H Ax$ and can reach a factor of 5 for some m.

The saw-tooth pattern in Figure 8.4 is caused by a non-optimal memory access pattern for odd numbers of right-hand sides. Since we store X_m in a row-major format and have real values as data the start of every other row in X_m for $m = 2k + 1$ and $k = 0, 1, ...$ is not 128-bit aligned resulting in costly assembler opcodes to fill the SSE registers. This problem does not arise for even numbers of right-hand sides and for complex valued data as is the case, for example, for the Wilson-Dirac operator.

8.2 Lanczos-Type Processes Tests

In this section we conduct some tests for our implementation of the deflated block Lanczos-type process and the block-featured deflated Lanczos-type process. These two processes are the basis for the methods from Chapter 7. Hence, we validate the block Krylov subspace generated by our algorithm by checking the orthogonality of the generated Lanczos vectors.

We use `WDStencil8` to compute an orthonormal basis of the deflated block Krylov subspace with the deflated block Lanczos-type process and the block-featured deflated Lanczos-type process. But first of all, we compute a basis for the Krylov subspace $\mathcal{K}_k(\tilde{D}_{W8}^H \tilde{D}_{W8}, x)$ using the Lanczos process of Algorithm 2.2 and a random vector $x \in \mathbb{C}^n$. Figure 8.5 shows the magnitude of the entries of $V_k^H V_k$ on a logarithmic scale for $k = 600$. We see the well-known behaviour that the Lanczos vectors lose orthogonality as the iteration progresses and only for a narrow band close to the diagonal of $V_k^H V_k$ we have small entries close to machine precision.

Now, we can compare this to the Lanczos-type processes of Chapter 7. In Figure 8.6 we display four plots of $V_k^H V_k$ where V_k was computed by the deflated block Lanczos-type process of Algorithm 7.2. From the top left to bottom right plot we built $\mathcal{K}_k^{\text{defl}}(\tilde{D}_{W8}^H \tilde{D}_{W8}, X_i)$ for $k = 600$, $X_i \in \mathbb{C}^{n\times 6}$ and $i = 1, 2, 3, 4$. The columns of X_1 were chosen randomly. In X_2 we computed one column to be nearly linearly dependent on the remaining columns of X_2. We enforced similar linear dependency in X_3 and X_4 except that in X_3 we have 3 inexact deflations and in X_4 all vectors but one are deflated. Moreover, we constructed X_3 and X_4 such that deflation occurs after some steps of the deflated block Lanczos-type process.

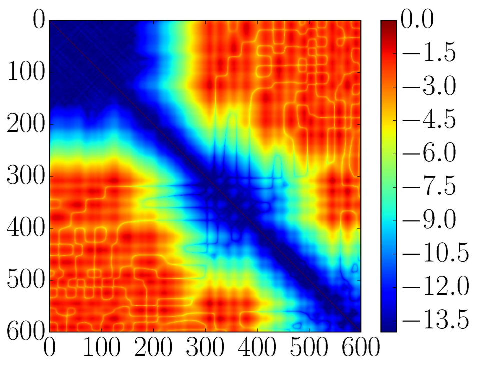


Figure 8.5: Orthogonality of the Lanczos vectors generated by the Lanczos process. We display the logarithm to base 10 of the magnitude of the entries of the matrix $V^H V$.

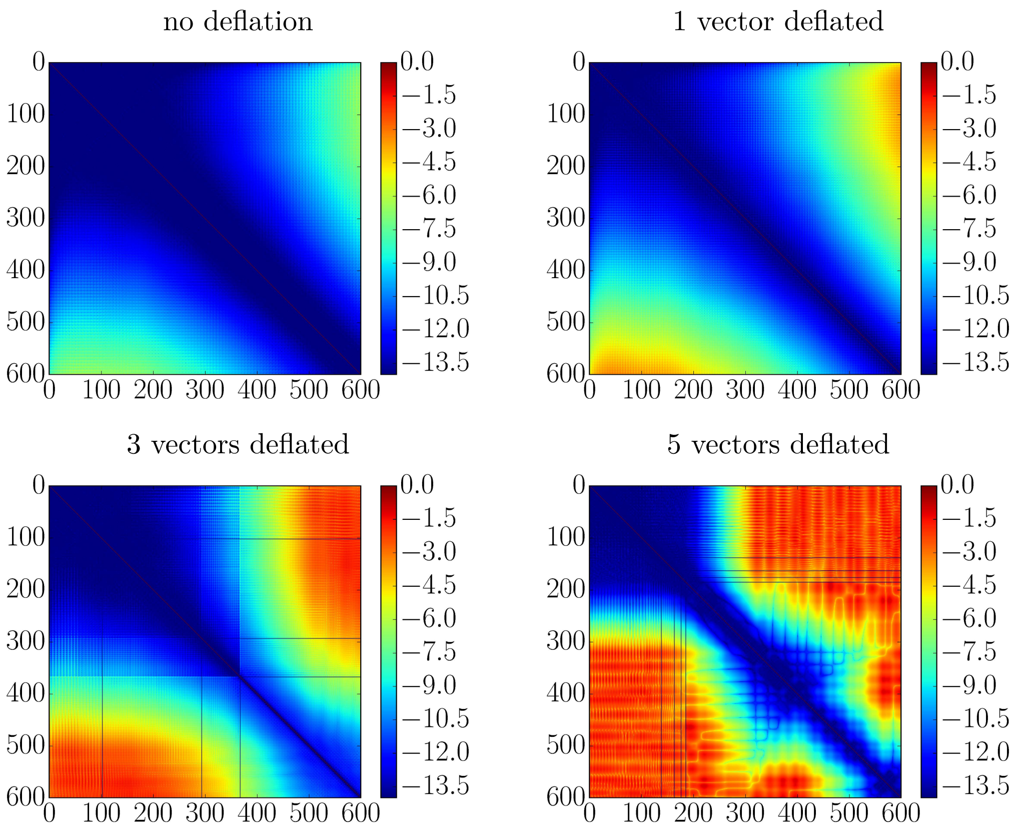


Figure 8.6: Orthogonality of the Lanczos vectors generated by the deflated block Lanczos-type process. We display the logarithm to base 10 of the magnitude of the entries of the matrix $V^H V$.

In these lower two pictures we can see dark blue bars as an indicator of the orthogonalisation against inexactly deflated vectors. Comparing the plots for X_1, X_2 and X_3 to the Lanczos process from Figure 8.5 shows that the orthogonality of the vectors spanning the block Krylov subspace is better than that of the Lanczos vectors spanning a Krylov subspace of the same dimension. For the former we have to spend more work on orthogonalisation, though. If all vectors except one are deflated as in the last plot of Figure 8.6 then the plot looks similar to Figure 8.5. This means that we do not obtain worse orthogonality than with the non-block Lanczos process.

For the block-featured deflated Lanczos-type process the same test for checking the orthogonality of V_k by computing $V_k^H V_k$ as in Figure 8.6 was performed and the results can be found in Figure 8.7.

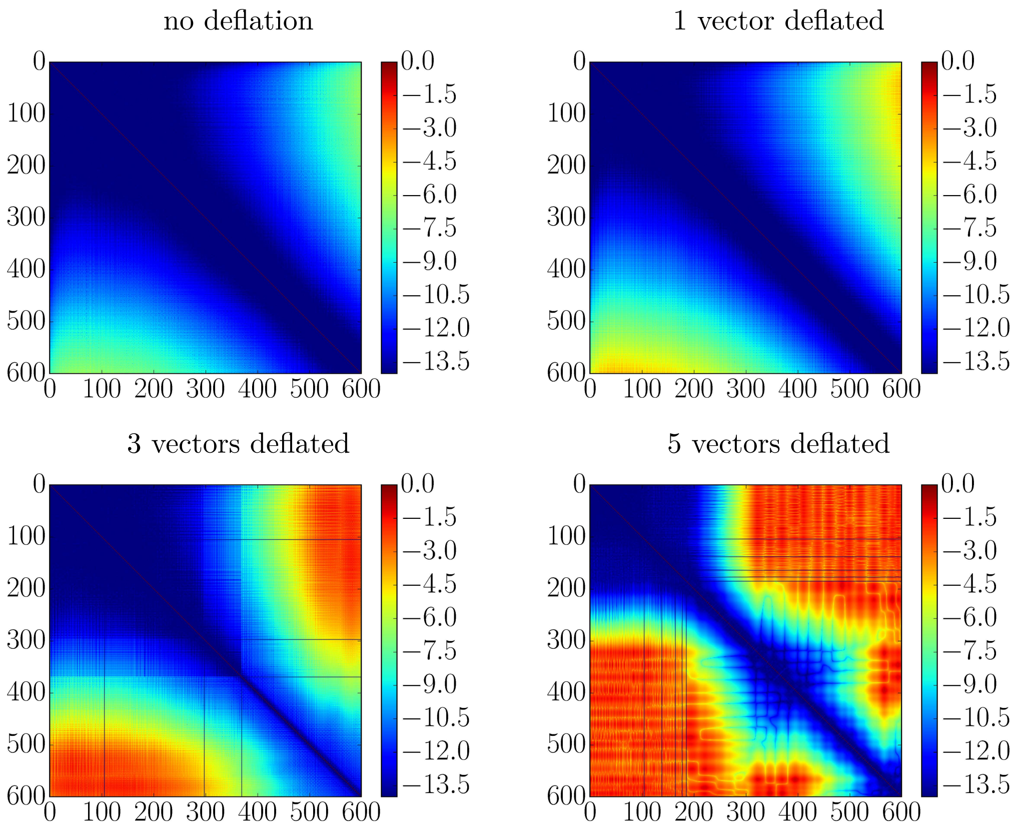


Figure 8.7: Orthogonality of the Lanczos vectors generated by the block-featured deflated Lanczos-type process. We display the logarithm to base 10 of the magnitude of the entries of the matrix $V^H V$.

We can observe an almost identical behaviour to the deflated block Lanczos-type process in Figure 8.6. The only difference is the step in which deflation occurs as we have explained in Section 7.3. For instance, in our example with 5 deflated vectors the first deflation in the deflated block Lanczos-type process (Figure 8.6) occurs when the subspace reaches a dimension of around 140 whereas in the block-featured deflated Lanczos-type process (Figure 8.7) this happens at about 100.

8.3 Comparison of Methods for Solving Shifted Block Systems

In this section we will present the numerical results for the tests that we have conducted for the applications from Chapter 4. First, we present and discuss the results for applications from lattice QCD. Afterwards, we focus on results for image deconvolution. For both applications we want to have some realistic examples at hand. In Section 8.1 we have already defined the operators that we use for our tests. In the following subsections we start by describing the test settings, e.g. how we obtained the shifts for the family of shifted systems with multiple right-hand sides

$$(A + \sigma_i I)x_{i,j} = b_j, \quad i = 1, \dots, s, \quad j = 1, \dots, m \tag{1.1}$$

and which right-hand sides we chose.

8.3.1 Lattice QCD - Prerequisites

For our tests with the Wilson-Dirac matrices we use several numbers of random right-hand sides. The systems (1.1) for the operators `WDSparse8` and `WDStencil8` are solved for up to 30 right-hand sides. For the operators representing a 16^4 lattice, i.e. `WDSparse16` and `WDStencil16`, we use up to 10 right-hand sides.

For stopping the iteration we use the condition (5.7), i.e. we monitor the residuals and stop the iteration for every shifted system depending on the number of shifts and the corresponding weight of the partial fraction expansion.

We simulate the computations involved in the application of the overlap operator from Section 4.1.3 by computing $(D_W^H D_W)^{-1/2}x$ as needed in equation (4.7). For this we compute rational approximations to $f(x) = x^{-1/2}$ for the Wilson-Dirac operators from Section 8.1. For each of the operators we use two different approximations that approximate $f(x)$ to a given accuracy. The rational approximations were obtained via a Zolotarev approximation, see Section 3.2.4, for $f(x)$ with respect to the spectrum of the operators. The resulting approximations can be found in Table 8.8.

For simulating the computations in the Rational Hybrid Monte Carlo algorithm, see (4.9) in Section 4.1.4, we computed rational approximations to $f(x) = x^{-1/4}$. As for the approximations to $f(x) = x^{-1/2}$ we computed a couple of approximations also displayed in Table 8.8. For these, however, we used Kronecker's algorithm 3.1 from Section 3.2.3 to obtain an n-point Padé approximation with the interpolation nodes as given by equation (3.10).

Table 8.9 and Table 8.10 display the complete sets of shifts σ_j and weights ω_j (using the notation from equation (3.7)) of the partial fraction expansions corresponding to the rational approximations from Table 8.8 as (shift,weight)-pairs. Note that we are using $[n-1/n]$-type rational approximation and therefore $\pi \equiv 0$ in (3.7).

One important conclusion that we can draw from the tables is that small shifts which result in a large condition number of the shifted system go along with small weights. This means that even though in general these systems are harder to solve we do not have to solve them to a high precision when we want to compute a matrix function. However, the weights cannot alleviate the large condition number completely and in computations we see that solving the systems belonging to smaller shifts still need more iterations.

The n-point Padé approximations are non-optimal since they were created based on heuristically chosen interpolation nodes. In Figure 8.8 we compare the Zolotarev approximations pfe_{Z8s7} and pfe_{Z8s14} from Table 8.8 to n-point Padé approximations resulting in the partial fraction expansions pfe_{P8s7} and pfe_{P8s14} with the same number of poles. Especially the plot for 14 poles shows that our n-point Padé approximation can be worse by some orders of magnitude caused by the heuristics. This means that one might achieve the same accuracy with a lower number of poles in the partial fraction expansion. For our tests, however, this is irrelevant since we are only interested in having realistic partial fraction expansions for comparing the performance of the algorithms.

8.3.2 Lattice QCD - Methods

Before actually presenting time measurements we need to give some custom explanations for some of the used algorithms. Moreover, we give estimates for the performance of the algorithms by applying our cost model. For this we use the measured ratio $\text{time}(A^H Ax)/\,\text{time}(axpy)$ from Section 8.1 as constant c_A

Table 8.8: Rational approximations for the Wilson-Dirac operators D_{W8} and D_{W16}. We computed rational approximations with an error $\frac{|f(x)-r(x)|}{|f(x)|}$ of at most the value specified in the column labelled error on the interval $[\lambda_{min}(A^H A), \lambda_{max}(A^H A)]$. For every operator we use obtained an approximation with an error less than 1×10^{-5} and another one with an error less than 1×10^{-10}. Additionally, we display the number of poles of the corresponding partial fraction expansion. The shifts and weights defining the partial fraction expansions are displayed in the tables 8.9 and 8.10.

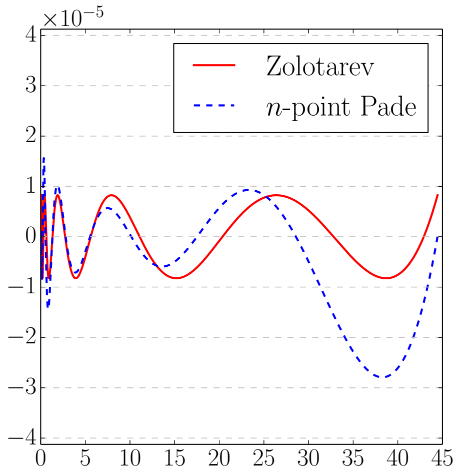
operator A	$\lambda_{min}(A^H A)$	$\lambda_{max}(A^H A)$	function	type	name	poles	error
D_{W8}	1.85×10^{-2}	4.44×10^{1}	$x^{-1/2}$	Zolotarev	pfe_{Z8s7}	7	8.25×10^{-6}
					pfe_{Z8s14}	14	1.70×10^{-11}
D_{W16}	5.70×10^{-4}	4.48×10^{1}	$x^{-1/2}$	Zolotarev	pfe_{Z16s10}	10	3.15×10^{-6}
					pfe_{Z16s18}	18	4.11×10^{-11}
D_{W8}	1.85×10^{-2}	4.44×10^{1}	$x^{-1/4}$	n-point Padé	pfe_{P8p9}	9	5.67×10^{-6}
					pfe_{P8p18}	18	5.88×10^{-11}
D_{W16}	5.70×10^{-4}	4.48×10^{1}	$x^{-1/4}$	n-point Padé	pfe_{P16p13}	13	3.78×10^{-6}
					pfe_{P16p23}	23	8.96×10^{-11}
D_{W8}	1.85×10^{-2}	4.44×10^{1}	$x^{-1/2}$	n-point Padé	pfe_{P8s7}	7	2.79×10^{-5}
					pfe_{P8s14}	14	3.48×10^{-9}
D_{W16}	5.70×10^{-4}	4.48×10^{1}	$x^{-1/2}$	n-point Padé	pfe_{P16s10}	10	4.71×10^{-5}
					pfe_{P16s18}	18	1.02×10^{-8}

Table 8.9: Partial fraction expansions corresponding to the rational approximations for $f(x) = x^{-1/2}$ from Table 8.8. We display the poles of the partial fraction expansions as (shift,weight)-pairs.

pfe_{Z8s7}	pfe_{Z8s14}	pfe_{Z16s10}	pfe_{Z16s18}
$(2.76 \times 10^{-3}, 7.01 \times 10^{-2})$	$(6.67 \times 10^{-4}, 3.33 \times 10^{-2})$	$(7.32 \times 10^{-5}, 1.13 \times 10^{-2})$	$(2.20 \times 10^{-5}, 6.04 \times 10^{-3})$
$(3.57 \times 10^{-2}, 1.12 \times 10^{-1})$	$(6.59 \times 10^{-3}, 3.81 \times 10^{-2})$	$(9.04 \times 10^{-4}, 1.72 \times 10^{-2})$	$(2.19 \times 10^{-4}, 6.97 \times 10^{-3})$
$(1.92 \times 10^{-1}, 2.21 \times 10^{-1})$	$(2.20 \times 10^{-2}, 4.83 \times 10^{-2})$	$(4.49 \times 10^{-3}, 3.18 \times 10^{-2})$	$(7.38 \times 10^{-4}, 8.98 \times 10^{-3})$
$(9.08 \times 10^{-1}, 4.67 \times 10^{-1})$	$(5.60 \times 10^{-2}, 6.56 \times 10^{-2})$	$(1.92 \times 10^{-2}, 6.28 \times 10^{-2})$	$(1.91 \times 10^{-3}, 1.24 \times 10^{-2})$
$(4.29 \times 10^{0}, 1.04 \times 10^{0})$	$(1.29 \times 10^{-1}, 9.23 \times 10^{-2})$	$(7.89 \times 10^{-2}, 1.26 \times 10^{-1})$	$(4.49 \times 10^{-3}, 1.77 \times 10^{-2})$
$(2.31 \times 10^{1}, 2.84 \times 10^{0})$	$(2.85 \times 10^{-1}, 1.33 \times 10^{-1})$	$(3.23 \times 10^{-1}, 2.55 \times 10^{-1})$	$(1.01 \times 10^{-2}, 2.57 \times 10^{-2})$
$(2.98 \times 10^{2}, 2.30 \times 10^{1})$	$(6.18 \times 10^{-1}, 1.93 \times 10^{-1})$	$(1.33 \times 10^{0}, 5.24 \times 10^{-1})$	$(2.24 \times 10^{-2}, 3.77 \times 10^{-2})$
	$(1.33 \times 10^{0}, 2.83 \times 10^{-1})$	$(5.68 \times 10^{0}, 1.13 \times 10^{0})$	$(4.93 \times 10^{-2}, 5.55 \times 10^{-2})$
	$(2.89 \times 10^{0}, 4.23 \times 10^{-1})$	$(2.82 \times 10^{1}, 3.03 \times 10^{0})$	$(1.08 \times 10^{-1}, 8.19 \times 10^{-2})$
	$(6.39 \times 10^{0}, 6.50 \times 10^{-1})$	$(3.49 \times 10^{2}, 2.47 \times 10^{1})$	$(2.36 \times 10^{-1}, 1.21 \times 10^{-1})$
	$(1.47 \times 10^{1}, 1.06 \times 10^{0})$		$(5.18 \times 10^{-1}, 1.80 \times 10^{-1})$
	$(3.75 \times 10^{1}, 2.00 \times 10^{0})$		$(1.14 \times 10^{0}, 2.68 \times 10^{-1})$
	$(1.25 \times 10^{2}, 5.24 \times 10^{0})$		$(2.52 \times 10^{0}, 4.05 \times 10^{-1})$
	$(1.24 \times 10^{3}, 4.53 \times 10^{1})$		$(5.68 \times 10^{0}, 6.28 \times 10^{-1})$
			$(1.34 \times 10^{1}, 1.03 \times 10^{0})$
			$(3.46 \times 10^{1}, 1.95 \times 10^{0})$
			$(1.17 \times 10^{2}, 5.10 \times 10^{0})$
			$(1.16 \times 10^{3}, 4.39 \times 10^{1})$

Table 8.10: Partial fraction expansions corresponding to the rational approximations for $f(x) = x^{-1/4}$ from Table 8.8. We display the poles of the partial fraction expansions as (shift,weight)-pairs.

pfe_{P8p9}	pfe_{P8p18}	pfe_{P16p13}	pfe_{P16p23}
$(1.24 \times 10^{-3}, 4.71 \times 10^{-3})$	$(3.04 \times 10^{-4}, 1.61 \times 10^{-3})$	$(4.23 \times 10^{-5}, 3.77 \times 10^{-4})$	$(1.35 \times 10^{-5}, 1.56 \times 10^{-4})$
$(9.72 \times 10^{-3}, 1.05 \times 10^{-2})$	$(2.08 \times 10^{-3}, 2.82 \times 10^{-3})$	$(3.41 \times 10^{-4}, 8.66 \times 10^{-4})$	$(9.44 \times 10^{-5}, 2.84 \times 10^{-4})$
$(3.70 \times 10^{-2}, 2.37 \times 10^{-2})$	$(5.89 \times 10^{-3}, 4.16 \times 10^{-3})$	$(1.35 \times 10^{-3}, 2.07 \times 10^{-3})$	$(2.79 \times 10^{-4}, 4.48 \times 10^{-4})$
$(1.29 \times 10^{-1}, 6.13 \times 10^{-2})$	$(1.28 \times 10^{-2}, 6.07 \times 10^{-3})$	$(5.02 \times 10^{-3}, 5.65 \times 10^{-3})$	$(6.47 \times 10^{-4}, 7.15 \times 10^{-4})$
$(4.73 \times 10^{-1}, 1.71 \times 10^{-1})$	$(2.50 \times 10^{-2}, 9.07 \times 10^{-3})$	$(1.93 \times 10^{-2}, 1.58 \times 10^{-2})$	$(1.38 \times 10^{-3}, 1.19 \times 10^{-3})$
$(1.83 \times 10^{0}, 4.86 \times 10^{-1})$	$(4.69 \times 10^{-2}, 1.40 \times 10^{-2})$	$(7.30 \times 10^{-2}, 4.08 \times 10^{-2})$	$(2.89 \times 10^{-3}, 2.08 \times 10^{-3})$
$(7.46 \times 10^{0}, 1.48 \times 10^{0})$	$(8.69 \times 10^{-2}, 2.24 \times 10^{-2})$	$(2.53 \times 10^{-1}, 9.58 \times 10^{-2})$	$(6.10 \times 10^{-3}, 3.71 \times 10^{-3})$
$(3.72 \times 10^{1}, 6.34 \times 10^{0})$	$(1.63 \times 10^{-1}, 3.65 \times 10^{-2})$	$(7.97 \times 10^{-1}, 2.10 \times 10^{-1})$	$(1.30 \times 10^{-2}, 6.60 \times 10^{-3})$
$(7.88 \times 10^{2}, 2.20 \times 10^{2})$	$(3.09 \times 10^{-1}, 6.05 \times 10^{-2})$	$(2.34 \times 10^{0}, 4.49 \times 10^{-1})$	$(2.78 \times 10^{-2}, 1.15 \times 10^{-2})$
	$(5.95 \times 10^{-1}, 1.01 \times 10^{-1})$	$(6.71 \times 10^{0}, 9.98 \times 10^{-1})$	$(5.84 \times 10^{-2}, 1.95 \times 10^{-2})$
	$(1.16 \times 10^{0}, 1.68 \times 10^{-1})$	$(2.04 \times 10^{1}, 2.57 \times 10^{0})$	$(1.19 \times 10^{-1}, 3.21 \times 10^{-2})$
	$(2.27 \times 10^{0}, 2.84 \times 10^{-1})$	$(8.12 \times 10^{1}, 1.03 \times 10^{1})$	$(2.37 \times 10^{-1}, 5.12 \times 10^{-2})$
	$(4.53 \times 10^{0}, 4.88 \times 10^{-1})$	$(1.52 \times 10^{3}, 3.58 \times 10^{2})$	$(4.56 \times 10^{-1}, 7.99 \times 10^{-2})$
	$(9.26 \times 10^{0}, 8.81 \times 10^{-1})$		$(8.54 \times 10^{-1}, 1.23 \times 10^{-1})$
	$(2.01 \times 10^{1}, 1.75 \times 10^{0})$		$(1.56 \times 10^{0}, 1.88 \times 10^{-1})$
	$(4.96 \times 10^{1}, 4.26 \times 10^{0})$		$(2.83 \times 10^{0}, 2.89 \times 10^{-1})$
	$(1.71 \times 10^{2}, 1.68 \times 10^{1})$		$(5.10 \times 10^{0}, 4.52 \times 10^{-1})$
	$(2.99 \times 10^{3}, 5.91 \times 10^{2})$		$(9.30 \times 10^{0}, 7.32 \times 10^{-1})$
			$(1.75 \times 10^{1}, 1.26 \times 10^{0})$
			$(3.51 \times 10^{1}, 2.44 \times 10^{0})$
			$(8.14 \times 10^{1}, 5.86 \times 10^{0})$
			$(2.69 \times 10^{2}, 2.31 \times 10^{1})$
			$(4.60 \times 10^{3}, 8.15 \times 10^{2})$



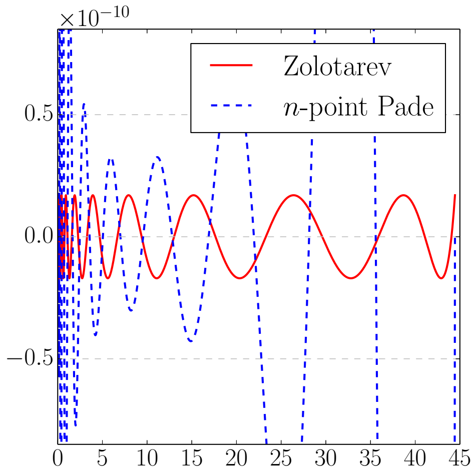


Figure 8.8: Comparison of Zolotarev and n-point Padé approximation where we used equation (3.10) to compute heuristically chosen interpolation nodes. On the left we compare pfe_{Z8s7} and pfe_{P8s7} and on the right pfe_{Z8s14} and pfe_{P8s14}, respectively. Displayed is the relative error $(x^{-1/2} - \text{pfe}(x))/x$ on the relevant interval.

for the corresponding operator. For those methods that perform matrix-block-vector multiplications instead of matrix-vector multiplications we measured $\text{time}(A^H AX_m)/(m \cdot \text{time}(axpy))$ to obtain $c_A^{\square}(m)$.

Shifted Methods

We want to apply shifted methods to (1.1) by applying them to all the right-hand sides separately. As already discussed after Proposition 5.1, shifted CG should almost always perform better than CG. Hence, the more important analysis that we want to address with numerical examples here is how restarted shifted CG performs as compared to shifted CG. For this we pick up on Proposition 5.5 and obtain from (5.15) that the quotient $\frac{\hat{k}}{\tilde{k}}$ is allowed to range from about 1 to $1 + \frac{2s-1}{c_A+5}$ for restarted shifted CG being faster than shifted CG depending on the number of steps after which the shifted systems converge. Note that we assumed for the cost model that every shifted system was solved after $\tilde{k}_i = k$ steps.

In Figure 8.9 we display four test cases in which we applied our cost model. Since there is no dependence on the number of right-hand sides we performed tests with one random right-hand side for the Wilson-Dirac operators described in Section 8.1 using the values of c_A that we measured there. Furthermore, we used the partial fraction expansions from tables 8.8 to 8.10 and the iteration was stopped as soon as the relative residual norm fulfilled the condition (5.7) for $\epsilon = 1 \times 10^{-5}$. For rsCG we performed restarts after 100 steps.

In three cases the quotient $\frac{\hat{k}}{\tilde{k}}$ obtained from our test exceeded the estimate from (5.15) and in two of which rsCG was actually slower than sCG. This leaves us with one false negative and no false positives from Proposition 5.5 for this example.

As a result from these tests one should not consider the estimate from (5.15) using our cost model as a definitive bound. Instead, one could use it as an indicator that rsCG should not be used if a large number of restarts can be expected or if the estimate is close to 1. In practical implementations it could make sense to solve one shifted system from (1.1) with sCG to get estimates for the number of steps after which the shifted systems are solved. These numbers can be used in (5.15) to compute the estimate bound. Moreover, the number of steps sCG performs can be used to obtain a lower bound for the number of restarts that rsCG has to perform. The quotient of iteration steps of rsCG over the iteration steps of sCG is a function of the number of restarts. Hence, knowledge of the lower bound of restarts combined with some estimate with the inflicted slow-down of convergence might help to support the decision if rsCG is to be used instead of sCG.

The profound impact of the restarts on the iteration is displayed in Figure 8.10. Since our cost model does not consider the effect of restarts it cannot be used to predict the number of rsCG steps. However, since it yielded no false positives in our test cases, it could prove useful if the iteration steps of rsCG can be guessed.

Since neither sCG nor rsCG can be identified as being always a better choice we will use both of them in later comparisons.

Methods for Multiple Right-Hand Sides

When we use methods for multiple right-hand sides to solve (1.1) then we need to apply them to all the shifts separately. This applies to the Single Seed Method, Seed-CG, incremental eigCG as well as the block methods BlockCG, BCGAdQ and BCGrQ. In order to make use of the cost model analysis that we have presented after each of the algorithms we need an informed guess for their iteration numbers as for restarted shifted CG. But unlike restarted shifted CG there are no parameters in the methods—except for incremental eigCG—which we could use for tuning the iteration number.

As parameters of the incremental eigCG method we use $n_{\text{ev}} = 10$ and $c_{\text{rest}} = 50$ which was found to result in goods eigenvector approximations in [SO10].

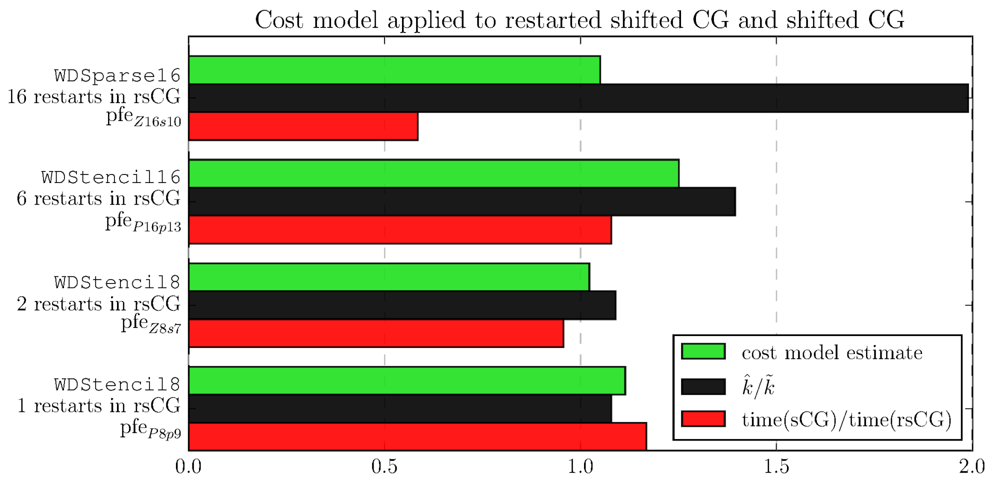


Figure 8.9: Comparison of rsCG and sCG using our cost model, actual iteration numbers, and time measurements for four test cases. The green bar represents the estimate bound from (5.15) using the actual number of steps after which each shifted system converged in sCG. The blue bar shows the quotient of rsCG iteration steps $\hat{k}$ and sCG iteration steps $\tilde{k}$. The red bar displays the comparison of measured times where values larger than one indicate that rsCG solved the systems faster than sCG. Additionally, we display the number of restarts that rsCG performed on the left side.

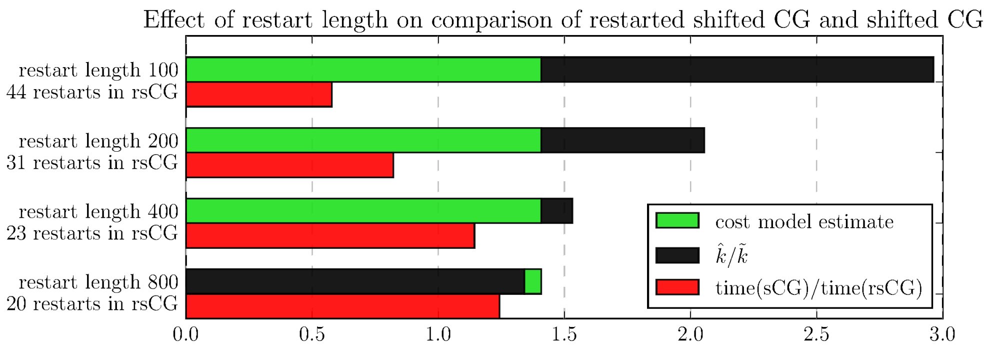


Figure 8.10: Comparison of sCG and rsCG with different restart lengths. We performed test runs for `WDStencil16` and pfe_{P16p23} and stopped the iteration as soon as the relative residual norm fulfilled (5.7). We use the same notation and colour-coding as in Figure 8.9.

We chose $m_{\text{eig}} = m - 1$ to obtain the best reduction in number of iterations which we found to often result in the best time measurements too.

Shifted Block Methods

The methods DSBlockCG and BFDSCG from Chapter 7 need no special tuning and can be applied directly to the systems (1.1). As deflation tolerance for exact deflation we use 5×10^{-14} and the inexact deflation tolerance is set one order of magnitude lower than the target relative residual norm. For example, for a target relative residual norm of 1×10^{-10} we use an inexact deflation tolerance of 1×10^{-11}.

Cost Models for All Tested Methods

To test our cost models and comparisons we computed the values of o_{x} for all the methods we presented in Chapters 5, 6, and 7 using an example setting. This means, we computed o_{cg} (2.16), o_{scg} (5.9), o_{rscg} (5.14), o_{blockcg} (6.23), o_{bcgadq} (6.25), o_{bcgrq} (6.26), o_{eigcg} (6.14), o_{ssm} (6.3), o_{seedcg} (6.8), $o_{\text{dsblockcg}}$ (7.33), and o_{bfdscg} (7.44).

As test setting we solved (1.1) for the operator `WDStencil8`, 10 random right-hand sides and shifts and weights stemming from pfe_{P8p18}. The target relative residual norms were computed using (5.7). From Section 8.1 we obtain $c_A \approx 47.6$ for a matrix-vector multiplication $c_A^{\square}(m) \approx 47.6/1.1 \approx 43.3$ for a matrix-block-vector multiplication with a block-size of $m = 10$. The time for a vector operation is about 6.28×10^{-5} seconds.

First we solved the systems belonging to the first right-hand side using CG to obtain the number of steps in which CG solves them. Then we used the following assumptions to feed our cost models that are inspired by observations during our test runs:

- The same number of iterations is needed to solve the systems belonging to different right-hand sides with CG.
- For rsCG the number of iteration steps increases by about 5% per restart.
- The number of iterations in the Single Seed Method and Seed-CG reduces linearly along the right-hand sides down to 90% for the last right-hand side.
- The number of iterations in incremental eigCG reduces linearly along the right-hand sides to 65% for the last right-hand side.

- All block methods reduce the dimension of the spanned block Krylov subspace to 65% of the accumulated dimension of the Krylov subspaces that CG needs to span for solving all the systems.

The results can be found in Table 8.11.

Table 8.11: Comparison of cost model estimates and time measurements for `WDStencil8` and all the contestant methods. The estimated time is the product of the estimated vector operations from our cost model and the measured time per vector operation (6.28×10^{-5} seconds).

method	vector operations	estimated time	measured time	ratio
CG	1530660	96.15	101.50	0.95
sCG	257376	16.17	19.28	0.84
rsCG	235039	14.76	15.95	0.93
SSM	1956883	122.93	126.79	0.97
Seed-CG	1558279	97.89	105.00	0.93
iEigCG	2023701	127.13	131.40	0.97
BlockCG	1846104	115.97	128.89	0.90
BCGAdQ	2489214	156.37	216.45	0.72
BCGrQ	2110914	132.60	204.34	0.65
DSBlockCG	575507	36.15	48.99	0.74
BFDSCG	638232	40.09	65.91	0.61

The most notable finding from Table 8.11 is that our cost model estimates predict the right order of the methods with respect to their runtime. However, some of the estimates in the table are closer to the actually measured time. This can be explained by some additional computations that are not reflected in our cost model. The most notable deviations from the estimate appear in the block methods except for plain BlockCG. This is caused by the cost of operations on small matrices which can be substantial. The remaining methods largely consist of vector operations and block-vector operations that are represented in our cost models.

Since rsCG computes the approximation x to $f(A)b$ directly and does not yield solutions for the shifted systems, we cannot check our results by computing residuals. However, we can compare the solutions computed by the different methods to each other as a sanity check for our implementations. As we can see in Figure 8.11 all the approximations are close by. Since we are using (5.7) as a stopping criterion we cannot guarantee that the error is smaller than the target relative residual norm plus the approximation error introduced by the rational approximation. However, Figure 8.11 suggests that the computed solutions are not too far off.

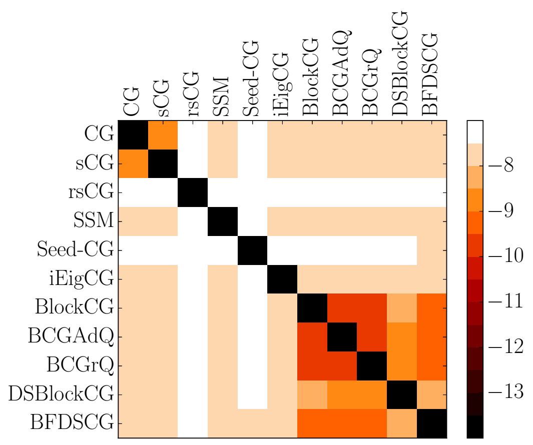


Figure 8.11: Check of the quality of the solution for all tested methods. We display the logarithm to base 10 of $\|x_i - x_j\|_2$ for x_i and x_j being the solution obtained by method i and method j. The largest value is 5.48×10^{-08} and the smallest off-diagonal value is found to be 2.15×10^{-10}. Here we chose $\epsilon = 1 \times 10^{10}$ as stopping criterion in (5.7).

A conclusion we draw from Table 8.11 and some more observations is that we do not need to include all the methods in every test in the remainder of this chapter. Even though seed, deflation, and block methods represent good methods for solving systems with multiple right-hand sides they cannot outcompete methods that can solve shifted systems like (1.1). Therefore, we will only compare sCG, rsCG, DSBlockCG, and BFDSCG in that case. However, we will include CG in some of our tests to have a comparison to the naïve approach for solving (1.1). Still, some of the methods we ruled out for systems that involve shifts will have a return in Section 8.4 when we consider solving non-shifted systems with multiple right-hand sides.

8.3.3 Lattice QCD - Results

In the following we want to present the results of the tests that we have conducted focussing mainly on the algorithms CG, sCG, rsCG, DSBlockCG, and BFDSCG. Our main goal will be to compare different (mostly) realistic QCD test settings and the runtime dependence on the number of right-hand sides.

We refrain from presenting convergence plots for two reasons. First, since we want to apply a matrix function the individual residuals for the involved systems are not of paramount interest. Second, displaying the convergence for the m right-hand sides and s shifts would be of limited value. Hence, we only checked whether the solutions are close in the sense of Figure 8.11 instead.

For each run with k right-hand sides we used the same k random right-hand sides for all methods. We encountered no deflation in our test settings.

Realistic Tests for a 8^4 Lattice

Figure 8.12 shows a realistic setting for solving the family of systems (1.1) for the operator `WDStencil8` with 1 to 30 random right-hand sides and the 18 shifts and weights stem from pfe_{P8p18}. The target relative residual norms were computed using (5.7) matching pfe_{P8p18}. In Table 8.11 we presented the actual numbers and cost model estimates for $m = 10$ right-hand sides in this test setting. This is why we include all methods in Figure 8.12 for once.

We can see that if we use the number of matrix-vector multiplications or matrix-block-vector multiplications as a metric for comparing the methods then all other methods are better than CG. Moreover, CG needs about 8 times the number of matrix-vector multiplications than what sCG needs to solve all systems. For larger numbers of right-hand sides DSBlockCG and BFDSCG even need about half the number of matrix-vector multiplications of sCG.

However, as our cost model estimated in Table 8.11 this is not in one-to-one correspondence when we compare actual time measurements. There, we can see that most methods are even slower than CG and that sCG as well as rsCG are the best methods to solve the systems. Furthermore, even though DSBlockCG and BFDSCG can solve the systems with less matrix-vector multiplications than every other method they cannot outperform sCG and rsCG when it comes to time measurements. This reflects the fact that the additional work to be invested to handle multiple right-hand sides is not negligible. We will see later that the situation changes in favour of DSBlockCG and BFDSCG not only when the cost c_A of a matrix-vector multiplication increases but also when the matrix is less well conditioned.

From Figure 8.13 we can get a glimpse of the effect of c_A on time measurements. There, we used the exact same test setting as in Figure 8.12 except for using the operator `WDSparse8` resulting in $c_A \approx 128.2$. We see that the numbers of matrix-vector multiplications are exactly the same as they should be and the gap between DSBlockCG/BFDSCG and sCG/rsCG closes. The remaining methods get closer to the time performance of CG, too, and BlockCG

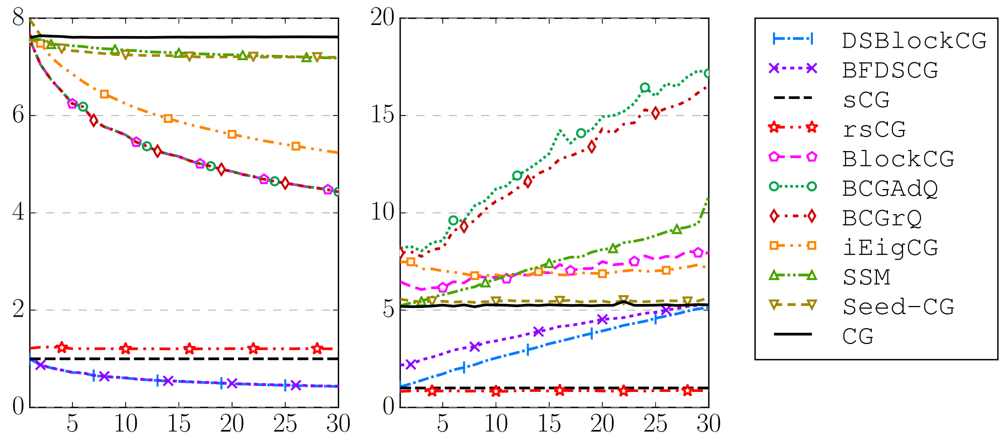


Figure 8.12: Comparison of all implemented methods for solving the family of linear systems (1.1) for the operator `WDStencil8` with 1 to 30 right-hand sides. The left plot displays the relative number of matrix-vector multiplications and the right plot shows the relative time normalised with respect to sCG in both cases.

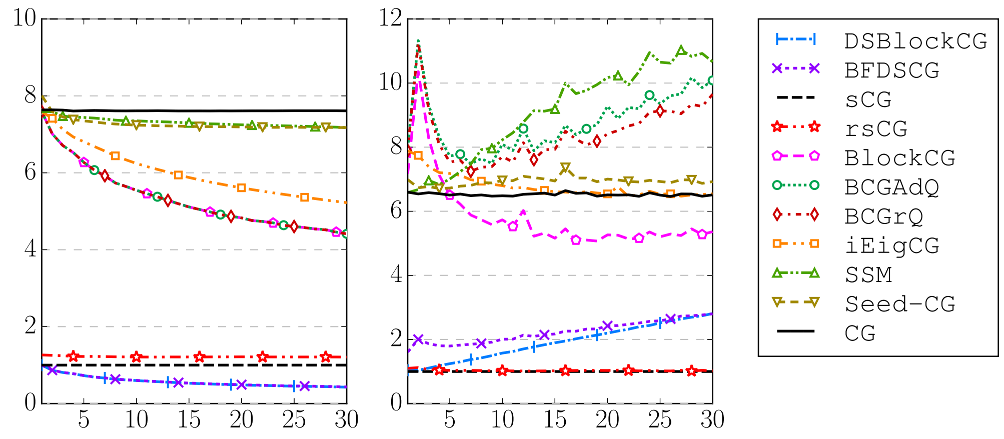


Figure 8.13: The same test setting as in Figure 8.12 except for using `WDSparse8`. The spike for the block methods at 2 right-hand sides stems from our sparse matrix-block-vector multiplication implementation as described for Figure 8.1. Left: relative mvms, right: relative time.

even overtakes CG.

Realistic Tests for a 16^4 Lattice

Results for tests with a larger and thus more realistic lattice are displayed in Figure 8.14. We used the operator `WDStencil16` with 1 to 10 random right-hand sides and we applied pfe_{Z16s18} and pfe_{P16p23}. For the computations with pfe_{Z16s18} we chose a restart length of 400 and a restart length of 100 for pfe_{P16p23} in rsCG.

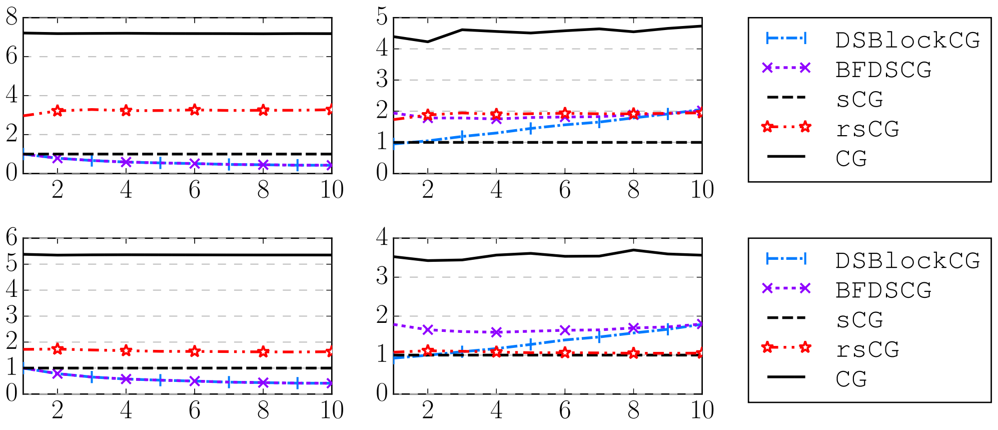


Figure 8.14: Comparison of some selected methods for the operator `WDStencil16` with 1 to 10 right-hand sides. In the top plots we used pfe_{P16p23} and in the bottom plots the shifts and weight stem from pfe_{Z16s18}. The left plots display the relative number of matrix-vector multiplications and the right plots show the relative time normalised with respect to sCG in both cases.

In Figure 8.14 we can see that DSBlockCG and BFDSCG perform slightly better when compared to sCG than in the plots in Figure 8.12 where we used a 8^4 lattice. However, in the time measurements the handling of multiple right-hand sides still outweighs the possible savings.

Test Runs Including Deflation

Even though we never observed deflation happening in our QCD test cases we cannot ignore the possibility of linear dependency while spanning the block Krylov subspace. Moreover, DSBlockCG and BFDSCG are designed to make use of deflation in the block Krylov subspace. Therefore, we present two test cases in Figure 8.15 in which we generated the right-hand sides in such a manner that after a few steps deflation occurs. With this we want to show that DSBlockCG and BFDSCG can properly handle deflation and might even profit from removing (nearly) linearly dependent vectors.

For the following examples we use `WDStencil16` with the partial fraction expansion pfe_{Z16s18} and corresponding target relative residual norms. Moreover, for rsCG we use a restart length of 400.

In the first test setting that is displayed in the upper plots in Figure 8.15 we chose random right-hand sides except for the last column of B. The last column b_m of B was set to $b_m = v^{(15)}$ where m is the number of right-hand sides and $v^{(15)}$ is the 15-th Lanczos vector generated by the Lanczos process applied to A and b_1. In exact arithmetic exact deflation would occur while spanning $\mathcal{K}^{\square}_{15}(A, B)$ if b_m is computed in this way. For finite precision arithmetic we encounter inexact deflation for larger numbers of right-hand sides instead. As we can see for up to 4 right-hand sides DSBlockCG is faster than sCG and rsCG in this test setting. After that it again loses ground.

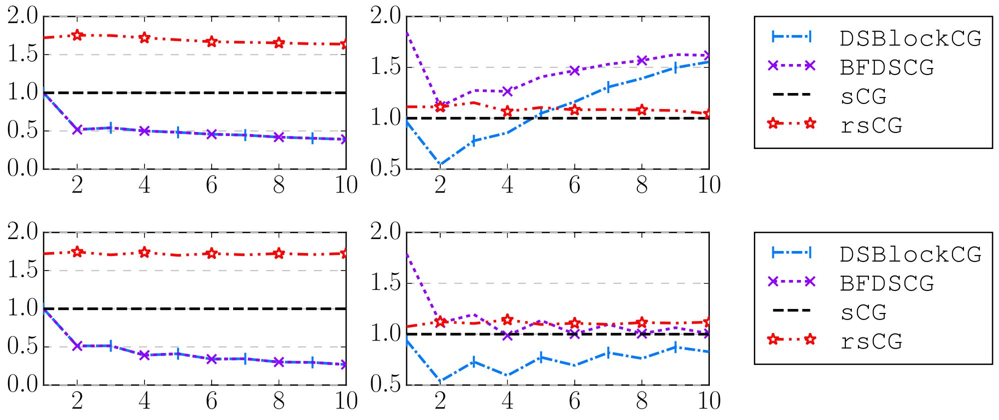


Figure 8.15: Comparison of some selected methods for the operator `WDStencil16` with 1 to 10 right-hand sides that are created such that after a few iteration steps one Lanczos vector becomes linearly dependent. Left: relative mvms, right: relative time.

In the lower plots in Figure 8.15 we again chose random right-hand sides but this time $\lfloor m/2 \rfloor$ columns of B were enforced to become linearly dependent. In detail, we computed $b_i = v^{(\lfloor 2\sqrt{i}*(i-1)+5\rfloor)}$ for $i = 2, 4, \ldots$ where $v^{(j)}$ is the j-th Lanczos vector generated by the Lanczos process applied to A and b_1. Again, we cannot expect exact deflation for larger numbers of right-hand sides. In this, admittedly unrealistic, setting DSBlockCG is faster than sCG and rsCG for all test up to 10 right-hand sides. BFDSCG gains from the deflation too and is about as fast as sCG and rsCG.

Test Simulating Larger Lattice Sizes

In lattice QCD, bigger lattices allow to adjust parameters close to physically relevant values, typically resulting in less well conditioned systems. Since we have chosen to implement all methods in a non-parallel version to have fair time comparisons we cannot increase the lattice size arbitrarily. However, we want to simulate less well conditioned systems on a 16^4 lattice. We do so by choosing a different mass parameter m_0 for the Wilson-Dirac operator `WDStencil16` and shift it such that the smallest eigenvalue of becomes 10^{-8}. This is consistent to what is done on large lattices, where smaller mass parameters are used, too.

We computed an n-point Padé approximation to $f(x) = x^{-1/4}$ with 14 poles which has a maximal error of less than 7×10^{-5} on the spectrum of the shifted operator. This was chosen since a higher precision approximation on an interval whose left end is 10^{-8} would need an unfeasible number of shifts. The iteration was stopped as soon as the relative residual norm fulfilled the condition (5.7) for 1×10^{-4}.

The results for these settings are shown in Figure 8.16. We can see that for 2 to 5 right-hand sides DSBlockCG performs slightly better than sCG before the costs for handling multiple right-hand sides starts to outweigh the time savings steming from a reduced number of matrix-vector multiplications. Since the time for sCG increases linearly in the number of right-hand sides one idea in this situation could be to always combine 3 or 4 of the m right-hand sides and solve them with DSBlockCG resulting in an overall gain as compared to sCG.

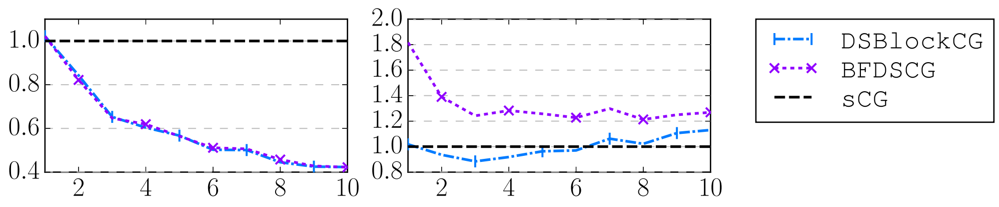


Figure 8.16: Comparison of some selected methods for the operator `WDStencil16` which we have shifted to have smallest ev 10^{-8} with 1 to 10 right-hand sides. Left: relative mvms, right: relative time.

8.3.4 Image Deconvolution - Prerequisites

For our tests with the operators PSF200 and PSF400 we use right-hand sides that stem from two sequences of a film.

We extracted the first sequence of 72 contiguous frames from the film “Big Buck Bunny” [Ble08] which is released under a Creative Commons (CC By 3.0) license[1]. While extracting we scaled and cropped the frames[2] so that we obtain images of the size 200×200 and 400×400. Then, we converted the two-dimensional RGB data into three vectors of size 40000 and 160000, respectively, by splitting the colour channels. Afterwards we blurred all extracted frames using A_{psf200} and A_{psf400}, respectively. We end up with two example right-hand sides $B_{1,200} \in \mathbb{R}^{40000\times 216}$ and $B_{1,400} \in \mathbb{R}^{160000\times 216}$.

For the second sequence we again chose 72 contiguous frames at a different position[3] which includes a fade-to/from-black section. The frames 23 to 44 are completely black and the fade-in/out takes about 10 frames each. We refer to these right-hand sides as $B_{2,200} \in \mathbb{R}^{40000\times 216}$ and $B_{2,400} \in \mathbb{R}^{160000\times 216}$.

When we perform our tests with PSF200 or PSF400 for k right-hand sides we will use $(B_{x,y})_{:,1:k}$ as the right-hand side block-vector. We use the 20 shifts σ_i with

$$\begin{aligned}\sigma_i \in \{&1.00\times10^{-2}, 1.65\times10^{-2}, 2.60\times10^{-2}, 4.04\times10^{-2}, 6.25\times10^{-2},\\ &9.81\times10^{-2}, 1.61\times10^{-1}, 2.90\times10^{-1}, 6.73\times10^{-1}, 2.12,\\ &3.24, 5.15, 6.07, 6.71, 7.29, 7.84, 8.39, 8.93, 9.46, 10.0\}.\end{aligned} \tag{8.2}$$

To help to determine the optimal regularisation parameter for the L-curve criterion we generated[4] the small shifts more densely. This heuristically resulted in distinctive L-curves.

In the QCD test cases we computed the target relative residual norm depending on the weight that corresponds to the shift of the system. Since we are not computing a matrix function we do not have weights and we stop the iterations at a fixed target relative residual norm of 10^{-5} for every shifted system. The restarted shifted CG algorithm cannot be applied in this situation because we need to compute individual solutions to all of the shifted systems

[1] `http://creativecommons.org/licenses/by/3.0`

[2] Using
```
ffmpeg -i big_buck_bunny_480p_h264.mov -r 24 -ss 00:05:06 -t 3
       -filter:v "scale=-1:200,crop=200:200:100:0" %03d.png
```
and the same with `"scale=-1:400,crop=400:400:200:0"` for the 400×400 images.

[3] Using `ffmpeg`[2] again with `-ss 00:07:40` as start time.

[4]
```
c = 20; lmin = -2; lmax = 1;
ls = linspace(-10,10,c); ls = ls.*abs(ls);
d = (pi/2+atan(ls))/pi;
sigma = ((1-d).*logspace(lmin,lmax,c)+(d).*linspace(10^lmin,10^lmax,c));
```

and not a single vector x as the result of a matrix function $x = f(A)b$.

8.3.5 Image Deconvolution - Methods

To determine which methods are most likely to perform well in our tests we applied our cost model to a test setting consisting of the operator `PSF400`, 26 right-hand sides and 20 shifts. As in the QCD case, we actually performed the computations to be able to compare the predictions of our cost models to measured times. For this setting we obtain from Section 8.1 that $c_A \approx 1384.5$ for a matrix-vector multiplication and $c_A^{\square} \approx 1384.5/4 \approx 346.1$ for a matrix-block-vector multiplication. The time for a vector operation is about 6.89×10^{-5} seconds.

We used the following assumptions to feed our cost models:

- The same number of iterations is needed to solve the systems belonging to different right-hand sides but the same shift.
- The number of iterations in the Single Seed Method, Seed-CG, and incremental eigCG reduces linearly along the right-hand sides to 50% for the last right-hand side.
- All block methods reduce the number of matrix-vector multiplications to 90% (counting a matrix-block-vector multiplication as 26 matrix-vector multiplications) of the number of matrix-vector multiplications that CG needs to solve all the systems.

The last point is due to the fact that for image deconvolution no high precision solutions are needed and therefore the number of iteration steps is quite small. As seen in the QCD test setting this results only in a minor reduction of iteration steps by block methods. In Table 8.12 we present our findings.

Again, our cost model estimates seem to be quite good in predicting the ranking of the methods with respect to their runtime. Only for methods whose runtime is roughly of the same magnitude our cost model fails to predict the right order as for example comparing CG and incremental eigCG. From Table 8.12 we can draw the conclusion that in our tests with `PSF200` and `PSF400` we only need to compare sCG, DSBlockCG, and BFDSCG.

8.3.6 Image Deconvolution - Results

Figure 8.17 displays the comparison of sCG, DSBlockCG, and BFDSCG for the operators `PSF200` and `PSF400` and the right-hand sides $B_{1,200}$ and $B_{1,400}$, respectively. We solved for $k = 1, 6, 11, \ldots, 216$ right-hand sides to generate

Table 8.12: Comparison of cost model estimates and time measurements for all the contestant methods. The estimated time is the product of the estimated vector operations from our cost model and the measured time per vector operation (6.89×10^{-5} seconds).

method	vector operations	estimated time	measured time	ratio
CG	7080892	487.87	632.38	0.77
sCG	1490333	102.68	104.88	0.98
SSM	5179214	356.85	380.83	0.94
Seed-CG	5025218	346.24	396.62	0.87
iEigCG	7782320	536.20	593.49	0.90
BlockCG	2183699	150.46	166.65	0.90
BCGAdQ	2559784	176.37	262.69	0.67
BCGrQ	2321291	159.94	259.50	0.62
DSBlockCG	1642574	113.17	141.68	0.80
BFDSCG	701043	48.30	69.08	0.70

the plots. In case of PSF200 both block methods can reduce the number of matrix-vector multiplications (or the equivalent number of matrix-block-vector multiplications) by up to a factor of 3 as compared to sCG. For the time measurement of this test setting we see that BFDSCG can make use of the notably better performance of a matrix-block-vector multiplication over a matrix-vector multiplication as we found out in Figure 8.4. This results in BFDSCG being faster than and needing only between 50% and 80% of sCG. For DSBlockCG the cost of handling multiple right-hand sides outweighs the reduction in matrix-vector multiplications so that sCG is faster.

The bottom two plots in Figure 8.17 show the same comparison for PSF400. There, the reduction in the number of matrix-vector multiplications is smaller than for PSF200. The outcome of this is that with respect to computing time for more than approximately 50 right-hand sides the tide turns in favour of sCG.

In Figure 8.18 we show some more details of the plot in Figure 8.17. There, we plotted the results for $k = 1, 2, \dots, 20$ right-hand sides. We see that already for about 10 right-hand sides the maximum performance gain as compared to sCG is achieved. The saw-tooth pattern that emerges in the time measurements in this plot is caused by the performance of the matrix-block-vector multiplication that we have discussed for Figure 8.4. Moreover, we can deduce

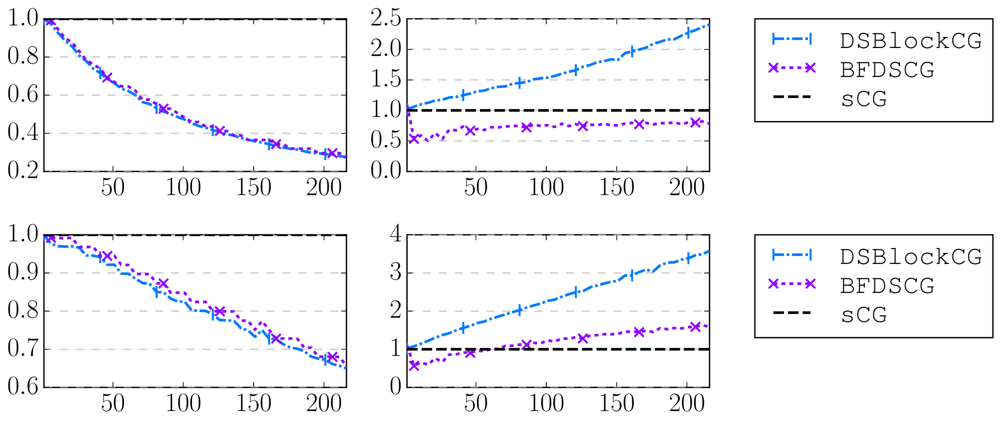


Figure 8.17: Comparison of sCG, DSBlockCG, and BFDSCG for the operator `PSF200` (top) and `PSF400` (bottom) with 1 to 216 right-hand sides from $B_{1,200}$ and $B_{1,400}$, respectively. The left plots display the relative number of matrix-vector multiplications and the right plots show the relative time normalised with respect to sCG in both cases.

that the time savings stem almost exclusively from the fast matrix-block-vector multiplication since the reduction in matrix-vector multiplications by the block methods is marginal. Here, BFDSCG seems to need more matrix-vector multiplications than DSBlockCG which is caused by the small number of iteration steps and the fact that BFDSCG can only stop after performing a complete matrix-block-vector multiplication. For every right-hand side the family of shifted systems is solved in about 40 steps of sCG.

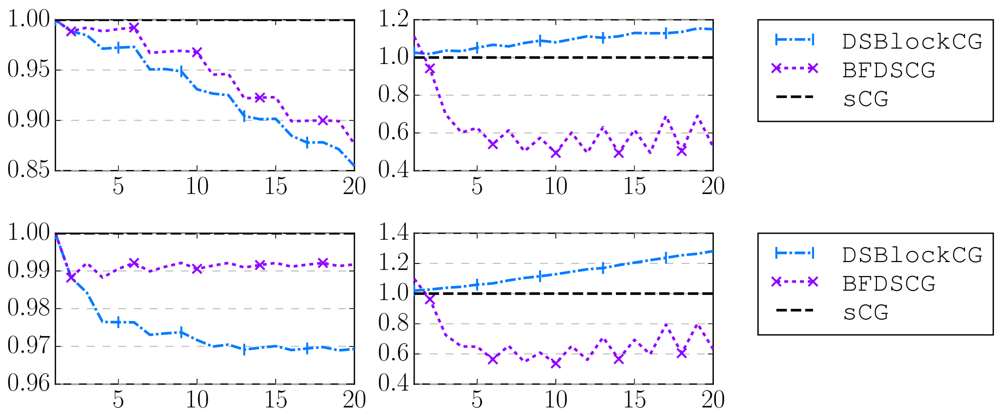


Figure 8.18: Detailed plot of the first 20 right-hand sides of the plots from Figure 8.17.

In Figure 8.19 we show the results of using $B_{2,200}$ and $B_{2,400}$ as right-hand sides. All other settings were the same as before. The black frames 23 to 44 result in the right-hand sides from 67 to 132 being zero. However, we cannot expect performance improvements over sCG there since sCG does not need to be applied to a zero right-hand side. On the other hand, these frames can be regarded as a stress test for our block methods since failing to deflate these would imply loosing ground to sCG or even failing to converge at all. Our intention of using $B_{2,200}$ and $B_{2,400}$ was to have an example where (inexact) deflation occurs naturally in the fade-in/out frames. But we did not observe any further (inexact) deflation than the removal of the black frames in the beginning of the iteration.

All in all, we see for the example in Figure 8.19 the same behaviour as in Figure 8.17 except that the gain in matrix-vector multiplications stagnates while adding those right-hand sides that get deflated.

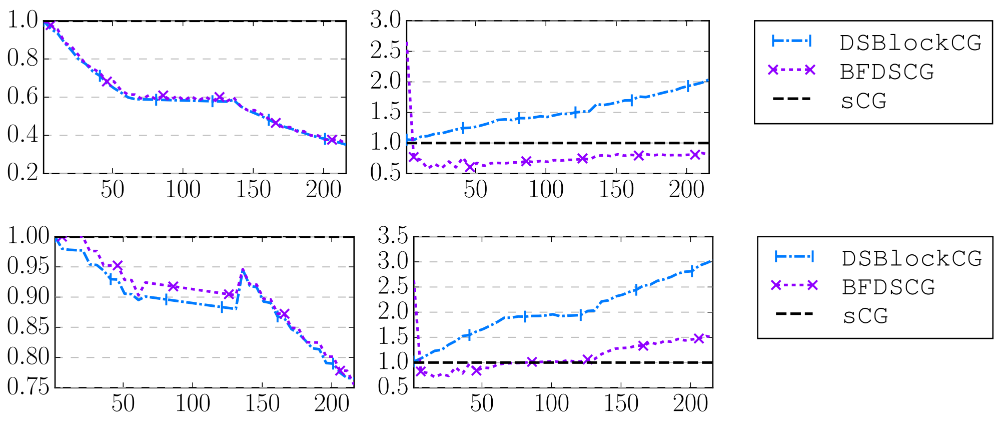


Figure 8.19: Comparison of sCG, DSBlockCG, and BFDSCG for the operator `PSF200` (top) and `PSF400` (bottom) with 1 to 216 right-hand sides from $B_{2,200}$ and $B_{2,400}$, respectively. The left plots display the relative number of matrix-vector multiplications and the right plots show the relative time normalised with respect to sCG in both cases.

Our final test for `PSF200` in Figure 8.20 shows how sCG, DSBlockCG, and BFDSCG compare depending on the number of shifts. For this setting we solved (1.1) for the first 20 columns of $B_{1,200}$. We used the first k shifts from (8.2) for $k = 1, \dots, 20$ and plotted the results from that against the number of matrix-vector multiplications and the measured time, respectively. Since the number of matrix-vector multiplications depends on the magnitude of the smallest shift, which in this test setting is always the same, we see no dependence on the number of shifts. The right plot in Figure 8.20 shows no significant dependence of the time for solving (1.1) depending on the number of shifts.

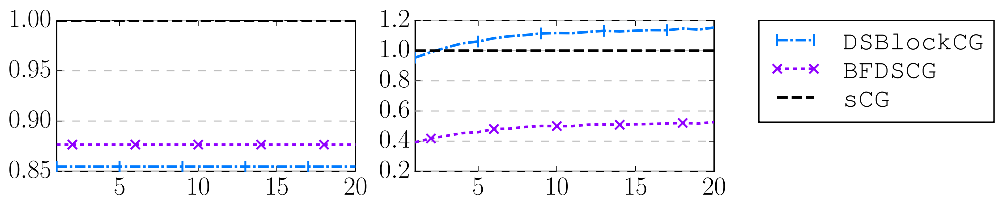


Figure 8.20: Comparison of sCG, DSBlockCG, and BFDSCG for the operator `PSF200` with 1 to 20 shifts for $B = (B_{1,200})_{:,1:20}$. The left plot displays the relative number of matrix-vector multiplications and the right plot shows the relative time normalised with respect to sCG in both cases.

As a conclusion of our tests with `PSF200` and `PSF400` we see that BFDSCG has the potential to yield considerable time improvements over sCG. Even though BFDSCG tends to be slower than sCG for a large number of right-hand sides one might pack a small number of right-hand sides into blocks and apply BFDSCG multiple times which is actually backed by the observations in Figure 8.18. Moreover, like in the QCD test cases we observed no naturally occurring deflation.

8.4 Comparison of Methods for Solving Block Systems

DSBlockCG and BFDSCG even though developed for shifted block systems can also be used as a non-shifted methods for block systems. Then they reduce to two block CG algorithms involving the deflation from [Ali+00]. Our goal here is to show that DSBlockCG and BFDSCG can compete with block methods like BCGrQ and outperform CG when applied to non-shifted block systems.

For this we use the four matrices `msc04515`, `Pres_Poisson`, `smt`, and `nd12k` from the Florida sparse matrix collection [DH11] that were presented in Table 8.1. We chose these matrices as examples, because they are hpd and have a relatively high number of non-zeros per column. For more detailed information on the matrices see [DH11]. For all matrices we computed solutions to random right-hand sides and compared our algorithm to BlockCG and BCGrQ.

In our tests we experienced that if inexact deflation happens early in the iteration the stability of the method suffers especially if a high precision as the target residual norm is requested. This behaviour was already described and discussed in [BPS11]. Thus, we have to balance the deflation tolerance for DSBlockCG and BFDSCG and the final required accuracy. We chose a deflation tolerance of 10^{-10} and stopped the iteration as soon as a relative residual norm of 10^{-8} was reached.

Figure 8.21 displays the results for these runs depending on the number of right-hand sides. Even though BlockCG needs slightly more iteration steps in the examples given it is significantly faster especially for those examples where matrix-vector multiplications are relatively cheap. In the test run with `nd12k`, where the time for matrix-vector multiplications gets more dominant, the gap between BlockCG and the other methods closes. For `smt` we could not include BlockCG in the test since it failed to converge for a larger number of right-hand sides. This is possibly related to the larger number of matrix-vector multiplications of BlockCG as compared to those block methods that involve a QR decomposition in the other tests. It seems as if the QR decomposition stabilises the iteration even when no deflation occurs.

Amongst the other block methods the number of matrix-vector multiplications is almost the same in all examples. But there is a noticeable difference in the total execution time for the more sparse matrices like `msc04515` and `Pres_Poisson`. In these matrices and for small numbers of right-hand sides DSBlockCG is faster than BCGrQ. In the tests with the matrix `msc04515` there is a turning point at 25 right-hand sides after which BlockCG solves the systems faster than DSBlockCG. When solving systems with the matrix `Pres_Poisson` DSBlockCG is faster than BlockCG for up to 12 right-hand sides. For the other two matrices DSBlockCG and BFDSCG are slower than BCGrQ.

In the examples presented in Figure 8.21 we encountered no deflation.

In the previous plots we did not include CG since it cannot compete with the block methods in these examples. To show how CG compares to the other methods we include Figure 8.22. This is the same test run for `msc04515` as in Figure 8.21. For the other tests CG behaves similar as in Figure 8.22.

Deflation is a situation in which DSBlockCG and BFDSCG should benefit and be able to reduce the computational time. Since there is a lack of naturally occurring situations in which deflation is needed we created special right-hand sides to show the impact of deflation on the time measurements. We artificially chose one right-hand side to be linearly dependent from the others after a few iteration steps. More specifically, we computed $b_m = v^{(15)}$ where m is the number of right-hand sides and $v^{(15)}$ is the 15-th Lanczos vector generated by the Lanczos process applied to A and b_1. In this way, in exact arithmetic we should encounter exact deflation while spanning $\mathscr{K}_{15}^{\square}(A, B)$. In finite precision,

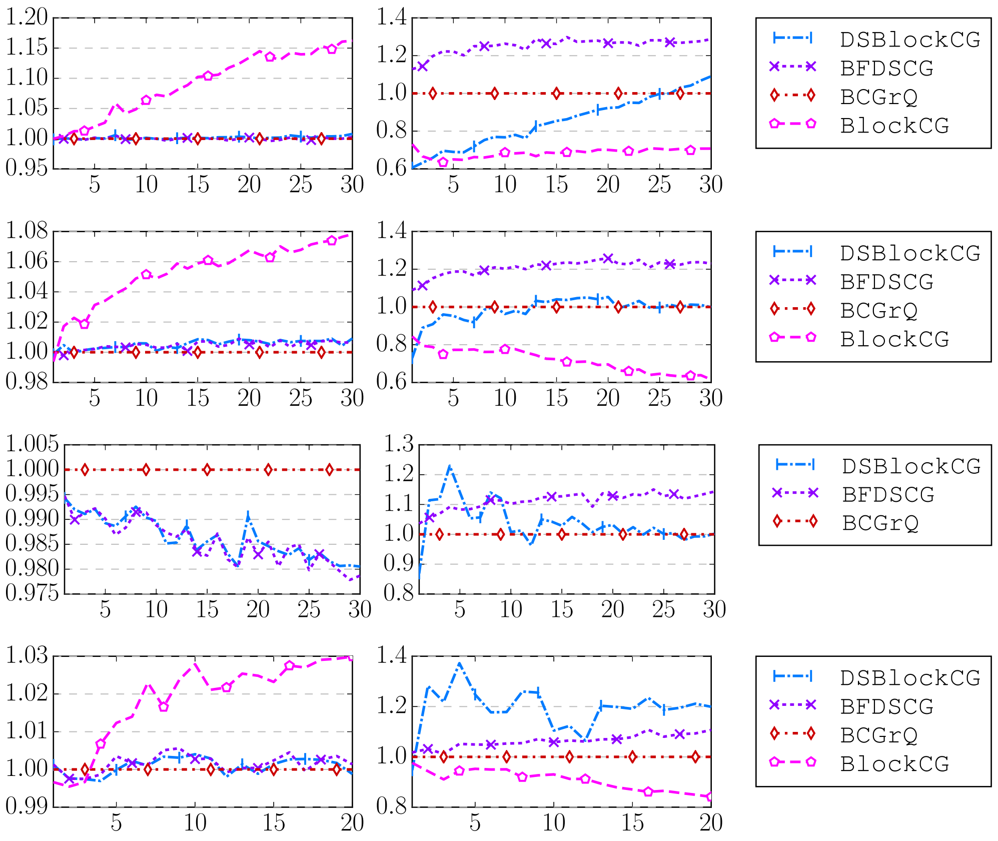


Figure 8.21: Comparison of block methods depending on the number of right-hand sides. The left plots display the number of matrix-vector multiplications normalised to BCGrQ and the right plots display the relative time as compared to BCGrQ. The matrices used for the examples are from top to bottom: `msc04515`, `Pres_Poisson`, `smt`, and `nd12k`.

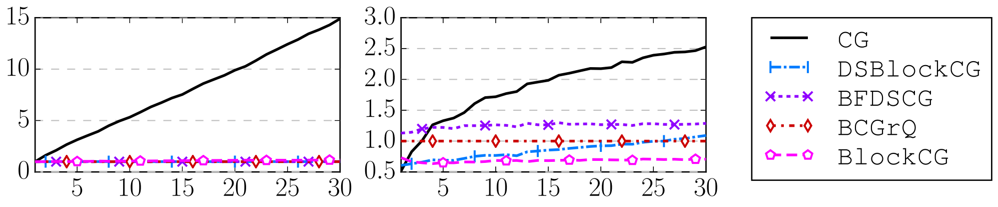


Figure 8.22: Comparison of block methods and CG. We show here the same test run as in the first row of Figure 8.21 but include CG this time.

however, we see this change to inexact deflation for larger numbers of right-hand sides.

In Figure 8.23 we used the same settings for the matrices `msc04515` and `nd12k` as described before for Figure 8.21. As expected our algorithms can benefit from deflating the linearly dependent vector. The reduction in the number of matrix-vector multiplications in the tests directly translates into improved timings. Hence, having a not too large number of right-hand sides and deflation early in the iteration leads to DSBlockCG and BFDSCG being faster than BCGrQ. Note that in these test BlockCG failed to converge and is therefore no alternative to the DSBlockCG, BFDSCG, and BCGrQ.

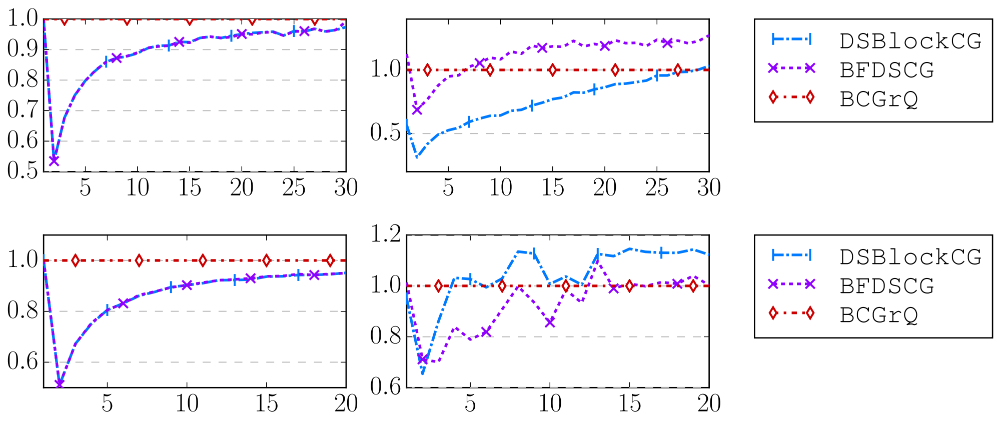


Figure 8.23: Comparison of block methods depending on the number of right-hand sides including deflation of one vector. The left plots display the number of matrix-vector multiplications normalised to BCGrQ and the right plots display the relative time as compared to BCGrQ. The matrices used for the examples are from `msc04515` (top) and `nd12k` (bottom).

In Figure 8.24 we want to display how the deflation tolerance for inexact deflation affects DSBlockCG and BFDSCG. We show two runs with `msc04515` there and we used the settings as described before. In the upper plot in Figure 8.24 we used 10^{-10} as deflation tolerance and in the lower 10^{-8}. For the 10^{-10} deflation tolerance (upper plot) in the DSBlockCG algorithm no inexact deflation occurred for 38 or more right-hand sides since the tolerance was not reached. In the BFDSCG algorithm the deflation tolerance was not reached any more starting with 32 right-hand sides. Hence, the nearly linearly dependent vector was kept and resulted in a slowed down convergence for DSBlockCG whereas BFDSCG failed to converge.

When adjusting the deflation tolerance to 10^{-8} (lower plot in Figure 8.24) both methods, DSBlockCG and BFDSCG, did apply inexact deflation and

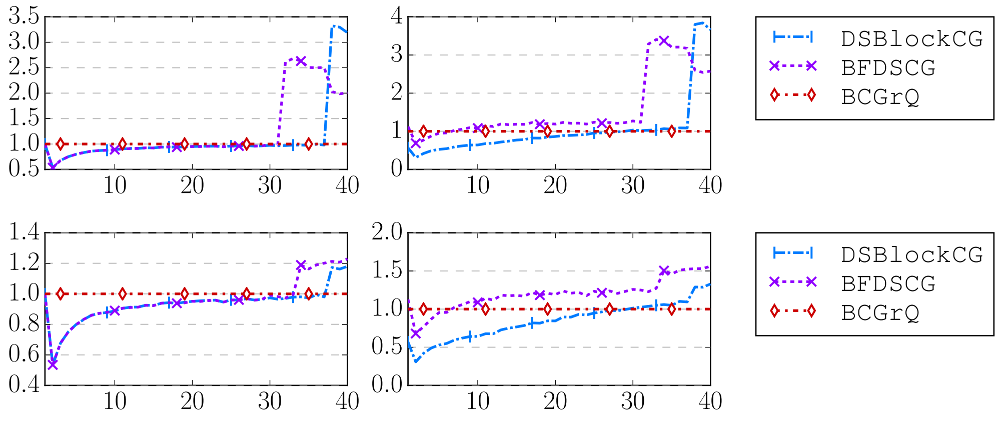


Figure 8.24: Comparison of deflation tolerance in DSBlockCG and BFDSCG. In the upper plot we used 10^{-10} as deflation tolerance and in the lower 10^{-8}.

converge. However, the number of matrix-vector multiplications still increased and with it the time.

8.5 Conclusion for the Numerical Results

In Section 8.2 we saw that the deflated block Lanczos-type processes from Chapter 7 provide bases that keep orthogonality well when deflation occurs. This is important for deflated block Krylov subspace methods that are based on these processes.

In Section 8.3 we applied our cost model for the tested algorithms. We found that it reflects the ranking of the different methods quite well.

From the examples for shifted block systems in Section 8.3 we can draw the conclusion that BFDSCG and DSBlockCG can be faster than all the other methods in some circumstances. Especially, when matrix-vector multiplications are costly as compared to vector operations and block-vector operations noticeably save computational time over vector operations then BFDSCG can outperform the other methods. DSBlockCG can be a good choice in situations where block-vector operations do not yield a larger performance gain. For the remaining examples sCG and—if the number of restarts can be kept low—rsCG remain the methods of choice for solving (1.1). In all cases, deflation, albeit occurring rarely, is beneficial for DSBlockCG and BFDSCG.

From the block system examples in Section 8.4 we can draw the conclusion that BFDSCG and DSBlockCG can be faster than BCGrQ in some cases, especially when deflation occurs. If no deflation occurs then BlockCG tends to be the fastest method to solve block systems. However, we found BlockCG to not be a reliable algorithm since even without deflation it sometimes fails to converge.

9 Conclusion and Outlook

Here, we summarise our findings and contributions. Afterwards we discuss some possible future work that can be based upon the work done in this thesis.

Contributions

The incentive for this work was the realisation of a lack of methods targeted at solving

$$(A + \sigma_i I)x_{i,j} = b_j, \quad i = 1, \dots, s, \quad j = 1, \dots, m. \tag{1.1}$$

for Hermitian positive definite matrices $(A + \sigma_i I) \in \mathbb{C}^{n \times n}$ efficiently and reliably. Primarily, this was driven by the need for solving this family of shifted linear systems in lattice QCD applications. But as we illustrated in Chapter 4 there are more applications requiring solutions for (1.1). In our search for the best way of solving (1.1) we have

- developed two novel shifted block Krylov subspace methods, namely the DSBlockCG and the BFDSCG algorithm, that are based on deflated Lanczos-type processes,
- worked out the algorithmic details of a restarted shifted CG method for computing $f(A)x$, and
- explored multiple other approaches like shifted methods and methods for multiple right-hand sides.

To tackle the problems of the common practice of simply counting matrix-vector multiplications as a method to compare Krylov subspace methods we developed a cost model that allows to estimate the runtime of an algorithm. This cost model differentiates between the costs of matrix-vector multiplications and matrix-block-vector multiplications and can thereby be used for algorithms using either approach. We applied this cost model to all the methods and used it to rank them by which method is most likely to show the shortest runtime. However, these estimates need some knowledge of the convergence behaviour, information on the matrix A, and a few educated guesses. Apart from that, it can be used as decision support for which method might be applied to solve (1.1) in a particular application and which method can be ruled out.

Our numerical examples showed that there is no best method in the sense that one method is always the fastest and thus first choice. In our observations the major performance gain is achieved by exploiting the shift invariance of Krylov subspaces. Block methods are typically faster than methods for single right-hand sides but in some situations the costs for handling multiple right-hand sides outweigh the benefits. We found that using block methods that involve matrix-block-vector multiplications instead of matrix-vector multiplications can yield a considerable gain especially for a large number of right-hand sides. The combination of handling multiple right-hand sides and multiple shifts at the same time has the potential to improve performance but is not guaranteed to be faster than shifted methods. Depending on the specific parameters of the application we found in our tests one of the methods sCG, rsCG, DSBlockCG, and BFDSCG to be the fastest to solve (1.1).

In our tests we never observed near linear dependency occurring naturally. In some of the tests in Chapter 8 we artificially introduced near linear dependency. These suggest that if there are real applications in which deflation occurs naturally then these would be opportunities for DSBlockCG and BFDSCG to shine. Moreover, when solving block systems DSBlockCG and BFDSCG can be alternatives for BlockCG which sometimes fails to converge even when no deflation occurs.

All in all, we have presented new methods that can—in certain situations—improve on existing methods for solving families of shifted linear systems with multiple right-hand sides.

Future Work

There are some paths that opened up in this thesis for being explored further in future work.

As a start it would be interesting to investigate if other restarting techniques could be used in rsCG. This might make the algorithm less prone to slowdowns caused by large numbers of restarts.

Especially for sparse matrices the block methods suffered from the matrix-block-vector multiplications not being much faster than matrix-vector multiplications. If other sparse matrix formats like the SELL-C-σ [Kre^{+}14] format could be employed then block methods could make up some ground to shifted (block) methods.

Another possible path of exploration which is of practical interest is how parallel implementations of these methods behave in terms of runtime. Depending on which operations benefit from parallelisation, the outcome of the tests could be quite different. But, if parallelisation only reweighs the ratio of the time for different operations then our cost model would stay valid and the new numbers could be fed back to it.

An algorithmically interesting question to answer is whether the rsCG algorithm could be extended to be able to solve systems with multiple right-hand sides. Clearly, this would immediately imply the question of how to address deflation properly.

Acknowledgements

I want to finish this thesis by expressing my gratitude to all the people that supported me. First, I like to thank my wife Doreen, my son Bjarne, and my daughter Finja without whom this work would not have been possible. Further, I thank my advisor Prof. Dr. Andreas Frommer for his advice and inspirations. Last but not least, I owe many thanks to my family, friends, officemates and colleagues that supported me along the way.

[..] universities are truly storehouses of knowledge: students arrive from school confident that they know nearly everything, and they leave years later certain that they know practically nothing. Where did the knowledge go in the meantime? Into the university, of course, where it is carefully dried and stored.

Terry Pratchett, Ian Steward, and Jack Cohen
in *The Science of Discworld*

List of Figures

List of Tables

List of Algorithms

Index

Bibliography

[AMW08] Abdou Abdel-Rehim, Ronald B. Morgan, and Walter Wilcox. “Seed methods for linear equations in lattice QCD problems with multiple right-hand sides”. In: *PoS (LAT2008) 038* (2008) (cit. on pp. 87, 89, 92, 93).
arXiv: `0901.3512 [hep-lat]`.

[AMW13] Abdou Abdel-Rehim, Ronald B. Morgan, and Walter Wilcox. “Improved seed methods for symmetric positive definite linear equations with multiple right-hand sides”. In: *Numer. Linear Algebra Appl.* 21 (2013), pp. 453–471 (cit. on pp. 89, 92–94).
DOI: `10.1002/nla.1892`.

[Afa$^+$08] Martin Afanasjew, Michael Eiermann, Oliver G. Ernst, and Stefan Güttel. “Implementation of a restarted Krylov subspace method for the evaluation of matrix functions”. In: *Linear Algebra and its Applications* 429.10 (2008). Special Issue in honor of Richard S. Varga, pp. 2293–2314 (cit. on p. 79).
DOI: `10.1016/j.laa.2008.06.029`.

[Ali$^+$00] José Ignacio Aliaga, Daniel L. Boley, Roland Freund, and Vicente Hernández. “A Lanczos-type method for multiple starting vectors”. In: *Math. Comp.* 69.232 (2000), pp. 1577–1601 (cit. on pp. 111, 131, 134, 135, 139, 141, 147, 148, 150, 216).
DOI: `10.1090/S0025-5718-99-01163-1`.

[ABH94] José Ignacio Aliaga, Daniel L. Boley, and Vicente Hernández. “A block clustered Lanczos algorithm”. Presentation at the workshop on ”Numerical Linear Algebra with Applications”, Oberwolfach, Germany. 1994 (cit. on p. 131).

[Ame$^+$99] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L’Excellent, and Jacko Koster. *A fully asynchronous multifrontal solver using distributed dynamic scheduling.* Tech. rep. RAL-TR-1999-059. Rutherford Appleton Laboratory, 1999 (cit. on p. 11).
DOI: `10.1137/S0895479899358194`.

[ADR92] Mario Arioli, Iain S. Duff, and Daniel Ruiz. “Stopping criteria for iterative solvers”. In: *SIAM J. Matrix Anal. Appl.* 13.1 (1992), pp. 138–144 (cit. on p. 12).
DOI: `10.1137/0613012`.

[BG96] George A. Baker and Peter R. Graves-Morris. *Padé Approximants.* 2nd. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1996 (cit. on pp. 39, 41, 42).
DOI: `10.1017/CBO9780511530074`.

[BPS11] Gianluca Barbella, Federico Perotti, and Valeria Simoncini. “Block Krylov subspace methods for the computation of structural response to turbulent wind”. In: *Comput. Methods Appl. Mech. Engrg.* 200.23–24 (2011), pp. 2067–2082 (cit. on p. 217).
DOI: 10.1016/j.cma.2011.02.017.

[BP06] Mario Bertero and Michele Piana. “Inverse problems in biomedical imaging: Modeling and methods of solution”. In: *Complex Systems in Biomedicine*. Ed. by Alfio Quarteroni, Luca Formaggia, and Alessandro Veneziani. Springer Milan, 2006, pp. 1–33 (cit. on p. 58).
DOI: 10.1007/88-470-0396-2_1.

[Bir08] Sebastian Birk. “Reusing information in relaxed inexact Krylov subspace methods”. Diploma thesis. Bergische Universität Wuppertal, 2008 (cit. on p. 3).

[BF14] Sebastian Birk and Andreas Frommer. “A deflated conjugate gradient method for multiple right hand sides and multiple shifts”. In: *Numerical Algorithms* 67.3 (2014), pp. 507–529 (cit. on pp. 3, 148, 166, 168, 169, 171, 183).
DOI: 10.1007/s11075-013-9805-9.

[Ble08] Blender Foundation. *Big Buck Bunny*. (C) Copyright 2008, Blender Foundation, www.blender.org, www.bigbuckbunny.org. 2008 (cit. on p. 211).
URL: http://download.blender.org/peach/bigbuckbunny_movies/big_buck_bunny_480p_h264.mov (visited on 2014-10-10).

[Bro+97] Richard C. Brower, Taras Ivanenko, Andrea Ruben Levi, and Kostas N. Orginos. “Chronological inversion method for the Dirac matrix in hybrid Monte Carlo”. In: *Nucl. Phys. B* 484.1–2 (1997), pp. 353–374 (cit. on p. 88).
DOI: 10.1016/S0550-3213(96)00579-2.

[Bro97] Charles George Broyden. “A breakdown of the block CG method”. In: *Optim. Methods Softw.* 7.1 (1997), pp. 41–55 (cit. on p. 119).
DOI: 10.1080/10556789608805643.

[BG65] Peter A. Businger and Gene H. Golub. “Linear least squares solutions by householder transformations”. In: *Numer. Math.* 7 (1965) (cit. on pp. 158, 159).
DOI: 10.1007/BF01436084.

[Cha87] Tony F. Chan. “Rank revealing QR factorizations”. In: *Linear Algebra Appl.* 88–89 (1987), pp. 67–82 (cit. on pp. 157, 158).
DOI: 10.1016/0024-3795(87)90103-0.

[CH94] Tony F. Chan and Per Christian Hansen. “Low-rank revealing QR factorizations”. In: *Numer. Linear Algebra Appl.* 1.1 (1994), pp. 33–44 (cit. on p. 158).
DOI: 10.1002/nla.1680010105.

[CW97] Tony F. Chan and Justin W.L. Wan. “Analysis of projection methods for solving linear systems with multiple right-hand sides”. In: *SIAM J. Sci. Comput.* 18.6 (1997), pp. 1698–1721 (cit. on pp. 87, 89, 92, 120).
DOI: 10.1137/S1064827594273067.

[Che82] Elliott W. Cheney. *Introduction to Approximation Theory*. AMS Chelsea Publishing Series. AMS Chelsea Pub., 1982 (cit. on pp. 38, 43).

[CC04] Ole Christensen and Khadija L. Christensen. *Approximation Theory: From Taylor Polynomials to Wavelets*. Applied and Numerical Harmonic Analysis. Birkhäuser Boston, 2004 (cit. on p. 31).

[Cla06] Michael A. Clark. “The Rational Hybrid Monte Carlo algorithm”. In: *PoS (LAT2006) 004* (2006) (cit. on p. 57).
arXiv: `hep-lat/0610048`.

[CD74] Jane K. Cullum and William E. Donath. “A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, real symmetric matrices”. In: *Decision and Control including the 13th Symposium on Adaptive Processes, 1974 IEEE Conference on.* Vol. 13. 1974, pp. 505–509 (cit. on p. 106).
DOI: `10.1109/CDC.1974.270490`.

[DMW08] Dean Darnell, Ronald B. Morgan, and Walter Wilcox. “Deflated GMRES for systems with multiple shifts and multiple right-hand sides”. In: *Linear Algebra Appl.* 429.10 (2008). Special Issue in honor of Richard S. Varga, pp. 2415–2434 (cit. on pp. 98, 130).
DOI: `10.1016/j.laa.2008.04.019`.

[DD95] Timothy A. Davis and Iain S. Duff. *A combined unifrontal/multifrontal method for unsymmetric sparse matrices.* Tech. rep. TR-95-020. Computer and Information Sciences Department, University of Florida, Gainesville, 1995 (cit. on p. 11).

[DH11] Timothy A. Davis and Yifan Hu. “The University of Florida Sparse Matrix Collection”. In: *ACM Trans. Math. Softw.* 38.1 (2011), 1:1–1:25 (cit. on pp. 185, 216).
DOI: `10.1145/2049662.2049663`.

[Del$^+$07a] Luigi Del Debbio, Leonardo Giusti, Martin Lüscher, Roberto Petronzio, and Nazario Tantalo. “QCD with light Wilson quarks on fine lattices (i): First experiences and physics results”. In: *JHEP* 0702:056,2007 (2007) (cit. on p. 185).
DOI: `10.1088/1126-6708/2007/02/056`.
arXiv: `hep-lat/0610059`.

[Del$^+$07b] Luigi Del Debbio, Leonardo Giusti, Martin Lüscher, Roberto Petronzio, and Nazario Tantalo. “QCD with light Wilson quarks on fine lattices (ii): DD-HMC simulations and data analysis”. In: *JHEP* 0702:082,2007 (2007) (cit. on p. 185).
DOI: `10.1088/1126-6708/2007/02/082`.
arXiv: `hep-lat/0701009`.

[Dem97] James W. Demmel. *Applied Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, 1997 (cit. on p. 22).
DOI: `10.1137/1.9781611971446`.

[Dem$^+$99] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. “A supernodal approach to sparse partial pivoting”. In: *SIAM J. Matrix Anal. Appl.* 20.3 (1999). www.netlib.org, pp. 720–755 (cit. on p. 11).
DOI: `10.1137/S0895479895291765`.

[DP03] Inderjit S. Dhillon and Beresford N. Parlett. “Orthogonal eigenvectors and relative gaps”. In: *SIAM J. Matrix Anal. Appl.* 25.3 (2003), pp. 858–899 (cit. on p. 77).
DOI: `10.1137/S0895479800370111`.

[Dub01] Augustin A. Dubrulle. “Retooling the method of block conjugate gradients”. In: *Electron. Trans. Numer. Anal.* 12 (2001), 216–233 (electronic) (cit. on pp. 88, 107, 114, 120, 123, 124, 126).

[Duf98] Iain S. Duff. *Direct Methods.* Tech. rep. RAL-TR-1998-054. Rutherford Appleton Laboratory, 1998 (cit. on p. 11).

[Dul$^+$12] George S. Dulikravich, Brian H. Dennis, George A. Baker, Stephen R. Kennon, Helcio R. B. Orlande, and Marcelo J. Colaco. “Inverse problems in aerodynamics, heat transfer, elasticity and materials design”. In: *Int’l J. of Aeronautical & Space Sci.* 13.4 (2012), pp. 405–420 (cit. on p. 58).
DOI: 10.5139/IJASS.2012.13.4.405.

[EEG11] Michael Eiermann, Oliver G. Ernst, and Stefan Güttel. “Deflated restarting for matrix functions”. In: *SIAM J. Matrix Anal. Appl.* 32.2 (2011), pp. 621–641 (cit. on p. 84).
DOI: 10.1137/090774665.

[EES00] Michael Eiermann, Oliver G. Ernst, and Olaf Schneider. “Analysis of acceleration strategies for restarted minimal residual methods”. In: *J. Comput. Appl. Math.* 123.1-2 (2000), pp. 261–292 (cit. on p. 96).
DOI: 10.1016/S0377-0427(00)00398-8.

[EJS03] Ahmed El Guennouni, Khalide Jbilou, and Hassane Sadok. “A block version of BiCGSTAB for linear systems with multiple right-hand sides”. In: *Electron. Trans. Numer. Anal.* 16 (2003), pp. 129–142 (cit. on p. 107).

[EG00] Jocelyne Erhel and Frédéric Guyomarc’h. “An augmented conjugate gradient method for solving consecutive symmetric positive definite linear systems”. In: *SIAM J. Matrix Anal. Appl.* 21.4 (2000), pp. 1279–1299 (cit. on pp. 97, 98).
DOI: 10.1137/S0895479897330194.

[Esh$^+$02] Jasper van den Eshof, Andreas Frommer, Thomas Lippert, Klaus Schilling, and Henk A. van der Vorst. “Numerical methods for the QCD overlap operator I: Sign-function and error bounds”. In: *Computer Physics Communications* 146 (2002), pp. 203–224 (cit. on pp. 46, 48).
DOI: 10.1016/S0010-4655(02)00455-1.

[ES04] Jasper van den Eshof and Gerard L.G. Sleijpen. “Accurate conjugate gradient methods for families of shifted systems”. In: *Appl. Numer. Math.* 49.1 (2004), pp. 17–37 (cit. on p. 77).
DOI: 10.1016/j.apnum.2003.11.010.

[FF95] Peter Feldmann and Roland Freund. “Efficient linear circuit analysis by Padé approximation via the Lanczos process”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 14.5 (1995), pp. 639–649 (cit. on p. 131).
DOI: 10.1109/43.384428.

[FFF97] Peter Fiebach, Andreas Frommer, and Roland Freund. “Variants of the Block-QMR method and applications in quantum chromodynamics”. In: *15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics.* Vol. 3. 1997, pp. 491–496 (cit. on p. 130).

[FF94] Roland Freund and Peter Feldmann. "Efficient circuit analysis by Padé approximation via the Lanczos process". Presentation at the workshop on "Numerical Linear Algebra with Applications", Oberwolfach, Germany. 1994 (cit. on p. 131).

[FM97] Roland Freund and Manish Malhotra. "A block QMR algorithm for non-hermitian linear systems with multiple right-hand sides". In: *Linear Algebra Appl.* 254 (1997), pp. 119–157 (cit. on pp. 107, 130, 131).
DOI: `10.1016/S0024-3795(96)00529-0`.

[FN91] Roland Freund and Noël M. Nachtigal. "QMR: A quasi-minimal residual method for non-Hermitian linear systems". In: *Numerische Mathematik* 60.1 (1991), pp. 315–339 (cit. on p. 29).
DOI: `10.1007/BF01385726`.

[Fro03] Andreas Frommer. "BiCGStab(ℓ) for families of shifted linear systems". In: *Computing* 70.2 (2003), pp. 87–109 (cit. on p. 69).
DOI: `10.1007/s00607-003-1472-6`.

[FG98] Andreas Frommer and Uwe Glässner. "Restarted GMRES for shifted linear systems". In: *SIAM J. Sci. Comput.* 19.1 (1998), pp. 15–26 (cit. on pp. 68, 69).
DOI: `10.1137/S1064827596304563`.

[Fro$^+$13a] Andreas Frommer, Karsten Kahl, Stefan Krieg, Björn Leder, and Matthias Rottmann. "Adaptive aggregation based domain decomposition multigrid for the lattice Wilson Dirac operator". In: (2013) (cit. on pp. 51, 54, 185).
arXiv: `1303.1377 [hep-lat]`.

[Fro$^+$13b] Andreas Frommer, Karsten Kahl, Thomas Lippert, and Hannah Rittich. "2-norm error bounds and estimates for Lanczos approximations to linear systems and rational matrix functions". In: *SIAM J. Matrix Anal. Appl.* 34.3 (2013), pp. 1046–1065 (cit. on p. 13).
DOI: `10.1137/110859749`.

[FM99] Andreas Frommer and Peter Maass. "Fast CG-based methods for Tikhonov-Phillips regularization". In: *SIAM J. Sci. Comput.* 20.5 (1999), 1831–1850 (electronic) (cit. on pp. 71, 75).
DOI: `10.1137/S1064827596313310`.

[FS08] Andreas Frommer and Valeria Simoncini. "Matrix functions". In: *Model Order Reduction: Theory, Research Aspects and Applications.* Ed. by Wil H. A. Schilders, Henk A. van der Vorst, and Joost Rommes. Vol. 13. Mathematics in Industry. Heidelberg: Springer Berlin Heidelberg, 2008, pp. 275–303 (cit. on pp. 31, 44, 48).
DOI: `10.1007/978-3-540-78841-6_13`.

[Fut$^+$13] Yasunori Futamura, Tetsuya Sakurai, Shinnosuke Furuya, and Jun-Ichi Iwata. "Efficient algorithm for linear systems arising in solutions of eigenproblems and its application to electronic-structure calculations". In: *High Performance Computing for Computational Science - VECPAR 2012.* Ed. by Michel Daydé, Osni Marques, and Kengo Nakajima. Vol. 7851. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 226–235 (cit. on pp. 130, 131, 180).
DOI: `10.1007/978-3-642-38718-0_23`.

[GL10] Christof Gattringer and Christian B. Lang. *Quantum Chromodynamics on the Lattice: An Introductory Presentation.* Lecture Notes in Physics. Springer, 2010 (cit. on pp. 51, 57).

[Gau$^+$13] André Gaul, Martin H. Gutknecht, Jörg Liesen, and Reinhard Nabben. “A framework for deflated and augmented Krylov subspace methods”. In: *SIAM J. Matrix Anal. Appl.* 34.2 (2013), pp. 495–518 (cit. on p. 97).
DOI: 10.1137/110820713.

[GSZ11] Martin B. van Gijzen, Gerard L.G. Sleijpen, and Jens-Peter M. Zemke. *Flexible and Multi-shift Induced Dimension Reduction Algorithms for Solving Large Sparse Linear Systems.* Tech. rep. 11-06. Delft University of Technology, Department of Applied Mathematical Analysis, 2011 (cit. on p. 69).

[GW82] Paul H. Ginsparg and Kenneth G. Wilson. “A remnant of chiral symmetry on the lattice”. In: *Phys. Rev. D* 25 (10 1982), pp. 2649–2657 (cit. on p. 55).
DOI: 10.1103/PhysRevD.25.2649.

[GRT06] Luc Giraud, Daniel Ruiz, and Ahmed Touhami. “A comparative study of iterative solvers exploiting spectral information for spd systems”. In: *SIAM J. Sci. Comput.* 27.5 (2006), pp. 1760–1786 (cit. on p. 98).
DOI: 10.1137/040608301.

[GL96] Gene H. Golub and Charles F. van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1996 (cit. on pp. 8–11, 13, 14, 17, 32, 157, 158).

[GM97] Gene H. Golub and Gerard Meurant. “Matrices, moments and quadrature. II. How to compute the norm of the error in iterative methods”. In: *BIT Numerical Mathematics* 37 (3 1997), pp. 687–705 (cit. on p. 12).
DOI: 10.1007/BF02510247.

[GU77] Gene H. Golub and Richard Underwood. “The block Lanczos method for computing eigenvalues”. In: *Proceedings of the Symposium on Mathematical Software, University of Wisconsin.* Ed. by John Rischard Rice. (Also in: Milestones in Matrix Computation: The selected works of Gene H. Golub, 2007, Raymond Chan, Chen Greif, and Dianne O’Leary). 1977, pp. 361–377 (cit. on pp. 106, 110, 113).

[GG08] Kazushige Goto and Robert van de Geijn. “High-performance implementation of the level-3 BLAS”. In: *ACM Trans. Math. Softw.* 35.1 (2008), 4:1–4:14 (cit. on p. 88).
DOI: 10.1145/1377603.1377607.

[Gre97] Anne Greenbaum. *Iterative Methods for Solving Linear Systems.* SIAM Frontiers in Applied Mathematics, 1997 (cit. on pp. 11, 14, 21, 23, 24, 29, 115).

[GPS96] Anne Greenbaum, Vlastimil Pták, and Zdeněk Strakoš. “Any nonincreasing convergence curve is possible for GMRES”. In: *SIAM J. Matrix Anal. Appl.* 17.3 (1996), pp. 465–469 (cit. on p. 29).
DOI: 10.1137/S0895479894275030.

[GE96] Ming Gu and Stanley C. Eisenstat. “Efficient algorithms for computing a strong rank-revealing QR factorization”. In: *SIAM J. Sci. Comput.* 17.4 (1996), pp. 848–869 (cit. on p. 158).
DOI: 10.1137/0917055.

[G$^+$10] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3.* http://eigen.tuxfamily.org. 2010 (cit. on p. 184).
URL: `http://eigen.tuxfamily.org`.

[Gut06] Martin H. Gutknecht. "Block Krylov space methods for linear systems with multiple right-hand sides: An introduction". In: *Modern Mathematical Models, Methods and Algorithms for Real World Systems.* Ed. by Abul Hassan Siddiqi, Iain S. Duff, and Ole Christensen. Anamaya Publishers, New Delhi, India, 2006, pp. 420–447 (cit. on pp. 107, 113).

[GS08] Martin H. Gutknecht and Thomas Schmelzer. "Updating the QR decomposition of block tridiagonal and block Hessenberg matrices". In: *Appl. Numer. Math.* 58.6 (2008), pp. 871–883 (cit. on p. 108).
DOI: `10.1016/j.apnum.2007.04.010`.

[GS09] Martin H. Gutknecht and Thomas Schmelzer. "The block grade of a block Krylov space". In: *Linear Algebra Appl.* 430.1 (2009), pp. 174–185 (cit. on pp. 108, 109).
DOI: `10.1016/j.laa.2008.07.008`.

[GS00] Martin H. Gutknecht and Zdeněk Strakoš. "Accuracy of two three-term and three two-term recurrences for Krylov space solvers". In: *SIAM J. Matrix Anal. Appl.* 22.1 (2000), pp. 213–229 (cit. on pp. 22, 77).
DOI: `10.1137/S0895479897331862`.

[Han00] Per Christian Hansen. "The L-curve and its use in the numerical treatment of inverse problems". In: *Computational Inverse Problems in Electrocardiology (Advances in Computational Bioengineering).* Ed. by P. Johnston. WIT Press, 2000, pp. 119–142 (cit. on pp. 59, 63).

[HS52] Magnus R. Hestenes and Eduard L. Stiefel. "Methods of conjugate gradients for solving linear systems". In: *J. Res. Natl. Bur. Stand.* 49.6 (1952), pp. 409–436 (cit. on p. 21).

[Hig08] Nicholas J. Higham. *Functions of Matrices: Theory and Computation.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2008 (cit. on pp. 31–38).
DOI: `10.1137/1.9780898717778`.

[Hig$^+$04] Nicholas J. Higham, D. Steven Mackey, Niloufer Mackey, and Françoise Tisseur. "Computing the polar decomposition and the matrix sign decomposition in matrix groups". In: *SIAM J. Matrix Anal. Appl.* 25.4 (2004), pp. 1178–1192 (cit. on p. 37).
DOI: `10.1137/S0895479803426644`.

[HP92] YooPyo Hong and Ching-Tsuan Pan. "Rank-revealing QR factorizations and the singular value decomposition". In: *Math. Comp.* 58.197 (1992), pp. 213–232 (cit. on p. 158).
DOI: `10.2307/2153029`.

[HJ91] Roger A. Horn and Charles R. Johnson. *Topics in Matrix Analysis.* Cambridge University Press, 1991 (cit. on pp. 31, 32, 34).
DOI: `10.1017/CBO9780511840371`.

[IEE08] IEEE. *IEEE Standard for Floating-Point Arithmetic.* IEEE Computer Society, 2008 (cit. on p. 40).
DOI: `10.1109/IEEESTD.2008.4610935`.

[Isa98] Victor Isakov. *Inverse Problems for Partial Differential Equations.* Applied Mathematical Sciences (Springer-Verlag), Vol 127 Bd. 127. Springer, 1998 (cit. on pp. 51, 58, 59).
DOI: `10.1007/0-387-32183-7`.

[Jeg96] Beat Jegerlehner. *Krylov space solvers for shifted linear systems.* Tech. rep. IUHET-353. Indiana University, Department of Physics, 1996 (cit. on p. 69).
arXiv: `hep-lat/9612014`.

[KLL98] Bo Kågström, Per Ling, and Charles F. van Loan. "GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark". In: *ACM Trans. Math. Softw.* 24.3 (1998), pp. 268–302 (cit. on p. 88).
DOI: `10.1145/292395.292412`.

[Kah09] Karsten Kahl. "Adaptive algebraic multigrid for lattice QCD computations". http://d-nb.info/1000531767. PhD thesis. Bergische Universität Wuppertal, 2009 (cit. on p. 51).

[KR12] Karsten Kahl and Hannah Rittich. *The Deflated Conjugate Gradient Method: Convergence, Perturbation and Accuracy.* 2012 (cit. on p. 98).
arXiv: `1209.1963 [math.NA]`.

[Ken05] Anthony D. Kennedy. "Fast evaluation of Zolotarev coefficients". In: *QCD and Numerical Analysis III.* Ed. by Artan Boriçi, Andreas Frommer, Bálint Joó, Anthony D. Kennedy, and Brian Pendleton. Vol. 47. Lecture Notes in Computational Science and Engineering. Springer Berlin Heidelberg, 2005, pp. 169–189 (cit. on p. 45).
DOI: `10.1007/3-540-28504-0_16`.

[Ken06] Anthony D. Kennedy. *Algorithms for Dynamical Fermions.* Tech. rep. School of Physics, University of Edinburgh, 2006 (cit. on p. 57).
arXiv: `hep-lat/0607038`.

[Kre$^+$14] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units". In: *SIAM J. Sci. Comput.* 36.5 (2014), pp. C401–C423 (cit. on p. 224).
DOI: `10.1137/130930352`.

[Kro81] Leopold Kronecker. "Zur Theorie der Elimination einer Variablen aus zwei algebraischen Gleichungen". In: *Monatsber. Königl. Preuss. Akad. Wiss. Berlin* (1881), pp. 535–600 (cit. on p. 41).

[Lan03] Julien Langou. "Solving large linear systems with multiple right-hand sides". PhD thesis. Institut National des Sciences Appliquées de Toulouse, 2003 (cit. on p. 92).

[Lew95] Leroy Anthony Drummond Lewis. "Solution of general linear systems of equations using block Krylov based iterative methods on distributed computing environments". CERFACS Technical Report TH/PA/95/40. PhD thesis. Institut National Polytechnique de Toulouse, 1995 (cit. on p. 120).

[Loh06] Damian Loher. "Reliable nonsymmetric block Lanczos algorithms". PhD thesis. ETH Zürich, Department of Mathematics, 2006 (cit. on pp. 107, 111).
DOI: `10.3929/ethz-a-005143390`.

[Lüs98] Martin Lüscher. “Exact chiral symmetry on the lattice and the Ginsparg-Wilson relation”. In: *Physics Letters B* 428.3–4 (1998), pp. 342–345 (cit. on p. 56).
DOI: `10.1016/S0370-2693(98)00423-7`.
arXiv: `hep-lat/9802011`.

[Mor05] Ronald B. Morgan. “Restarted block-GMRES with deflation of eigenvalues”. In: *Appl. Numer. Math.* 54.2 (2005). 6th IMACS International Symposium on Iterative Methods in Scientific Computing, pp. 222–236 (cit. on pp. 98, 107).
DOI: `10.1016/j.apnum.2004.09.028`.

[MRD92] Christopher W. Murray, Stephen C. Racine, and Ernest R. Davidson. “Improved algorithms for the lowest few eigenvalues and associated eigenvectors of large matrices”. In: *J. Comput. Phys.* 103 (1992), pp. 382–389 (cit. on p. 100).
DOI: `10.1016/0021-9991(92)90409-R`.

[Neu98] Herbert Neuberger. “A practical implementation of the overlap Dirac operator”. In: *Phys. Rev. Lett.* 81.19 (1998), pp. 4060–4062 (cit. on p. 56).
DOI: `10.1103/PhysRevLett.81.4060`.

[Neu00] Herbert Neuberger. “The overlap Dirac operator”. In: *Numerical Challenges in Lattice Quantum Chromodynamics.* Ed. by Andreas Frommer, Thomas Lippert, Björn Medeke, and Klaus Schilling. Vol. 15. Lecture Notes in Computational Science and Engineering. Springer Berlin Heidelberg, 2000, pp. 1–17 (cit. on p. 56).
DOI: `10.1007/978-3-642-58333-9_1`.
arXiv: `hep-lat/9910040`.

[NY95] Andy A. Nikishin and Alex Yu. Yeremin. “Variable block CG algorithms for solving large sparse symmetric positive definite linear systems on parallel computers, I: General iterative scheme”. In: *SIAM J. Matrix Anal. Appl.* 16.4 (1995), pp. 1135–1153 (cit. on pp. 117, 119, 120).
DOI: `10.1137/S0895479893247679`.

[NY03] Andy A. Nikishin and Alex Yu. Yeremin. “An automatic procedure for updating the block size in the block conjugate gradient method for solving linear systems”. In: *J. Math. Sci.* 114 (6 2003), pp. 1844–1853 (cit. on pp. 119, 120).
DOI: `10.1023/A:1022462721147`.

[OLe80] Dianne P. O’Leary. “The block conjugate gradient algorithm and related methods”. In: *Linear Algebra Appl.* 29 (1980), pp. 293–322 (cit. on pp. 88, 106, 107, 115–117, 119).
DOI: `10.1016/0024-3795(80)90247-5`.

[PPV95] Chris C. Paige, Beresford N. Parlett, and Henk A. van der Vorst. “Approximate solutions and eigenvalue bounds from Krylov subspaces”. In: *Numer. Linear Algebra Appl.* 2.2 (1995), pp. 115–133 (cit. on pp. 68, 166).
DOI: `10.1002/nla.1680020205`.

[PS90] Manolis Papadrakakis and S. Smerou. “A new implementation of the Lanczos method in linear problems”. In: *International Journal for Numerical Methods in Engineering* 29.1 (1990), pp. 141–159 (cit. on p. 89).
DOI: `10.1002/nme.1620290110`.

[PS79] Beresford N. Parlett and Dana S. Scott. “The Lanczos algorithm with selective orthogonalization”. In: *Math. Comp.* 33.145 (1979), pp. 217–238 (cit. on p. 106).

[PP87] Penco Petrov Petrushev and Vasil Atanasov Popov. *Rational Approximation of Real Functions*. Cambridge University Press, 1987 (cit. on pp. 44, 45).

[PO02] Armin Pruessner and Dianne P. O'Leary. "Blind deconvolution using a regularized structured total least norm algorithm". In: *SIAM J. Matrix Anal. Appl.* 24.4 (2002), pp. 1018–1037 (cit. on p. 51).
DOI: 10.1137/S0895479801395446.

[Ruh79] Axel Ruhe. "Implementation aspects of band Lanczos algorithms for computation of eigenvalues of large sparse symmetric matrices". In: *Math. Comp.* 33.146 (1979), pp. 680–687 (cit. on pp. 106, 111, 113, 131).
DOI: 10.1090/S0025-5718-1979-0521282-9.

[Rui92] Daniel Ruiz. "Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment". CERFACS Technical Report, TH/PA/92/06. PhD thesis. Institut National Polytechnique de Toulouse, 1992 (cit. on p. 120).

[Saa87] Youcef Saad. "On the Lanczos method for solving symmetric linear systems with several right-hand sides". In: *Math. Comp.* 48.178 (1987), pp. 651–662 (cit. on p. 97).
DOI: 10.1090/S0025-5718-1987-0878697-3.

[Saa92] Youcef Saad. "Analysis of some Krylov subspace approximations to the matrix exponential operator". In: *SIAM J. Numer. Anal* 29.1 (1992), pp. 209–228 (cit. on p. 48).
DOI: 10.1137/0729014.

[Saa03] Youcef Saad. *Iterative Methods for Sparse Linear Systems*. 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003 (cit. on pp. 11, 14, 18, 19, 21, 28, 30, 99, 110).

[SS86] Youcef Saad and Martin H. Schultz. "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems". In: *SIAM J. Sci. Stat. Comput.* 7.3 (1986), pp. 856–869 (cit. on p. 26).
DOI: 10.1137/0907058.

[Saa$^+$00] Youcef Saad, Man-Chung Yeung, Jocelyne Erhel, and Frédéric Guyomarc'h. "A deflated version of the conjugate gradient algorithm". In: *SIAM J. Sci. Comput.* 21.5 (2000), pp. 1909–1926 (cit. on p. 96).
DOI: 10.1137/S1064829598339761.

[SW93] Robert Schaback and Helmut Werner. *Numerische Mathematik*. Springer, 1993 (cit. on pp. 21, 25).

[Sim03] Valeria Simoncini. "Restarted full orthogonalization method for shifted linear systems". In: *BIT Numerical Mathematics* 43 (2 2003), pp. 459–466 (cit. on p. 69).
DOI: 10.1023/A:1026000105893.

[SG95] Valeria Simoncini and Efstratios Gallopoulos. "An iterative method for nonsymmetric systems with multiple right-hand sides". In: *SIAM J. Sci. Comput* 16 (4 1995), pp. 917–933 (cit. on p. 89).
DOI: 10.1137/0916053.

[SG96] Valeria Simoncini and Efstratios Gallopoulos. “A hybrid block GMRES method for nonsymmetric systems with multiple right-hand sides”. In: *Journal of Computational and Applied Mathematics* 66.66 (1996). Proceedings of the Sixth International Congress on Computational and Applied Mathematics, pp. 457–469 (cit. on p. 107).
DOI: `10.1016/0377-0427(95)00198-0`.

[SS07] Valeria Simoncini and Daniel B. Szyld. “Recent computational developments in Krylov subspace methods for linear systems”. In: *Numer. Linear Algebra Appl.* 14.1 (2007), pp. 1–59 (cit. on pp. 97, 98).
DOI: `10.1002/nla.499`.

[Smi87] Charles Frederick Smith. “The performance of preconditioned iterative methods in computational electromagnetics”. AAI8803202. PhD thesis. Champaign, IL, USA: Air Force Inst. of Tech., Wright-Patterson AFB, OH, 1987 (cit. on p. 89).

[SPM89] Charles Frederick Smith, Andrew F. Peterson, and Raj Mittra. “A conjugate gradient algorithm for the treatment of multiple incident electromagnetic fields”. In: *IEEE Transactions on Antennas and Propagation* 37.11 (1989), pp. 1490–1493 (cit. on pp. 87, 89, 90).
DOI: `10.1109/8.43571`.

[SZ11] Tomohiro Sogabe and Shao-Liang Zhang. “An extension of the COCR method to solving shifted linear systems with complex symmetric matrices”. In: *East Asian Journal on Applied Mathematics* 1.2 (2011), pp. 97–107 (cit. on p. 69).
DOI: `10.4208/eajam.260410.240510a`.

[Soo14] Kirk Soodhalter. “A block MINRES algorithm based on the banded Lanczos method”. In: *Numerical Algorithms* (2014), pp. 1–22 (cit. on pp. 107, 114).
DOI: `10.1007/s11075-014-9907-z`.

[SO10] Andreas Stathopoulos and Konstantinos Orginos. “Computing and deflating eigenvalues while solving multiple right-hand side linear systems with an application to quantum chromodynamics”. In: *SIAM J. Sci. Comput.* 32.1 (2010), pp. 439–462 (cit. on pp. 88, 98–100, 103, 201).
DOI: `10.1137/080725532`.

[SS98] Andreas Stathopoulos and Youcef Saad. “Restarting techniques for the (Jacobi-)Davidson symmetric eigenvalue methods”. In: *Electron. Trans. Numer. Anal.* 7 (1998), pp. 163–181 (cit. on p. 100).

[SB93] Josef Stoer and Roland Bulirsch. *Introduction to Numerical Analysis*. 1993 (cit. on p. 48).
DOI: `10.1007/978-1-4757-2272-7`.

[ST05] Zdeněk Strakoš and Petr Tichý. “Error estimation in preconditioned conjugate gradients”. In: *BIT Numerical Mathematics* 45.4 (2005), pp. 789–817 (cit. on p. 12).
DOI: `10.1007/s10543-005-0032-1`.

[TSK09] Hiroto Tadano, Tetsuya Sakurai, and Yoshinobu Kuramashi. “Block BiCGGR: A new block Krylov subspace method for computing high accuracy solutions”. In: *JSIAM Lett.* 1 (2009), p. 44 (cit. on p. 107).

[Tar04] Albert Tarantola. *Inverse Problem Theory and Methods for Model Parameter Estimation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004 (cit. on p. 51).

[TB97] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, 1997 (cit. on p. 4).

[Vor87] Henk A. van der Vorst. “An iterative solution method for solving $f(A)x = b$, using Krylov subspace information obtained for the symmetric positive definite matrix A”. In: *J. Comput. Appl. Math.* 18.2 (1987), pp. 249–263 (cit. on p. 89). DOI: 10.1016/0377-0427(87)90020-3.

[Zen13] Mohamed Zenadi. “The solution of large sparse linear systems on parallel computers using a hybrid implementation of the block Cimmino method”. PhD thesis. Université de Toulouse, 2013 (cit. on p. 120).

[ZŠM12] Xiang Zhu, Filip Šroubek, and Peyman Milanfar. “Deconvolving PSFs for a better motion deblurring using multiple images”. In: *Computer Vision – ECCV 2012.* Ed. by Andrew Fitzgibbon, Svetlana Lazebnik, Pietro Perona, Yoichi Sato, and Cordelia Schmid. Vol. 7576. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 636–647 (cit. on p. 63). DOI: 10.1007/978-3-642-33715-4_46.

[Zol77] Egor I. Zolotarev. “Application of the elliptic functions to the problems on the functions of the least and most deviation from zero”. In: *Zapiskah Rossijskoi Akad. Nauk.* (1877). Золотарёв Е. И. Приложение эллиптических функций к вопросам о функциях, наименее уклоняющихся отнуля // Известия Санкт-Петербургск. Акад. Наук.— 1877.—Т. XXX, вып. 5. (См. также: Золотарёв Е. И. Собр. соч. Т. II.—Л.: АН СССР, 1932.—С. 1—59.) (cit. on p. 44).