Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

IDR(s) as a projection method

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

MASTER OF SCIENCE
in
Science Education and Communication

by

Marijn Bartel Schreuders

Delft, the Netherlands
February 22, 2014

MSc Thesis Science Education and Communication

IDR(s) as a projection method

Marijn Bartel Schreuders

Delft University of Technology

Daily supervisor

Dr. ir. M.B. van Gijzen

Responsible professor

Prof. dr. ir. C. Vuik

Other thesis committee members

Dr. J.G. Spandaw

R.A. Astudillo

February 22, 2014

Delft, the Netherlands

Table of contents

1. Introduction

Numerical linear algebra is concerned with the study of algorithms for performing linear algebra computations on computers. It is often a fundamental part of engineering and computational science problems and it is put into practice in a lot of area's. Within this field, *iterative methods* methods play an important role. Methods like the Arnoldi method, the Generalised Minimal Residual (GMRES) method and the Conjugate Gradient method make it possible to solve complex problems on a computer that were impossible to solve in earlier times.

The methods in this literature thesis are all examples of iterative methods. These are methods that generate a sequence of improving approximate solutions, rather than finding an exact solution in a finite number of steps. Moreover, these methods are examples of *projection methods.* In a projection method we try to find an approximate solution x_m in a subspace $\mathcal{K}_m$, such that the residual $r_m = b - Ax_m$ is orthogonal to another subspace $\mathcal{L}_m$. A *Krylov subspace method* is a projection method for which $\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2r_0 \ldots, A^{m-1}r_0\}$. Krylov subspace methods work by forming a basis for the Krylov subspace.

One example of an iterative method is the IDR method, which was first proposed in 1980 by Peter Sonneveld [Wesseling and Sonneveld, 1980]. In recent years there has been renewed interest in this method, which led to the IDR(s) method [Sonneveld and Van Gijzen, 2008] [Simoncini and Szyld, 2010]. The IDR(s) method is the main topic of this graduation project. We will see that the IDR(s) method fits perfectly in the framework of projection method and Krylov subspace methods.

Structure of this literature thesis

Chapter 2 describes the definitions that are used frequently or are of great importance to the rest of this thesis. Chapter 3 is the foundation of this thesis. It describes the theory behind projection methods. It also gives a general sketch of what a projection method looks like. Finally it explains what Krylov subspaces are and how they can be seen in the light op projection methods.

Chapter 4 will make the reader acquainted with examples of Krylov subspace methods. These can be divided into methods that solve eigenvalue problems and methods that solve linear systems of equations. We will treat the Arnoldi method, the Lanczos method and the Bi-Lanczos method as examples of methods that solve eigenvalue problems. Methods that solve linear systems of equations include the Full Orthogonalisation Method, the GMRES method, the Conjugate Gradient method and the Bi-Conjugate Gradient method. The Lanczos and Bi-Lanczos method can also be adapted for solving linear systems of equations. Moreover, chapter 4 also explains how all these method can be seen as projection methods. In chapter 5 we will see that the CG-type methods are mathematically equivalent to Lanczos-type methods. Hence the convergence behaviour of these methods should be the same. We will illustrate this with four examples.

In chapter 6 we arrive at the core of this literature thesis, that is the IDR(s) method. We will describe the IDR(s) algorithm and its performance. This chapter is the basis of this literature thesis. In chapter 7 we will present some motivating examples that show the power of the IDR(s) method. In the last chapter, chapter 8, we will describe the goals that we want to achieve with this graduation project.

2. Definitions

Throughout this thesis, we will use various definitions. The ones that are relevant and / or frequently used are listed below.

Definition 2.1 (Inner product).
Let a and b be two vectors in $\mathbb{R}^n$. The inner product (a, b) of a and b is defined as

$$(a, b) = a^T \cdot b = \sum_{i=1}^{n} a_i \cdot b_i.$$

It is easy to see that $(a, b) = (b, a)$, since multiplication is a commutative operation.

Definition 2.2 (Orthogonality).
Two vectors $a_i \in \mathbb{R}^n$ and $a_j \in \mathbb{R}^n$ are said to be orthogonal if $(a_i, a_j) = 0$ when $i \neq j$.

Vectors in a set $S = \{a_1, a_2, \ldots, a_m\}$ are said to be *pairwise orthogonal* if $(a_i, a_j) = 0 \ \ \forall i \neq j$.

Definition 2.3 (Orthonormality).
Two vectors $a_i \in \mathbb{R}^n$ and $a_j \in \mathbb{R}^n$ are said to be orthonormal if $(a_i, a_j) = \delta_{ij}$, where δ_{ij} denotes the Kronecker Delta function:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j. \end{cases}$$

Definition 2.4 (Eigenvalue, Eigenvector).
A scalar $\lambda \in \mathbb{C}$ is called an eigenvalue of $A \in \mathbb{C}^{n \times n}$ if a nonzero vector $x \in \mathbb{C}^n$ exists such that $Ax = \lambda x$. The vector x is called an eigenvector of A associated with λ.

Definition 2.5 (Hessenberg Matrix).
An upper Hessenberg matrix $H_n \in \mathbb{R}^{n \times n}$ is a matrix whose entries below the first subdiagonal are all zero:

$$H_n = \begin{pmatrix} h_{11} & h_{12} & h_{13} & \ldots & h_{1(n-1)} & h_{1n} \\ h_{21} & h_{22} & h_{23} & \ldots & h_{2(n-1)} & h_{2n} \\ 0 & h_{32} & h_{33} & \ldots & h_{3(n-1)} & h_{3n} \\ 0 & 0 & h_{43} & \ldots & h_{4(n-1)} & h_{4n} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & h_{n(n-1)} & h_{nn} \end{pmatrix}.$$

A *lower Hessenberg Matrix H_n* $\in \mathbb{R}^{n \times n}$ is a matrix whose entries above the first superdiagonal are all zero.

3. Projection methods

In this chapter we will explore the area of projection methods. For the information in this chapter, we have used the book'Iterative methods for sparse linear systems', written by Yousef Saad [Saad, 2003], which is considered an influential book in Numerical linear algebra.

Consider the linear system

$$Ax = b, \tag{3.1}$$

where $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$.

Contrary to direct methods, which try to find an exact solution, iterative methods try to find an approximate solution to equation (3.1). The iterative methods that we will discuss in chapter 4 are examples of *projection methods*. A projection method tries to find an approximate solution to equation (3.1) by extracting it from a subspace of $\mathbb{R}^n$ with dimension $m \leq n$. This subspace is often denoted by $\mathcal{K}_m$ and is called the *subspace of candidate approximants* or *search subspace*. From now on, we will use the latter.

In order to find an approximate solution, m constraints must be imposed on $\mathcal{K}_m$. This is typically done by requiring that the residual vector $r = b - Ax$ is orthogonal to m linearly independent vectors. These vectors give rise to another subspace $\mathcal{L}_m$ with dimension m and it is called the *subspace of constraints* or *left subspace*. Once again, we will use the latter throughout the rest of my thesis.

There are two kinds of projection methods: *orthogonal* projection methods and *oblique* projection methods. In orthogonal projection methods the search subspace $\mathcal{K}_m$ equals the left subspace $\mathcal{L}_m$ and in oblique projection methods $\mathcal{L}_m$ and $\mathcal{K}_m$ are different from each other.

3.1 General projection methods

Consider equation (3.1) and let $\mathcal{K}_m$ and $\mathcal{L}_m$ be two subspaces of $\mathbb{R}^n$ with dimension m. Define $r_m = b - Ax_m$. A projection method onto the subspace $\mathcal{K}_m$ and orthogonal to $\mathcal{L}_m$ tries to find an approximate solution x_m to equation (3.1) by requiring that x_m belongs to $\mathcal{K}_m$ such that $r_m \perp \mathcal{L}_m$:

$$\text{Find} \quad x_m \in \mathcal{K}_m \qquad \text{such that} \qquad r_m \perp \mathcal{L}_m. \tag{3.2}$$

These conditions are called the *Petrov-Galerkin* conditions. When $\mathcal{L}_m = \mathcal{K}_m$, the Petrov-Galerkin conditions are referred to as the Galerkin conditions.

It is also possible to use the initial guess x_0 as a source of extra information to find an approximate solution. The approximate solution must now be found in the affine subspace $x_0 + \mathcal{K}_m$ instead of the vector space $\mathcal{K}_m$. Hence, (3.2) changes into:

$$\text{Find} \quad x_m \in x_0 + \mathcal{K}_m \qquad \text{such that} \qquad r_m \perp \mathcal{L}_m. \tag{3.3}$$

According to (3.3), it is possible to write $x_m = x_0 + \delta$ with $\delta \in \mathcal{K}_m$. Using $r_0 = b - Ax_0$, we can rewrite r_m as:

$$r_m = b - Ax_m = b - A(x_0 + \delta) = b - Ax_0 - A\delta = r_0 - A\delta.$$

Hence, (3.3) can be written as:

$$\text{Find} \quad x_m \in x_0 + \mathcal{K}_m \qquad \text{such that} \qquad r_0 - A\delta \ \perp \ \mathcal{L}_m.$$

Let w be a vector in $\mathcal{L}_m$. Since all vectors $w \in \mathcal{L}_m$ are orthogonal to $r_m = r_0 - A\delta$, the inner product $(r_0 - A\delta, w) = 0$. The solution to equation (3.1) can now be defined as:

$$x_m = x_0 + \delta, \qquad \delta \in \mathcal{K}_m, \tag{3.4}$$
$$(r_0 - A\delta, w) = 0, \qquad \forall w \in \mathcal{L}_m, \tag{3.5}$$

where $(r_0 - A\delta, w)$ is the inner product of the vectors $r_0 - \delta A$ and w. In order to find the approximate solution x_m, we have to solve $(r_0 - A\delta, w)$ for δ.

In each step of a projection method, a new residual is calculated. This new residual should be orthogonal to the search subspace $\mathcal{L}_m$. Recalling that $r_m = r_0 - A\delta$, Figure 3.1 illustrates the orthogonality condition [Saad, 2003, p. 134].

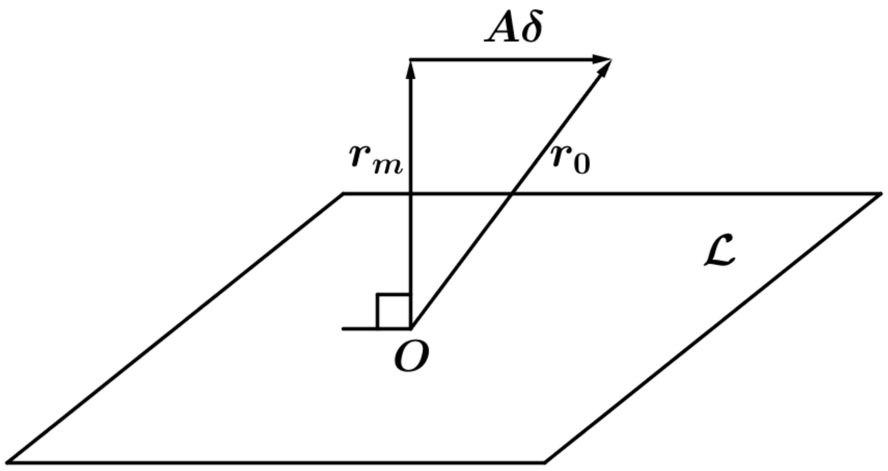

Figure 3.1: Interpretation of the orthogonality condition

3.2 Matrix-vector representation of a projection process

Let the column-vectors of $V_m = [v_1, v_2, \ldots, v_m]$ and $W_m = [w_1, w_2, \ldots, w_m]$, both $n \times m$ matrices, form orthonormal bases for $\mathcal{K}_m$ and $\mathcal{L}_m$ respectively. The approximate solution to equation (3.1) can be written as:

$$x_m = x_0 + V_m y_m, \tag{3.6}$$

This is true, since the approximate solution can be written as the initial guess plus a linear combination of the orthonormal vectors in $\mathcal{K}_m$. The vector y_m contains the coefficients for the column vectors of V_m.

$$\begin{aligned} V_m y_m &= [v_1, v_2, \ldots v_m]\, y_m \\ &= \begin{pmatrix} v_{11}y_1 + v_{12}y_2 + \ldots + v_{1m}y_m \\ v_{21}y_1 + v_{22}y_2 + \ldots + v_{2m}y_m \\ \vdots \\ v_{n1}y_1 + v_{n2}y_2 + \ldots + v_{nm}y_m \end{pmatrix} \\ &= y_1 v_1 + y_2 v_2 + \ldots y_m v_m, \end{aligned}$$

where v_{nm} is the n-th element of v_m. When we substitute (3.6) in r_m, we get

$$r_m = b - Ax_m = b - Ax_0 - AV_m y_m = r_0 - AV_m y_m.$$

Since $r_m \perp W_m$ by definition, the orthogonality condition in (3.5) can be written as

$$W_m^T(r_0 - AV_m y_m) = 0 \qquad \Longleftrightarrow \qquad W_m^T r_0 = \left(W_m^T A V_m\right) y_m.$$

If we assume that the matrix $W_m^T A V_m$ is nonsingular (invertible), than we have an explicit solution for y_m and, since x_m is a function of y_m, also for x_m:

$$\begin{aligned} y_m &= \left(W_m^T A V_m\right)^{-1} W_m^T r_0 \\ x_m &= x_0 + V_m \left(W_m^T A V_m\right)^{-1} W_m^T r_0 \end{aligned}$$

This gives rise to the following general algorithm for projection methods.

Algorithm 3.1 General Projection Method

```
1: Until convergence; **Do**
2:     Select a pair of subspaces $\mathcal{K}_m$ and $\mathcal{L}_m$
3:     Choose bases $V_m = [v_1, v_2, \ldots, v_m]$ and $W_m = [w_1, w_2, \ldots, w_m]$ for $\mathcal{K}_m$ and $\mathcal{L}_m$
4:     $r_m := b - Ax_m$
5:     $y_m := \left(W_m^T A V_m\right)^{-1} W_m^T r_0$
6:     $x_m := x_0 + V_m y_m$
7: **EndDo**
```

In many projection methods, the matrix $W_m^T A V_m$ does not have to be computed explicitly, because it is a by- product of the algorithm. For instance, in FOM (see section 4.2.1) an upper Hessenberg matrix H_m is calculated, which is equal to the matrix $W_m^T A V_m$.

We have assumed that the matrix $W_m^T A V_m$ is nonsingular, but this might not always be the case. However, there are two important cases in which the nonsingularity of $W_m^T A V_m$ is guaranteed [Saad, 2003, p. 136].

Theorem 3.1.
Let A, $\mathcal{K}_m$ and $\mathcal{L}_m$ satisfy either one of the two following conditions

(i) A is positive definite and $\mathcal{L}_m = \mathcal{K}_m$;

(ii) A is nonsingular and $\mathcal{L}_m = A\mathcal{K}_m$.

Then the matrix $B = W^T A V$ is nonsingular for any bases V_m and W_m of $\mathcal{K}_m$ and $\mathcal{L}_m$ respectively.

Proof. see [Saad, 2003, p. 136].

3.3 Krylov subspace methods

Section 3.1 explained that a projection method searches for an approximate solution $x_m = x_0 + \mathcal{K}_m$ of a linear system $Ax = b$ such that $b - Ax_m \perp \mathcal{L}_m$, where x_0 is the initial guess and $\mathcal{K}_m$ and $\mathcal{L}_m$ are subspaces of dimension m.

In Krylov subspace methods, the subspace $\mathcal{K}_m$ is defined as:

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\},$$

where $r_0 = b - Ax_0$ is the initial residual. The vectors $r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0$ are called the Krylov vectors.

There is a wide variety of Krylov subspace methods, such as the Full Orthogonalisation Method (FOM, see section 4.2.1), the GMRES method (see section 4.2.2) and the Conjugate Gradient method (CG, see section 4.2.4). Different Krylov subspace methods arise from using different subspaces for $\mathcal{L}_m$. Two widely used choices of $\mathcal{L}_m$ give rise to the best-known techniques. The first one is simply $\mathcal{L}_m = \mathcal{K}_m$ and the other one is $\mathcal{L}_m = A\mathcal{K}_m$. Other methods, such as the Lanczos Biorthogonalisation method (see section 4.1.3), take $\mathcal{L}_m = \mathcal{K}_m(A^T, r_0)$.

Since $x_m \in \mathcal{K}_m$, we can write x_m as a linear combination of the first m Krylov vectors or simply as a polynomial p_{m-1} in A of degree $(m-1)$, multiplied by r_0:

$$x_m = \left(\sum_{j=0}^{m-1} \alpha_j A^j \right) r_0 = p^{m-1}(A) r_0,$$

with $\alpha \in \mathbb{R}$ and $A^0 = I$.

We can obtain a similar expression for the residuals. First we write:

$$r_m = b - Ax_m = b - Ax_0 + Ax_0 - Ax_m = r_0 - A(x_m - x_0).$$

Since $x_m - x_0 \in \mathcal{K}_m$, we have that $A(x_m - x_0) \in \mathcal{K}_{m+1}$. r_0 is in any Krylov subspace, so also in $\mathcal{K}_{m+1}$. Therefore $r_m \in \mathcal{K}_{m+1}$. We can write r_m as a linear combination of the first $(m+1)$ Krylov vectors or simply as a polynomial q_{m+1} of degree $(m+1)$ in A multiplied by r_0:

$$r_m = \left(\sum_{j=0}^{m+1} \beta_j A^j \right) r_0 = q_{m+1}(A) r_0.$$

4. Krylov subspace methods

Numerical linear algebra is often concerned with two kinds of problems: eigenvalue problems and solving (linear) systems of equations.

In an eigenvalue problem one tries to find an eigenvalue λ and an eigenvector u corresponding to λ such that $Au = \lambda u$. Eigenvalues have many applications in mathematics. For example, eigenvalues can be used to get a better understanding of the convergence behaviour of numerical methods. The second type of problem is finding the solution of a linear system of equations. When given a matrix A and a vector b, one tries to find the solution x of the linear system $Ax = b$.

There is a wide variety of methods available to solve either problem. Krylov subspace methods are one of those. Krylov subspace methods can be put into several categories, depending on the kind of problem we want to solve and the characteristics of A. Chapter 4 will discuss several of these methods. Paragraph 4.1 will discuss several Krylov subspace methods to solve eigenvalue problems and paragraph 4.2 will discuss several Krylov subspace methods to solve linear systems of equations. Figure 4.1 shows the methods that will be discussed and shows in which category they belong.

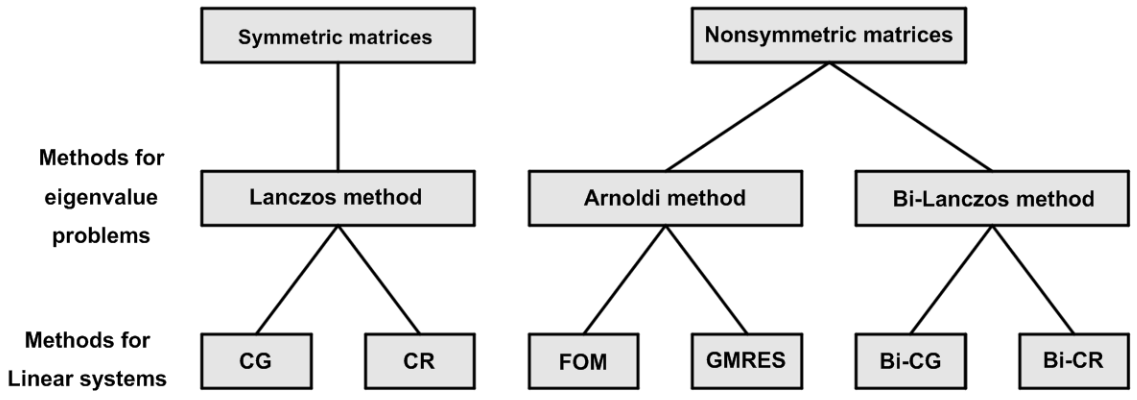

Figure 4.1: Krylov subspace methods

4.1 Krylov subspace methods for eigenvalue problems

In an eigenvalue problem, one wants to find the eigenvalues $\lambda \in \mathbb{R}$ and corresponding eigenvectors $u \in \mathbb{R}^n$ of a matrix $A \in \mathbb{R}^{n\times n}$. The eigenvalues can be found by solving the equation $\det(A - \lambda I) = 0$, where I is the $(n \times n)$ identity matrix. The eigenvector corresponding to a particular eigenvalue λ_i can be found by solving the equation $(A - \lambda_i I)u = 0$. However, this is often an expensive calculation, since A might be a large dense matrix. The Arnoldi method (see section 4.1.1), the Lanczos method (see section 4.1.2) and the Lanczos Biorthogonalisation method (see section 4.1.3) are three Krylov subspace methods for solving eigenvalue problems that work around this problem.

4.1.1 The Arnoldi method

The Arnoldi method [Saad, 2003, pp. 160-165] is a Krylov subspace method (see section 3.1). It finds the eigenvalues of a general non-Hermitian (nonsymmetric with $a_{ij} \in \mathbb{R}$) matrix $A \in \mathbb{R}^{n \times n}$. It was proposed by Walter Edwin Arnoldi in 1951 [Arnoldi, 1951]. For large matrices, it can be very expensive to calculate the eigenvalues. The main idea of the Arnoldi method is to find an upper Hessenberg matrix $H_m \in \mathbb{R}^{m \times m}$ with $m \ll n$, whose eigenvalues are accurate approximations to some of the eigenvalues of A. This is accomplished by building an orthonormal basis of vectors $V_m = [v_1, \dots v_m]$ for the search subspace $\mathcal{K}_m$, with

$$\mathcal{K}_m(A, v_1) = span\{v_1, Av_1, A^2 v_1, \dots, A^{m-1} v_1\}. \tag{4.1}$$

In iteration j, an extra vector v_j is added to the basis. Since H_m is much smaller than A, the eigenvalues are much cheaper to compute. Algorithm 4.1 shows one possible form of the Arnoldi method. The implementation can be found in appendix A.2.

Algorithm 4.1 The Arnoldi Method

```
 1: Choose an initial vector $v_1$ such that $||v_1|| = 1$
 2: For $j = 1, 2, \dots, m$ **Do**
 3:     $w_j := Av_j$
 4:     For $i = 1, 2, \dots, j$ **Do**
 5:         $h_{ij} = (w_j, v_i)$
 6:         $w_j := w_j - h_{ij} v_i$
 7:     **EndDo**
 8:     $h_{j+1,j} = ||w_j||_2$
 9:     **If** $h_{j+1,j} = 0$
10:         Stop
11:     **EndIf**
12:     $v_{j+1} = w_j / h_{j+1,j}$
13:     Build the Hessenberg matrix $H_j$ and calculate its eigenvectors
14:     **If** stop criterion satisfied
15:         Stop
16:     **EndIf**
17: **EndDo**
18: Approximate the eigenvalues and eigenvectors of $A$ using the upper
19: Hessenberg matrix $H_m$ and the orthonormal basis $V_m$.
```

First we have to choose a starting vector v_1 with $||v_1||_2 = 1$. In the remainder of the text, we will use $|| \cdot ||$ for the Euclidian norm when there's no ambiguity. Each subsequent basis vector v_{j+1} ($j = 1, \dots, m$) is calculated by multiplying the previous vector v_j with A (line 3), orthogonalising it with respect to all the previous basis vectors using the modified Gram-Schmidt process (lines 5-6) and finally orthonormalising it (line 12) [Saad, 2003, p. 12]. When the stopping criterion is satisfied, the algorithm calculates the Ritz values θ_i and the eigenvectors s_i of H_m with $i = 1, \dots, m$. The Ritz values of H_m are good approximations to some of the eigenvalues of A. The eigenvectors u_i of A can be approximated by the Ritz vector $V_m s_i$ [Holub and Van Loan, 1996, p. 500].

For the stopping criterion, we use $||r_j|| < 10^{-8}$, where $||r_j|| := ||Au_i - \lambda_i u_i||, i = 1, \ldots, j$, is the residual in the j-th iteration. Note that this stopping criterion is an expensive one, since we have to compute a matrix-vector product in each iteration and A might be a large dense matrix. Fortunately, we can work around this problem.

We substitute line 3 of the algorithm into line 6, line 6 into line 12 and we multiply both sides of the equation with $h_{j+1,j}$ to obtain

$$h_{j+1,j}v_{j+1} = Av_j - \sum_{i=1}^{j} h_{ij}v_i. \qquad \text{for } j = 1, \ldots, m$$

We can rewrite this as

$$\begin{aligned} Av_j &= h_{j+1,j}v_{j+1} + \sum_{i=1}^{j} h_{ij}v_i \qquad \text{for } j = 1, \ldots, m & (4.2)\\ &= \sum_{i=1}^{j+1} h_{ij}v_i \qquad \text{for } j = 1, \ldots, m. & (4.3)\end{aligned}$$

If we define $V_m = [v_1, \ldots, v_m]$, we can write these equations in matrix-vector notation:

$$\begin{aligned} AV_m &= V_m H_m + h_{m+1,m}v_{m+1}e_m^T, & (4.4)\\ &= V_{m+1}\bar{H}_m, & (4.5)\end{aligned}$$

where $A \in \mathbb{R}^{n\times n}$, $V_m \in \mathbb{R}^{n\times m}$, $H_m \in \mathbb{R}^{m\times m}$, $\bar{H}_m \in \mathbb{R}^{(m+1)\times j}$ and e_m^T the transpose of the m-th unit vector.

Equation (4.4) can be used to formulate an efficient stopping criterion for the Arnoldi method. We substitute it into the definition of the residual and find (for $j = 1, \ldots, m$ and $i = 1, \ldots, j$):

$$\begin{aligned} ||r_j|| &= ||Au_i - \lambda_i u_i||\\ &= ||AV_j s_i - \theta_i V_j s_i||\\ &= ||V_j H_j s_i + h_{j+1,j}v_{j+1}e_j^T s_i - \theta_i V_j s_i||\\ &= ||V_j(H_j s_i - \theta_i s_i) + h_{j+1,j}v_{j+1}s_i(j)||\\ &= ||h_{j+1,j}v_{j+1}s_i(j)||\\ &= ||h_{j+1,j}s_i(j)||\\ &= |h_{j+1,j}| \cdot |s_i(j)|.\end{aligned}$$

Equation (4.4) is used in the third line and the fifth line reduces to $||h_{j+1,j}s_i(j)||$, because $||v_{j+1}|| = 1$, since the vectors in $V_j = [v_1, \ldots, v_j]$ are pairwise orthonormal. Hence, we use the stopping criterion:

$$|h_{j+1,j}| \cdot |s_i(j)| < 10^{-8}. \qquad (4.6)$$

In the j-th iteration, the algorithm produces j eigenvectors. The eigenvector(s) of H_j that we should use, depends on which eigenvalue(s) of A. If we are interested in. For instance, if we want to approximate the largest eigenvalue of A, then we should use the eigenvector corresponding to the eigenvalue of H_j with the largest magnitude.

Finally, we have the following result:

Theorem 4.1.
A Hessenberg matrix produced by the Arnoldi method will be a tridiagonal matrix if $A \in \mathbb{R}^{n\times n}$ is a symmetric matrix.

Proof.

Recall that $V_m^T V_m$ is equal to the identity matrix, since the column vectors of V_m^T are pairwise orthonormal. Multiplying both sides of equation (4.4) with V_m^T, we have:

$$V_m^T A V_m = H_m. \tag{4.7}$$

When we transpose both sides, we get

$$V_m^T A^T V_m = H_m^T. \tag{4.8}$$

Since A is symmetric, we have $A = A^T$. Hence, equation (4.8) can be written as

$$V_m^T A V_m = H_m^T. \tag{4.9}$$

Since the right-hand sides of equations (4.7) and (4.9) are the same, we have $H_m^T = H_m$. From this it follows that H_j is a tridiagonal matrix, since every element above the first superdiagonal and under the first subdiagonal must be 0 because of the structure of a Hessenberg matrix.

$\blacksquare$

4.1.2 The Lanczos method

The Lanczos method [Saad, 2003, pp. 194-195] is a Krylov subspace method that is used for finding the eigenvalues of symmetric matrices. It can be seen as a simplification of the Arnoldi method for the case that A is symmetric. The Lanczos method was named after Cornelius Lanczos, a Hungarian mathematician. Symmetry implies that the eigenvalues of A are real. The Lanczos algorithm is especially useful in situations where a few of A's largest or smallest eigenvalues are desired [Holub and Van Loan, 1996, p. 470]. Just as the Arnoldi method, it builds an orthonormal basis V_m for the Krylov subspace $\mathcal{K}_m$, which was defined in equation (3.7). The Lanczos method also produces a tridiagonal matrix T_m. Algorithm 4.2 shows one possible form of the Arnoldi method. The implementation can be found in appendix A.3.

In line 4-6 the algorithm finds a new search direction orthogonal to all the previous vectors v and in line 11 orthonormalisation takes place. The vectors $\{v_j\}_{j=1}^m$ are the 'Lanczos vectors' and they can be used to find an approximation to the eigenvectors of A. In order to do this, all the Lanczos vectors have to be stored. The algorithm builds a (growing) tridiagonal matrix $T_j \in \mathbb{R}^{j\times j}$ in each iteration. T_j takes the following form:

$$T_j = \begin{pmatrix} \alpha_1 & \beta_2 & & & & \\ \beta_2 & \alpha_2 & \beta_3 & & O & \\ & \beta_3 & \alpha_3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & O & & \ddots & \ddots & \beta_j \\ & & & & \beta_j & \alpha_j \end{pmatrix}. \tag{4.10}$$

When the stopping criterion is satisfied (after m iterations), the Lanczos algorithm calculates the Ritz values θ_i and the eigenvectors s_i of T_m with $i = 1, \ldots, m$. The Ritz values of T_m are good approximations to some of the eigenvalues of A. The eigenvectors u_i of A can be approximated by the Ritz vector $V_m s_i$.

Algorithm 4.2 Lanczos method

```
 1: Choose an initial vector $v_1$ such that $||v_1||_2 = 1$.
 2: Set $\beta_1 = 0$ and $v_0 = 0$.
 3: For $j = 1, 2, \ldots, m$ Do
 4:     $w_j := Av_j - \beta_j v_{j-1}$
 5:     $\alpha_j := (w_j, v_j)$
 6:     $w_j := w_j - \alpha_j v_j$
 7:     $\beta_{j+1} := ||w_j||_2$
 8:     If $\beta_{j+1} = 0$
 9:         Stop
10:     EndIf
11:     $v_{j+1} := w_j / \beta_{j+1}$
12:     Set $T_j = \text{tridiag}(\{\beta_i\}_{i=2}^{j}, \{\alpha_i\}_{i=1}^{j}, \{\beta_i\}_{i=2}^{j})$ and calculate its eigenvectors
13:     If stop criterion satisfied
14:         Stop
15:     EndIf
16: EndDo
17: Approximate the eigenvalues and eigenvectors of $A$ using the tridiagonal
18: matrix $T_m$ and the orthonormal basis $V_m$.
```

We can formulate a cheap stopping criterion for the Lanczos method in the same way as we did for the Arnoldi method. By substituting line 4 in line 6, line 6 in line 11, multiplying both sides of the equation with β_{j+1} and rewriting this equation, we get

$$Av_j = \beta_j v_{j-1} + a_j v_j + \beta_{j+1} v_{j+1} \qquad \text{for } j = 1, \ldots, m. \tag{4.11}$$

From this expression it is very easy to see that an orthonormal basis can be build using only three vectors in each step. Therefore, the Lanczos method is called a short-recurrence method, an iterative method that only needs a few previous vectors to build a new one. This is contrary to the Arnoldi method, which is a long-recurrence method: an iterative method which needs all the previous vectors to build a new one. (compare equation (4.3) to equation (4.11)).

However, we will see that we need all the basis vectors for approximating the eigenvectors of A. (4.11) can be written in matrix-vector notation as

$$AV_m = V_m T_m + \beta_{m+1} v_{m+1} e_m^T. \tag{4.12}$$

Note that equation (4.12) is similar to equation (4.4) (with H_m replaced by T_m), since $h_{m+1,m} = ||w_m|| = \beta_{m+1}$. We can therefore use the same stopping criterion as used in the Arnoldi method, that is:

$$|\beta_{j+1}| \cdot |s_i(j)| < 10^{-8}. \tag{4.13}$$

4.1.3 The Lanczos Biorthogonalisation method

Although the Arnoldi method has some good properties (it is a stable method with respect to rounding errors and breakdown does not occur), it does have some disadvantages. Arnoldi uses Modified Gram-Schmidt orthogonalisation of all vectors v_j and this causes the work (the number of vector operations) to increase quadratically in each subsequent step. Although H_m is relatively small compared to A, this might result in having to restart the algorithm. However, in this case the good convergence properties are lost [Vuik and Lahaye, 2010, p.107]

The Lanczos Biorthogonalisation method [Saad, 2003, pp. 230-233], also called the Bi-Lanczos method or nonsymmetric Lanczos method, is a Krylov subspace method that uses biorthogonalisation to find the eigenvalues of a nonsymmetric matrix A. The Bi-Lanczos method produces two sequences of vectors $\{v_j\}_{j=1}^m$ and $\{w_j\}_{j=1}^m$ that are biorthogonal. That means that if $V_m = [v_1, \ldots, v_j]$ and $W_m = [w_1, \ldots w_j]$, then $V_m^T W_m = W_m^T V_m = I$. V_m and W_m are two bases for the two subspaces $\mathcal{K}_m(A, v_1)$ and $\mathcal{K}_m(A^T, w_1)$:

$$\begin{aligned}
\mathcal{K}_m(A, v_1) &= span\{v_1, Av_1, A^2v_1, \ldots, A^{m-1}v_1\} \\
\mathcal{K}_m(A^T, w_1) &= span\{w_1, A^T w_1, (A^T)^2 w_1, \ldots, (A^T)^{m-1} w_1\}.
\end{aligned}$$

Algorithm 4.3 shows one possible form of the Lanczos Biorthogonalisation method. The implementation can be found in appendix A.4. First we have to choose two starting vectors v_1 and w_1 such that $(v_1, w_1) = 1$. In line 4-6 the algorithm finds a new search direction orthogonal to all the previous vectors v and in line 12 and 13 normalisation takes place. Next, the algorithm builds a (growing) tridiagonal matrix $T_j \in \mathbb{R}^{j\times j}$ in each iteration. T_j takes the following form:

$$T_j = \begin{pmatrix} \alpha_1 & \beta_2 & & & & \\ \delta_2 & \alpha_2 & \beta_3 & & O & \\ & \delta_3 & \alpha_3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & O & & \ddots & \ddots & \beta_j \\ & & & & \delta_j & \alpha_j \end{pmatrix}. \tag{4.14}$$

Algorithm 4.3 Lanczos Biorthogonalisation method

```
1: Choose two vectors $v_1$ and $w_1$ such that $(v_1, w_1) = 1$.
2: Set $\beta_1 = \delta_1 = 0$ and $v_0 = w_0 = 0$
3: For $j = 1, 2, \ldots, m$ Do
4:     $\alpha_j := (w_j, v_j)$
5:     $\hat{v}_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}$
6:     $\hat{w}_{j+1} = A^T w_j - \alpha_j w_j - \delta_j w_{j-1}$
7:     $\delta_{j+1} := |(\hat{v}_{j+1}, \hat{w}_{j+1})|^{1/2}$
8:     If $\delta_{j+1} = 0$
9:         Stop
10:    EndIf
11:    $\beta_{j+1} := (\hat{v}_{j+1}, \hat{w}_{j+1})/\delta_{j+1}$
12:    $v_{j+1} = \hat{v}_{j+1}/\delta_{j+1}$
13:    $w_{j+1} = \hat{w}_{j+1}/\beta_{j+1}$
14:    Set $T_j = \text{tridiag}(\{\delta_i\}_{i=2}^{j}, \{\alpha_i\}_{i=1}^{j}, \{\beta_i\}_{i=2}^{j})$ and calculate its eigenvectors
15:    If stop criterion satisfied
16:        Stop
17:    EndIf
18: EndDo
19: Approximate the eigenvalues and eigenvectors of $A$ using the tridiagonal
20: matrix $T_m$ and the basis $V_m$.
```

Note that $\beta_{j+1} = \pm\delta_{j+1}$. This is easily seen when looking at the formulas for β_{j+1} and δ_{j+1} in line 7 and 11. If $(\hat{v}_{j+1}, \hat{w}_{j+1})$ is positive, then $\beta_{j+1} = \delta_{j+1}$. If $(\hat{v}_{j+1}, \hat{w}_{j+1})$ is negative, then $\beta_{j+1} = -\delta_{j+1}$.

When the stopping criterion is satisfied (after m iterations), the Bi-Lanczos algorithm calculates the Ritz values θ_i and the eigenvectors s_i of T_m with $i = 1, \ldots, m$. The Ritz values of T_m are good approximations to some of the eigenvalues of A. The eigenvectors u_i of A can be approximated by the Ritz vector $V_m s_i$ [Holub and Van Loan, 1996, p. 500]

By substituting line 5 in line 12, multiplying both sides of the equation with $\delta_{j+1,j}$ and rewriting this equation, we get

$$Av_j = \beta_j v_{j-1} + \alpha_j v_j + \delta_{j+1} v_{j+1} \qquad \text{for } j = 1, \ldots, m.$$

We see that the Bi-Lanczos method is also a short recurrence method. The above expression can be written in matrix-vector notation as

$$AV_m = V_m T_m + \delta_{m+1} v_{m+1} e_m^T. \tag{4.15}$$

We can use equation (4.15) to obtain a cheap stopping criterion for the Bi-Lanczos method in a similar fashion as in the Arnoldi and Lanczos method. However, in the case of the Bi-Lanczos method, the vectors $v_1, \ldots v_j$ are not orthonormal ($||v_{j+1}|| \neq 1$). Hence, we obtain:

$$|\delta_{j+1}| \cdot |s_i(j)| \cdot ||v_{j+1}|| < 10^{-8}. \tag{4.16}$$

Note that we do not use the other basis W_m for finding the eigenvalues of A. This is a severe drawback of the Lanczos Biorthogonalisation method, since the extra calculations are more or less wasted. Moreover, there are more opportunities for the algorithm to break down. On the other hand, the Bi-Lanczos algorithm only a few vectors of storage compared to the Arnoldi method, since it is a short recurrence method (which Arnoldi is not) [Saad, 2003, p. 231].

4.2 Krylov subspace methods for solving linear systems

Suppose we want to solve the linear system $Ax = b$. Let $V_j = [v_1, \dots, v_j]$ be an orthonormal basis for the Krylov subspace $\mathcal{K}_j$. We can write the solution in the j-th iteration x_j as a x_0 and a linear combination of the vectors in V_j. Hence, $x_j = x_0 + V_j y_j$, where y_j is a vector with coefficients for the vector v_j. We now show how to find the coefficients for the column vectors of V_j.

Define the residual in the j-th iteration as $r_j = b - Ax_j$. Substituting x_j in the expression for the residual yields:

$$r_j = b - Ax_j = b - Ax_0 - AV_j y_j = r_0 - AV_j y_j. \tag{4.17}$$

Chapter 3 explained that in a projection method we have $r_j \perp \mathcal{L}_j$. If $\mathcal{L}_j = \mathcal{K}_j$, we find $V_j^T r_j \perp V_j^T \cdot \mathcal{K}_j$. Since V_j is an orthonormal basis for $\mathcal{K}_j$, we find that $V_j^T r_j = 0$. If $\mathcal{L}_j = A \cdot \mathcal{K}_j$, we find that $V_j^T r_j = 0$. In a similar fashion, since the projection method still builds an orthonormal basis for $\mathcal{K}_j$. Multiplying both sides of equation (4.17) with V_j^T gives:

$$V_j^T r_j = V_j^T (r_0 - AV_j y_j) = V_j^T r_0 - V_j^T AV_j y_j = V_j^T r_0 - B_j y_j = 0, \tag{4.18}$$

where $B_j = V_j^T AV_j$. As seen in section 4.1.1 and section 4.1.2, $V_j^T AV_j = H_j$ in the Arnoldi method and $V_j^T AV_j = T_j$ in the Lanczos method.

For the Bi-Lanczos method (with $w_1 = v_1$), we have $\mathcal{L}_j = \mathcal{K}_j(A^T, w_1) = \mathcal{K}_j(A^T, v_1)$ and we arrive at the following equation (using $W_j^T r_j =$):

$$W_j^T r_j = W_j^T (r_0 - AV_j y_j) = W_j^T r_0 - W_j^T AV_j y_j = W_j^T r_0 - B_j y_j = 0,$$

where B_j is the tridiagonal matrix T_j from the Bi-Lanczos method.

Since $v_1 = w_1 = r_0/||r_0||$, $\beta = ||r_0||$, $v_i^T v_j = \delta_{ij}$ and $w_i^T v_j = \delta_{ij}$ for $i, j = 1 \dots m$, we have:

$$V_j^T r_0 = ||r_0|| \cdot V_j^T \cdot v_1 = \beta e_1 \qquad \text{and} \qquad W_j^T r_0 = ||r_0|| \cdot W_j^T \cdot w_1 = \beta e_1, \tag{4.19}$$

where e_1 is the first unit vector. Using equation (4.18) and equation (4.19), we find:

$$\begin{aligned} y_j &= B_j^{-1} \beta e_1 & (4.20) \\ x_j &= x_0 + V_j y_j. & (4.21) \end{aligned}$$

4.2.1 The Full Orthogonalisation Method (FOM)

The Full Orthogonalisation Method (FOM) is a Krylov subspace method that is used for solving linear systems of equations. It is based on the Arnoldi method. FOM is an orthogonal projection method, meaning that $\mathcal{L}_m(A, v_1) = \mathcal{K}_m(A, v_1)$, with

$$\mathcal{K}_m(A, v_1) = \{v_1, Av_1, A^2v_1, \ldots, A^{m-1}v_1\}$$

We can find a solution to a linear system $Ax = b$ by using equation (4.20) and (4.21). Algorithm 4.4 shows one possible form of the Arnoldi method. The implementation of the algorithm can be found in appendix B.1.

Algorithm 4.4 Full Orthogonalisation Method (FOM)

```
1: Compute $r_0 = b - Ax_0$, $\beta := ||r_0||_2$ and $v_1 := r_0/\beta$
2: For $j = 1, 2, \ldots, m$ **Do**
3:     $w_j := Av_j$
4:     For $i = 1, 2, \ldots, j$ **Do**
5:         $h_{ij} = (w_j, v_i)$
6:         $w_j := w_j - h_{ij}v_i$
7:     **EndDo**
8:     $h_{j+1,j} = ||w_j||_2$
9:     **If** $h_{j+1,j} = 0$
10:        Stop
11:    **EndIf**
12:    $v_{j+1} = w_j/h_{j+1,j}$
13:    $y_j = H_j^{-1}(\beta e_1)$
14:    **If** stop criterion satisfied
15:        Stop
16:    **EndIf**
17: **EndDo**
18: $x = x_0 + V_m y_m$
```

Until line 12, the algorithm is exactly the same as the Arnoldi method's algorithm. In the j-th iteration, the algorithm builds an orthonormal basis for $\mathcal{K}_j$. In line 13 the coefficients for the orthonormal column vectors of V_j are computed. After the algorithm finished, both V_j and y_j are used to compute the approximate solution x_m in line 18. For the stopping criterion, we use equation (4.22) with $\epsilon = 10^{-8}$ [Vuik and Lahaye, 2010, p. 56]:

$$\frac{||r_j||}{||b||} = \frac{||b - Ax_j||}{||b||} < 10^{-8}. \tag{4.22}$$

As in the Arnoldi method, it is not efficient to compute the residual directly. Instead we substitute the approximate solution into equation (4.22) and using equation (4.4) in the third line and equation (4.20) in the fourth line with $B_j = H_j$, we find:

$$\begin{aligned}
||r_j|| &= ||b - Ax_j|| \\
&= ||b - Ax_0 - AV_jy_j|| \\
&= ||r_0 - AV_jy_j|| \\
&= ||\beta v_1 - V_jH_jy_j - h_{j+1,j}v_{j+1}e_j^Ty_j|| \\
&= ||\beta V_je_1 - \beta V_je_1 - h_{j+1,j}v_{j+1}y_j(j)|| \\
&= ||h_{j+1,j}v_{j+1}y_j(j)|| \\
&= |h_{j+1,j}| \cdot |y_j(j)|.
\end{aligned}$$

Substituting this into equation (4.22) and rewriting it, we find the following stopping criterion:

$$|h_{j+1,j}| \cdot |y_j(j)| < ||b|| \cdot 10^{-8}. \tag{4.23}$$

4.2.2 The Generalised Minimal RESidual (GMRES) method

The Generalised Minimal RESidual method, abbreviated as the GMRES method, is a Krylov subspace method for solving linear systems of equations. It minimises the residual norm in each step. GMRES is an oblique projection method, so $\mathcal{L}_m \neq \mathcal{K}_m$. Instead, we have:

$$\mathcal{L}_m(A, v_1) = A\mathcal{K}_m(A, v_1) = \ span\{Av_1, A^2v_1, \ldots, A^mv_1\}. \tag{4.24}$$

Just as FOM, GMRES is based on the Arnoldi method. GMRES is almost similar to FOM. However, there are some minor differences. Instead of equation (4.4), we use equation (4.5). We cannot use equation (4.20), since $\bar{H}_j$ isn't a square matrix. However, using equations (4.21) and (4.5) and substituting these in the equation for the norm of the residual yields:

$$||r_j|| = ||\beta e_1 - \bar{H}y_j||. \tag{4.25}$$

We now define y_j as the y that minimises equation (4.25):

$$y_j = min_y||\beta e_1 - \bar{H}_jy||. \tag{4.26}$$

y_j can be computed in Matlab using the backslash operator. Algorithm 4.5 shows one possible form of the GMRES method. The exact implementation can be found in appendix B.2.

Until line 12, the algorithm is exactly the same as the Arnoldi method's algorithm. The algorithm builds an orthonormal basis for $\mathcal{K}_m$. In line 13 the coefficients y_j for the vectors v_j are calculated by solving a minimisation problem. In line 18 the approximate solution x_m is calculated. For the stopping criterion, we can use equation (4.22), together with equation (4.25) and equation (4.26) to obtain:

$$||\beta e_1 - \bar{H}y_j|| < ||b|| \cdot 10^{-8}. \tag{4.27}$$

Algorithm 4.5 Generalised Minimal Residual method (GMRES)

```
1: Compute $r_0 = b - Ax_0$, $\beta := ||r_0||_2$ and $v_1 := r_0/\beta$
2: For $j = 1, 2, \ldots, m$ **Do**
3:     $w_j := Av_j$
4:     For $i = 1, 2, \ldots, j$ **Do**
5:         $h_{ij} = (w_j, v_i)$
6:         $w_j := w_j - h_{ij}v_i$
7:     **EndDo**
8:     $h_{j+1,j} = ||w_j||_2$
9:     **If** $h_{j+1,j} = 0$
10:        Stop
11:    **EndIf**
12:    $v_{j+1} = w_j/h_{j+1,j}$
13:    $y_j = min_y||\beta e_1 - \bar{H}_j y||$
14:    **If** stop criterion satisfied
15:        Stop
16:    **EndIf**
17: **EndDo**
18: $x = x_0 + V_m y_m$
```

4.2.3 The Lanczos method for systems of equations

The Lanczos method can be adapted to solve linear systems of equations. It is an analogue of FOM for symmetric matrices. The Lanczos method for systems of equations is an orthogonal projection method, so $\mathcal{L}_m(A, v_1) = \mathcal{K}_m(A, v_1)$, where $\mathcal{K}_m$ is defined as in equation (3.7).

In order to solve a linear system, all the Lanczos vectors need to be stored in a matrix $V_j = [v_1, v_2, \ldots, v_j]$. This matrix is needed for calculating a solution in each iteration. The approximate solution of the linear system $Ax = b$ can be calculated by using equations (4.20) and (4.21). A possible algorithm is given in algorithm 4.6 on the next page. An implementation is given in appendix B.3.

Until line 12 the algorithm is the same as the Lanczos method that is used for eigenvalue problems. In line 11 the algorithm builds the tridiagonal matrix T_j, which was defined in equation (4.10). In line 12 the coefficients y_j for the vectors v_j are calculated by solving a minimisation problem. In line 18 the approximate solution x_m is calculated.

In a similar way as in the Arnoldi method, we can derive a cheap stopping criterion. We substitute equation (4.21) into equation (4.22). Using equation (4.12) and equation (4.20) (with $B_j = T_j$), we find the following equation for the stopping criterion:

$$|\beta_{j+1}| \cdot |y_j(j)| < ||b|| \cdot 10^{-8}. \qquad (4.28)$$

Algorithm 4.6 Lanczos method for systems of equations

```
1: Compute $r_0 = b - Ax_0$, $\beta = ||r_0||$ and $v_1 = r_0/\beta$.
2: Set $\beta_1 = 0$ and $v_0 = 0$
3: For $j = 1, 2, \ldots, m$ **Do**
4:     $w_j := Av_j - \beta_j v_{j-1}$
5:     $\alpha_j := (w_j, v_j)$
6:     $w_j := w_j - \alpha_j v_j$
7:     $\beta_{j+1} := ||w_j||_2$
8:     **If** $\beta_{j+1} = 0$
9:         Stop
10:    **EndIf**
11:    $v_{j+1} := w_j/\beta_{j+1}$
12:    Set $T_j = \text{tridiag}(\{\beta_i\}_{i=2}^{j}, \{\alpha_i\}_{i=1}^{j}, \{\beta_i\}_{i=2}^{j})$
13:    $y_j := T_j^{-1}(\beta e_1)$
14:    **If** stop criterion satisfied
15:        Stop
16:    **EndIf**
17: **EndDo**
18: $x = x_0 + V_m y_m$
```

4.2.4 The Conjugate Gradient (CG) method

The Conjugate Gradient (CG) method is one of the best-known iterative methods for solving linear systems of equations. It is an orthogonal projection method, so $\mathcal{L}_m(A, r_0) = \mathcal{K}_m(A, r_0)$, where $\mathcal{K}_m$ is defined in equation (3.7). The matrix A must be symmetric and positive definite ($x^T Ax > 0$ for $x \neq 0$).

Let x be the exact solution of the linear system $Ax = b$. The idea of CG is to construct a vector $x_j \in \mathcal{K}_j$ in each iteration such that $||x - x_j||$ is minimal. It turns out that it is not possible to calculate this norm, since we do not know x beforehand. Instead we define a new norm, called the A-norm: $||y||_A = \sqrt{y^T Ay}$ [Vuik and Lahaye, 2010, p. 66]. In each iteration we now compute the approximate solution x_j such that

$$||x - x_j||_A = \min_{x_j \in \mathcal{K}_j} ||x - x_j||_A.$$

This gives rise to the Conjugate Gradient method as seen in algorithm 4.7. An implementation can be found in appendix B.4. In each iteration the algorithm calculates the new solution x_{j+1}, updates the residual r_{j+1} and updates the search direction p_{j+1}. We use equation (4.22) as a stopping criterion, where $||r_j||$ is computed directly.

The search directions are A-orthogonal: $||p_i^T Ap_j|| = 0$ for $i < j$. The name of the method is derived from the fact that r_{j+1}, the new residual vector, is orthogonal (conjugate) to all the previous search directions (gradients), so $p_i^T r_{j+1} = 0$ for $i < j$. Using this, we find that r_{j+1} is also orthogonal to the previous residuals:

Algorithm 4.7 The Conjugate Gradient method (CG)

```
1: Compute $r_0 = b - Ax_0$ and $p_0 := r_0$
2: For $j = 0, 1, 2, \dots$ m **Do**
3:     $\alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4:     $x_{j+1} := x_j + \alpha_j p_j$
5:     $r_{j+1} := r_j - \alpha_j Ap_j$
6:     **If** stop criterion satisfied
7:         Stop
8:     **EndIf**
9:     $\beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
10:    $p_{j+1} := r_{j+1} + \beta_j p_j$
11: **EndDo**
12: $x = x_m$
```

$$p_i^T r_{j+1} = \left(r_i + \beta_{i-1} p_{i-1}\right)^T r_{j+1} = r_i^T r_{j+1} + \beta_{i-1} p_{i-1} T r_{j+1} = r_i^T r_{j+1} = 0 \tag{4.29}$$

It is possible to derive the Lanczos method from CG and vice versa [Holub and Van Loan, 1996, §9.3.1 ; p. 528]. Moreover, CG is mathematically equivalent to the Lanczos method for systems of equations and therefore the two methods give the same convergence results (see section 5.1). One important difference is that the Lanczos method needs all the Lanczos vectors v_j for calculating the approximate solution: $x_j = x_0 + V_j y_j$. In each iteration an extra basis vector becomes available to approximate the solution more accurately. CG avoids this by updating the solution in each iteration: $x_{j+1} = x_j + \alpha_j p_j$. Once an iteration is complete, the previous v_j are not needed anymore.

4.2.5 The Conjugate Residual (CR) method)

In section 4.2.4 we found that CG was an analogue of FOM for Symmetric Positive Definite (SPD) matrices. Something similar holds for the Conjugate Residual (CR) method. It is an analogue of GMRES for Hermitian matrices. Because of this, the CR method is an orthogonal oblique method, that is: $\mathcal{L}_m = A\mathcal{K}_m$, with $\mathcal{K}_m$ as defined in equation (3.7). One possible algorithm of the CR method can be found in algorithm 4.8. The implementation of the CR method can be found in appendix B.5.

The algorithm is similar to the algorithm of CG. In each iteration the new solution x_{j+1}, the new residual r_{j+1} and the new search direction p_{j+1} are calculated. Just as in CG, the residual is calculated directly using (4.22).

In the CR method, the residual vectors r_i are A-orthogonal: $r_i A r_j = 0$ if $i \neq j$. Hence the name of the method. Moreover, the vectors $\{Ap_i\}_{i=1}^m$ are orthogonal. The convergence behaviour of CR and CG is similar and since the CR method requires extra storage and one extra vector update compared to CG, the latter is often preferred [Saad, 2003, pp. 203-204].

Algorithm 4.8 The Conjugate Residual method (CR)

```
1:  Compute $r_0 = b - Ax_0$ and set $p_0 := r_0$
2:  For $j = 0, 1, 2, \ldots$ m **Do**
3:      $\alpha_j := (r_j, Ar_j)/(Ap_j, Ap_j)$
4:      $x_{j+1} := x_j + \alpha_j p_j$
5:      $r_{j+1} := r_j - \alpha_j Ap_j$
6:      **If** stop criterion satisfied
7:          Stop
8:      **EndIf**
9:      $\beta_j := (r_{j+1}, Ar_{j+1})/(r_j, Ar_j)$
10:     $p_{j+1} := r_{j+1} + \beta_j p_j$
11:     $Ap_{j+1} = Ar_{j+1} + \beta_j Ap_j$
12: **EndDo**
13: $x = x_m$
```

4.2.6 The Lanczos Biorthogonalisation method for systems of equations

Just as the Lanczos method can be adapted to solve linear systems of equations, so can the Lanczos Biorthogonalisation method [Saad, 2003, p. 234]. This method is an oblique projection method and it builds two biorthogonal bases for the Krylov subspaces

$$\begin{aligned} \mathcal{K}_m(A, v_1) &= span\{v_1, Av_1, A^2v_1, \ldots, A^{m-1}v_1\} \\ \mathcal{L}_m(A^T, w_1) &= span\{w_1, A^Tw_1, (A^T)^2w_1, \ldots, (A^T)^{m-1}w_1\}. \end{aligned}$$

A possible algorithm is given in algorithm 4.9 and the implementation can be found in appendix B.6. Until line 14 the algorithm is exactly the same as the 'eigenvalue' version of the Bi-Lanczos algorithm. In line 14 the algorithm builds the tridiagonal matrix T_j, which was defined in equation (4.14). In line 15 the coefficients y_j for the vectors v_j are calculated by solving a minimisation problem. In line 20 the approximate solution x_m is calculated.

We can derive a cheap stopping criterion for the Bi-Lanczos method for systems of equations in a similar way as in the Lanczos method for linear systems (section (4.2.3)). Firstly, note that the vectors v_j are not orthogonal, so $||v_{j+1}|| \neq 1$. We substitute equation (4.21) into equation (4.22). Using equation (4.15) and equation (4.20) (with $B_j = T_j$), we find the following equation for the stopping criterion:

$$|\delta_{j+1}| \cdot |y_j(j)| \cdot ||v_{j+1}|| < ||b|| \cdot 10^{-8}. \tag{4.30}$$

Just as in the Bi-Lanczos method used for finding eigenvalues, the Bi-Lanczos method for systems of equations has the drawback that it only uses the matrix V_j for finding the approximate solution x_j (section (4.2.3)). The vectors w_j are essentially useless if it not desired to solve the dual system $A^Tx^* = b^*$.

Algorithm 4.9 Lanczos Biorthogonalisation method for systems of equations

```
1: Compute $r_0 = b - Ax_0$, $\beta = ||r_0||$ and $v_1 = w_1 = r_0/\beta$.
2: Set $\beta_1 = \delta_1 = 0$ and $v_0 = w_0 = 0$
3: For $j = 1, 2, \ldots, m$ Do
4:     $\alpha_j := (w_j, v_j)$
5:     $\hat{v}_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}$
6:     $\hat{w}_{j+1} = A^T w_j - \alpha_j w_j - \delta_j w_{j-1}$
7:     $\delta_{j+1} := |(\hat{v}_{j+1}, \hat{w}_{j+1})|^{1/2}$
8:     If $\delta_{j+1} = 0$
9:         Stop
10:    EndIf
11:    $\beta_j := (\hat{v}_{j+1}, \hat{w}_{j+1})/\delta_{j+1}$
12:    $v_{j+1} = \hat{v}_{j+1}/\delta_{j+1}$
13:    $w_{j+1} = \hat{w}_{j+1}/\beta_{j+1}$
14:    Set $T_j = \text{tridiag}(\{\delta_i\}_{i=2}^{j}, \{\alpha_i\}_{i=1}^{j}, \{\beta_i\}_{i=2}^{j})$
15:    $y_j = T_j^{-1}(\beta e_1)$
16:    If stop criterion satisfied
17:        Stop
18:    EndIf
19: EndDo
20: $x = x_0 + V_m y_m$
```

4.2.7 The Bi-Conjugate Gradient (Bi-CG) method

The Conjugate Gradient algorithm is one of the most widely used algorithms to solve a linear systems $Ax = b$. However, CG can only be used if A is an SPD matrix. When A is a general matrix, one of the methods to consider is the Bi-Conjugate Gradient method (Bi-CG). It was first proposed by Lanczos in 1952 [Lanczos, 1952]. The Bi-CG method is an oblique projection method onto $\mathcal{K}_m$ and orthogonal to $\mathcal{L}_m$ with

$$\begin{aligned}\mathcal{K}_m(A, v_1) &= span\{v_1, Av_1, A^2v_1, \ldots, A^{m-1}v_1\} \\ \mathcal{L}_m(A^T, w_1) &= span\{w_1, A^Tw_1, (A^T)^2w_1, \ldots, (A^T)^{m-1}w_1\}.\end{aligned}$$

where $v_1 = r_0/||r_0||$. The vector w_1 may be chosen arbitrarily provided that $(v_1, w_1) \neq 0$, but normally it is chosen to be equal to v_1.

A possible algorithm for the Bi-CG method is given in algorithm 4.10 and an algorithm is given in appendix B.7. First we define the dual system of $Ax = b$ as the system $A^Tx^* = b^*$. Next, we define $r_j^* = b^* - A^Tx_j^*$ as the j-th residual of the dual system. r_j* is often called the 'shadow residual'. In each iteration, the algorithm calculates the updated solution x_{j+1}, the updated residual r_{j+1}, the updated shadow residual r_{j+1}^*, the new search direction p_{j+1} and p_{j+1}^*, the 'shadow search direction'. We use equation (4.22) as a stopping criterion, where $||r_j||$ is computed directly. In theory the Bi-CG method can be used to solve the dual system too. To do this, we need to insert $x_{j+1}^* = x_j^* + \alpha_j p_j^*$ after line 5 of the algorithm.

Algorithm 4.10 Bi-Conjugate Gradient algorithm

```
1:  Compute $r_0 = b - Ax_0$,.
2:  Set $p_0 = r_0$, and $p_0^* = r_0^*$
3:  For $j = 0, 1, 2, \ldots, m$ **Do**
4:      $\alpha_j := (r_j, r_j^*)/(p_j, A^T p_j^*)$
5:      $x_{j+1} := x_j + \alpha_j p_j$
6:      $r_{j+1} := r_j - \alpha_j A p_j$
7:      $r_{j+1}^* := r_j^* - \alpha_j A^T p_j^*$
8:      **If** stop criterion satisfied
9:          Stop
10:     **EndIf**
11:     $\beta_j := (r_{j+1}, r_{j+1}^*)/(r_j, r_j^*)$
12:     $p_{j+1} := r_{j+1} + \beta_j p_j$
13:     $p_{j+1}^* := r_{j+1}^* + \beta_j p_j^*$
14: **EndDo**
15: $x = x_m$
```

The Bi-CG method is similar to the Bi-Lanczos method for systems of equations, since they both build two biorthogonal bases for the subspaces $\mathcal{K}_m$ and $\mathcal{L}_m$. Bi-CG can be derived from the Bi-Lanczos method just as CG can be derived from the Lanczos method [Saad, 2003, pp 196-200, 235]. Moreover, it is mathematically equivalent to the Bi-Lanczos method for systems of equations. Bi-CG builds the *LU* factorisation [Poole, 2006, pp. 178-184] of the tridiagonal matrix T_m from the Lanczos algorithm ($T_m = L_m U_m$, with L_m a lower triangular matrix and U_m an upper triangular matrix). For symmetric matrices the Bi-CG generates the same solution as CG.

The method in which Bi-CG approximates the solution differs from the Bi-Lanczos method, just as CG differs from the Lanczos method (see section 4.2.4). The Bi-Lanczos method needs all the Lanczos vectors v_j for calculating the approximate solution: $x_j = x_0 + V_j y_j$. In each iteration an extra basis vector becomes available to approximate the solution more accurately. CG avoids this by updating the solution in each iteration: $x_{j+1} = x_j + \alpha_j p_j$. Once an iteration is complete, the previous v_j are not needed anymore.

4.2.8 The Bi-Conjugate Residual (Bi-CR) method

The Bi-CG method is an extension of CG for nonsymmetric matrices. Similarly, the Bi-Conjugate Residual method (Bi-CR) was recently suggested as an extension to CR for non-symmetric matrices [Sogabe et al., 2009]. Bi-CR is an oblique projection method which uses A^T in the left subspace:

$$\begin{aligned}\mathcal{K}_m(A, r_0) &= span\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\} \\ \mathcal{L}_m(A^T, r_0^*) &= span\{r_0^*, A^T r_0^*, (A^T)^2 r_0^*, \ldots, (A^T)^{m-1} r_0^*\}.\end{aligned}$$

Algorithm 4.11 Bi-Conjugate Residual algorithm

```
1: Compute $r_0 = b - Ax_0$ and set $p_0 = r_0$ and $p_0^* = r_0^*$.
2: For $j = 0, 1, 2, \ldots, m$ **Do**
3:     $\alpha_j := (r_j, A^T r_j^*)/(p_j, A^T(A^T p_j^*))$
4:     $x_{j+1} := x_j + \alpha_j p_j$
5:     $r_{j+1} := r_j - \alpha_j A p_j$
6:     $r_{j+1}^* := r_j^* - \alpha_j A^T p_j^*$
7:     **If** stop criterion satisfied
8:         Stop
9:     **EndIf**
10:    $\beta_j := (r_{j+1}, A^T r_{j+1}^*)/(r_j, A^T r_j^*)$
11:    $p_{j+1} := r_{j+1} + \beta_j p_j$
12:    $p_{j+1}^* := r_{j+1}^* + \beta_j p_j *$
13: **EndDo**
14: $x = x_m$
```

A possible algorithm for Bi-CR can be found in algorithm 4.11. When we look closely at the algorithm, we find that it is almost similar to the algorithm of Bi-CG. This is no coincidence, since Bi-CR can be obtained by multiplying the initial shadow residual r_0^* in the Bi-CG method by A^T, that is: $r_0^* \mapsto A^T r_0^*$. This can be seen as follows.

Suppose s_{j+1} and t_{j+1} are polynomials of degree $(j+1)$. We can write (in a similar fashion as in section 3.3) the shadow residual in the Bi-CG method as a polynomial in A^T multiplied by r_0^*:

$$r_j^* = s_{j+1}(A^T) r_0^*.$$

If $r_0^* \mapsto A^T r_0^*$ in the Bi-CG method, then

$$r_j^* = s_{j+1}(A^T) A^T r_0^*. \tag{4.31}$$

Since $p_0^* = r_0^*$, we have $p_0^* \mapsto A^T p_0^*$. In a similar fashion as for the shadow residual, we find:

$$p_j^* = t_{j+1}(A^T) A^T p_0^*. \tag{4.32}$$

From equation (4.31) and (4.32), is it clear that all the shadow residuals and the 'shadow search vectors' in the Bi-CG algorithm have been multiplied by A^T. However, this will give an algorithm that is exactly the same as the algorithm of the Bi-CR method. Hence, we have:

$$\mathcal{L}_m^{Bi-CR}(A^T, r_0^*) = A^T \cdot \mathcal{L}_m^{Bi-CG}(A^T, r_0^*) = \mathcal{L}_m^{Bi-CG}(A^T, A^T r_0^*).$$

5. Comparing CG-type methods with Lanczos-type methods

Section 4.2.4 and section 4.2.7 mentioned that the Lanczos method and the Bi-Lanczos method for systems of equations were mathematically equivalent to the CG method and Bi-CG method respectively. Being mathematically equivalent, means that the convergence behaviour of the (Bi)-CG method and the (Bi)-Lanczos method is the same. Hence, the residual vector produced by both methods should be the same. In the next two sections we will discuss four examples which show this.

5.1 Comparing CG with Lanczos for linear systems

In this section we will discuss the convergence behaviour of the CG method and the Lanczos method for linear systems. We use the Matlab code in appendix C.1 and discuss two examples of real SPD matrices: the two-dimensional Poisson matrix and a 'Moler' matrix.

5.1.1 Example 1 - The two-dimensional Poisson matrix

Consider the two-dimensional Poisson equation, which has a broad applicability in engineering.

$$-\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}\right)\varphi(x,y)=f(x,y) \tag{5.1}$$

We discretise this equation using finite difference method with the 5-point operator on an n-by-n mesh. This results in the *Poisson* matrix of size n^2, which is a pentadiagonal SPD matrix. (5.2) gives an example for $n=3$:

$$A=\begin{pmatrix} 4 & -1 & 0 & -1 & & & & & \\ -1 & 4 & -1 & 0 & -1 & & & O & \\ 0 & -1 & 4 & 0 & 0 & -1 & & & \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & & \\ & -1 & 0 & -1 & 4 & -1 & 0 & -1 & \\ & & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ & & & -1 & 0 & 0 & 4 & -1 & 0 \\ & O & & & -1 & 0 & -1 & 4 & -1 \\ & & & & & -1 & 0 & -1 & 4 \end{pmatrix}. \tag{5.2}$$

A has eigenvalues of the form $4-2\cos(\frac{\pi i}{n+1})-2\cos(\frac{\pi j}{n+1})$ with $i=1:n$ and $j=1:n$. Note that the elements on the main diagonal are always 4 and that all the elements on the n-th subdiagonal and the n-th superdiagonal are -1. Most of the elements on the first subdiagonal and the first superdiagonal are -1, except for the elements that correspond to points on the boundary of the mesh.

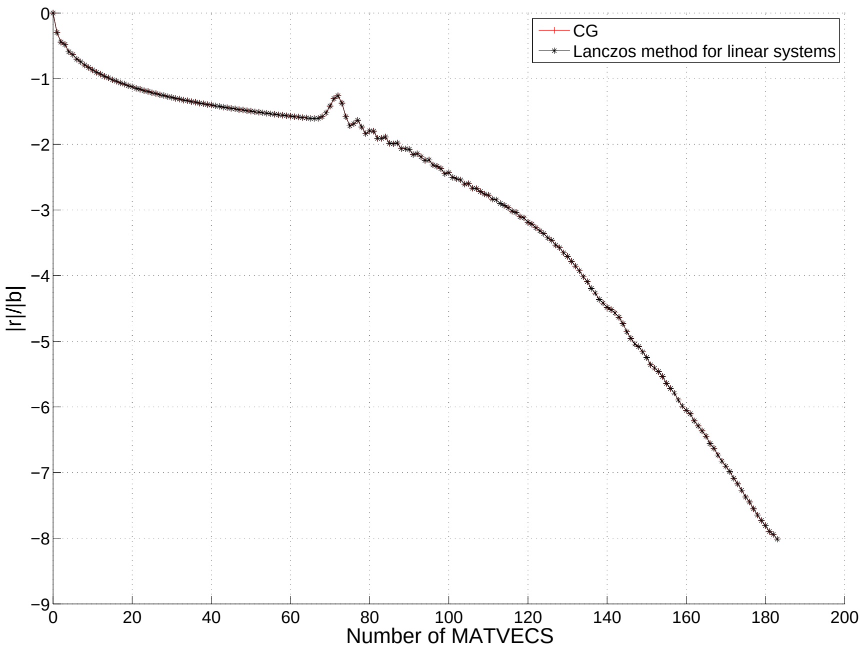

Figure 5.1: Convergence behaviour of CG and Lanczos for the Poisson matrix with $n = 100$

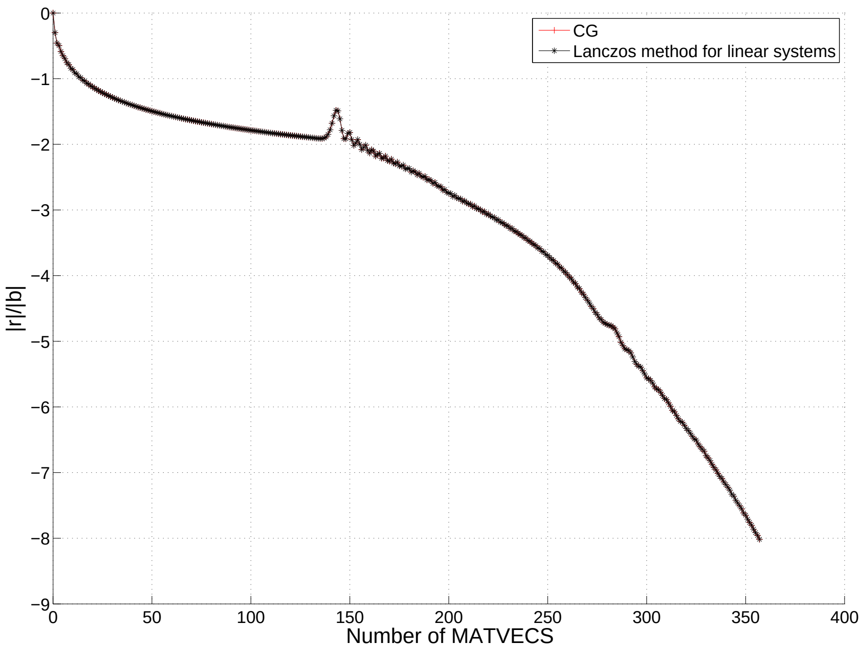

Figure 5.2: Convergence behaviour of CG and Lanczos for the Poisson matrix with $n = 200$

In Figure 5.1 and Figure 5.2 we see the convergence behaviour of CG and the Lanczos method for linear systems for the Poisson matrix for $n = 100$ and $n = 200$. On the horizontal axis we see the number of matrix-vector operations (MATVECS) and on the vertical axis we see the scaled residual norm $||r||/||b||$ in powers of ten. For CG and the Lanczos method the number of MATVECS is equal to the number of iterations. Both algorithms should stop if $||r|| < ||b|| \cdot 10^{-8}$ and we see that this happens after 183 MATVECS for $n = 100$ and approximately 357 MATVECS for $n = 200$. Moreover, we see that both convergence curves overlap exactly, meaning that the residual vectors are the same.

Matrix	**MATVECS**	**CPU time**
Poisson 100	183	0.0624s
Poisson 200	357	0.5460s

Table 5.1: Convergence for CG

Matrix	**MATVECS**	**CPU time**
Poisson 100	183	0.1872s
Poisson 200	357	1.4976s

Table 5.2: Convergence for Lanczos

In Table 5.1 and Table 5.2 we summarise the results. Although both methods need the same number of MATVECS, we see that the CG method computes a solution more time-efficiently.

5.1.2 Example 2 - The Moler matrix

The Moler matrix can be invoked from Matlab's 'gallery' by $A = \text{gallery}('\text{Moler}', n, \alpha)$. The Moler matrix (named after Cleve Barry Moler, the inventor of Matlab) is a symmetric positive definite n-by-n matrix which can be written as U^TU, where U is an upper triangular matrix with ones on the diagonal and α on the first n superdiagonals. One of the eigenvalues is small. In this example, we have chosen $\alpha = -1$, the default value. Furthermore $A(i,j) = \min(i,j) - 2$ and $A(i,i) = i$. For $n = 9$, this results in the following matrix:

$$A = \begin{pmatrix} 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 3 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & 0 & 1 & 4 & 2 & 2 & 2 & 2 & 2 \\ -1 & 0 & 1 & 2 & 5 & 3 & 3 & 3 & 3 \\ -1 & 0 & 1 & 2 & 3 & 6 & 4 & 4 & 4 \\ -1 & 0 & 1 & 2 & 3 & 4 & 7 & 5 & 5 \\ -1 & 0 & 1 & 2 & 3 & 4 & 5 & 8 & 6 \\ -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 9 \end{pmatrix}. \tag{5.3}$$

In Figure 5.3 and Figure 5.4 we see the convergence behaviour of CG and the Lanczos method for linear systems for the Moler matrix for $n = 500$ and $n = 1000$. On the horizontal axis we see the number of matrix-vector operations (MATVECS) and on the vertical axis we see the scaled residual norm $||r||/||b||$ in powers of ten. Just as in the previous section, we see both algorithms stop if $||r|| < ||b|| \cdot 10^{-8}$. However, the Lanczos method for systems of equations needs more MATVECS than the CG method. If $n = 500$, the CG method is finished after 58 MATVECS and the Lanczos methods for systems of equations is finished after 60 MATVECS. For $n = 1000$ this is 83 MATVECS and 93 MATVECS respectively.

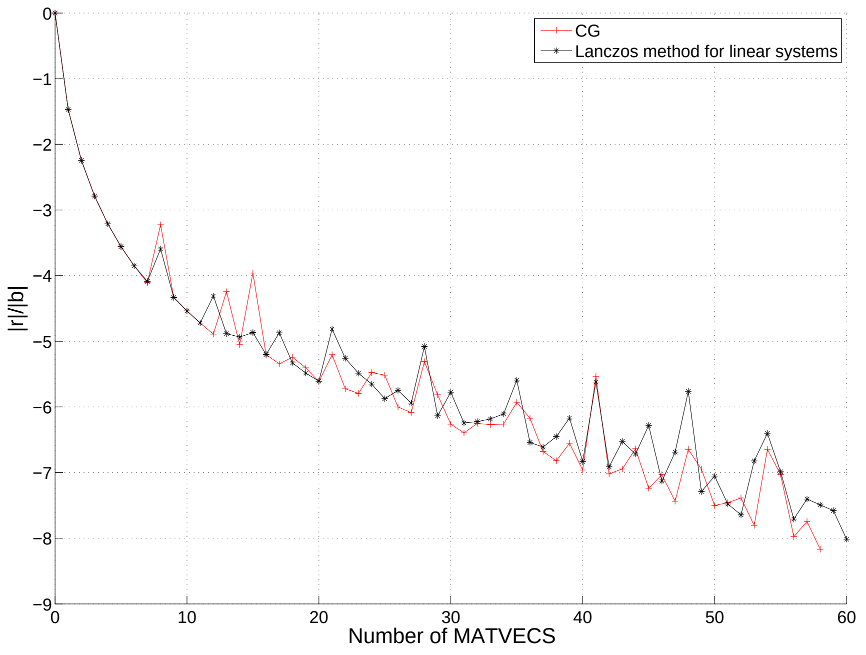

Figure 5.3: Convergence behaviour of CG and Lanczos for the Moler matrix with $n = 500$

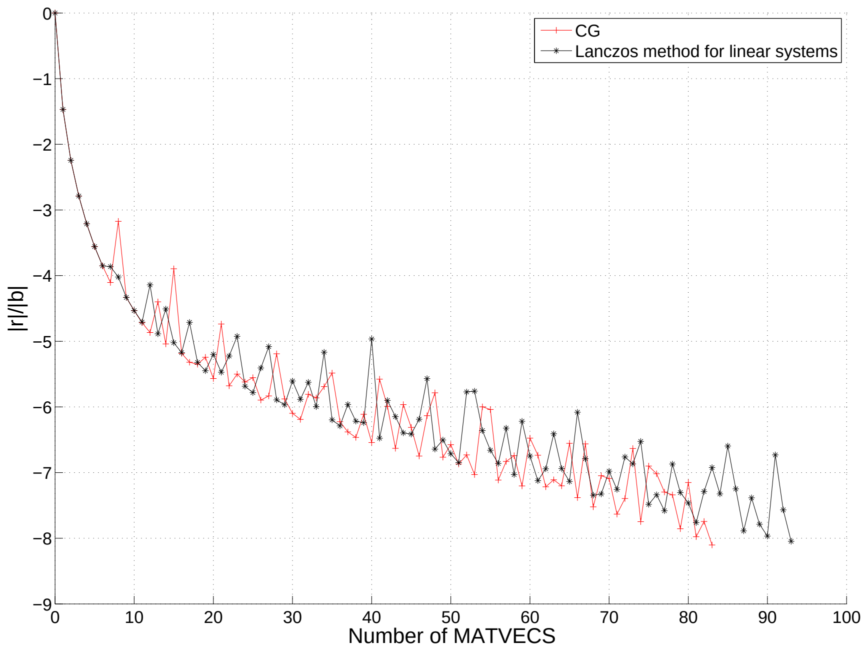

Figure 5.4: Convergence behaviour of CG and Lanczos for the Moler matrix with $n = 1000$

The convergence behaviour of both methods follows the same trend, but the mathematical rounding errors cause some discrepancies. These errors also result in a slightly different number of MATVECS in both methods. However, since the difference is small, we can conclude that the residual vectors of both methods are equal.

Matrix	**MATVECS**	**CPU time**
Moler 500	58	0.0156s
Moler 1000	83	0.2028s

Table 5.3: Convergence for CG

Matrix	**MATVECS**	**CPU time**
Moler 500	60	0.0312s
Moler 1000	93	0.3120s

Table 5.4: Convergence for Lanczos

In Table 5.3 and Table 5.4 we summarise the results. Although both methods approximately need the same number of MATVECS, we see that the CG method computes a solution more time-efficiently.

5.2 Comparing Bi-CG with Bi-Lanczos for linear systems

In this section we will discuss the convergence behaviour of the Bi-CG method and the Bi-Lanczos method for linear systems. We use two examples of general real matrices: The Rutishauser matrix and the Hanowa matrix, both from Matlab's 'gallery' function. The Bi-CG method and the Bi-Lanczos method for linear systems of equations both use two MATVECS per iteration, so the number of iterations can be found by dividing the number of MATVECS by 2. We use the Matlab code in appendix C.2.

5.2.1 Example 3 - The Rutishauser matrix

A Toeplitz matrix is a pentadiagonal n-by-n matrix with the first two subdiagonals and superdiagonals are nonzero. When the second subdiagonal is 1, the first subdiagonal is -10, the main diagonal is 0, the first superdiagonal is 10 and the second superdiagonal is 1, we call this matrix a Rutishauser matrix, named after the Swiss mathematician Heinz Rutishauser. It can be invoked from Matlab's gallery by $A = \text{gallery}('\text{toeppen}', n, 1, -10, 0, 10, 1)$. This matrix has complex eigenvalues lying approximately on the line segment $2\cos(2t) + 20i\sin(t)$. An example for $n = 9$ is given in (5.4).

$$A = \begin{pmatrix} 0 & 10 & 1 & & & & & & \\ -10 & 0 & 10 & 1 & & & & & \\ 1 & -10 & 0 & 10 & 1 & & & O & \\ & 1 & -10 & 0 & 10 & 1 & & & \\ & & 1 & -10 & 0 & 10 & 1 & & \\ & & & 1 & -10 & 0 & 10 & 1 & \\ & O & & & 1 & -10 & 0 & 10 & 1 \\ & & & & & 1 & -10 & 0 & 10 \\ & & & & & & 1 & -10 & 0 \end{pmatrix}. \tag{5.4}$$

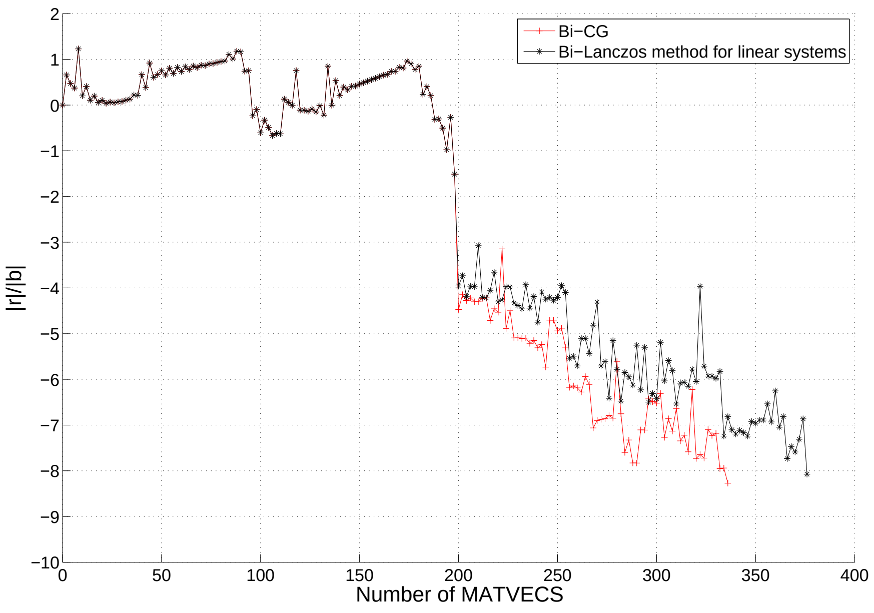

Figure 5.5: Convergence behaviour of Bi-CG and Bi-Lanczos for the Rutishauser matrix with $n = 100$

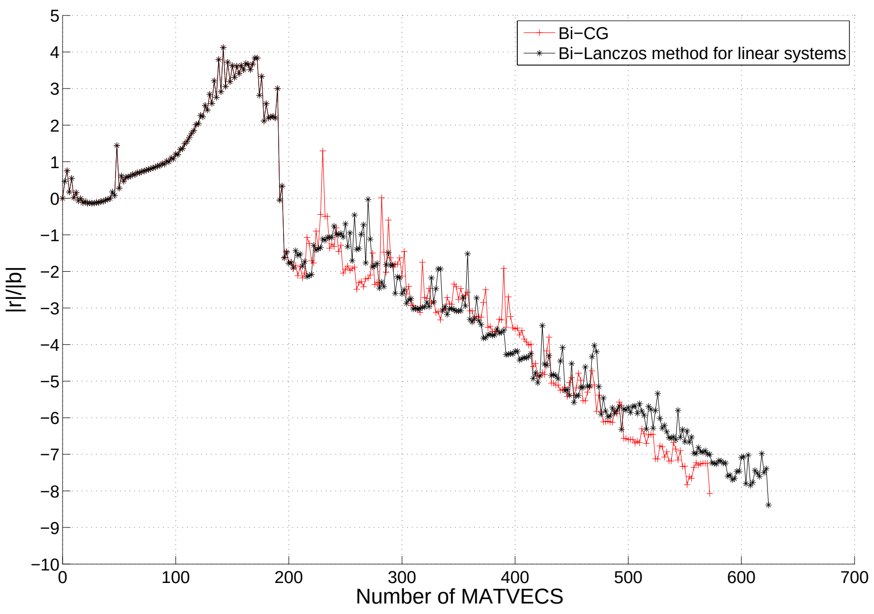

Figure 5.6: Convergence behaviour of Bi-CG and Bi-Lanczos for the Rutishauser matrix with $n = 200$

In Figure 5.5 and Figure 5.6 we see the convergence behaviour of Bi-CG and the Bi-Lanczos method for linear systems for the Rutishauser matrix for $n = 100$ and $n = 200$. On the horizontal axis we see the number of matrix-vector operations (MATVECS) and on the vertical axis we see the scaled residual norm $||r||/||b||$ in powers of ten.

Until about 200 MATVECS (100 iterations), the convergence behaviour of both methods is exactly the same, since the convergence curves overlap. After this the Bi-CG method performs better than the Bi-Lanczos method for both $n = 100$ and $n = 200$. For $n = 100$, the Bi-CG method is finished after 336 MATVECS, while the Bi-Lanczos method is finished after 376 MATVECS. For $n = 200$, this is 572 MATVECS for the Bi-CG method and 624 MATVECS for the Bi-Lanczos method. If we take n larger than 250 we see that both methods fail to find a solution. Apparently, the Bi-CG and Bi-Lanczos method do not perform well for the Rutishauser matrix.

Matrix	**MATVECS**	**CPU time**
Rutishauser 100	336	0.0156s
Rutishauser 200	572	0.0624s

Table 5.5: Convergence for Bi-CG

Matrix	**MATVECS**	**CPU time**
Rutishauser 100	376	0.0780s
Rutishauser 200	624	0.2340s

Table 5.6: Convergence for Bi-Lanczos

In Table 5.5 and Table 5.6 we summarise the results. We see that the CG method computes a solution more time-efficiently while also needing fewer MATVECS. The number of MATVECS differs significantly and this might be caused by rounding errors.

5.2.2 Example 4 - The Hanowa matrix

The Hanowa matrix is invoked from Matlab's gallery by $A = \text{gallery}('hanowa', n, d)$, where n must be an even integer. It produces an n-by-n block 2×2 matrix of the form

$$A = \begin{pmatrix} d \cdot \text{eye}(m) & -\text{diag}(1:m) \\ -\text{diag}(1:m) & d \cdot \text{eye}(m) \end{pmatrix}, \tag{5.5}$$

with $m = n/2$. The Hanowa matrix has complex eigenvalues which lie on the vertical line $d \pm ki$, with $1 \leq k \leq m$. For $n = 8$ and $d = -1$, we have the following matrix:

$$A = \begin{pmatrix} -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & -3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & -4 \\ -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -3 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -4 & 0 & 0 & 0 & -1 \end{pmatrix}. \tag{5.6}$$

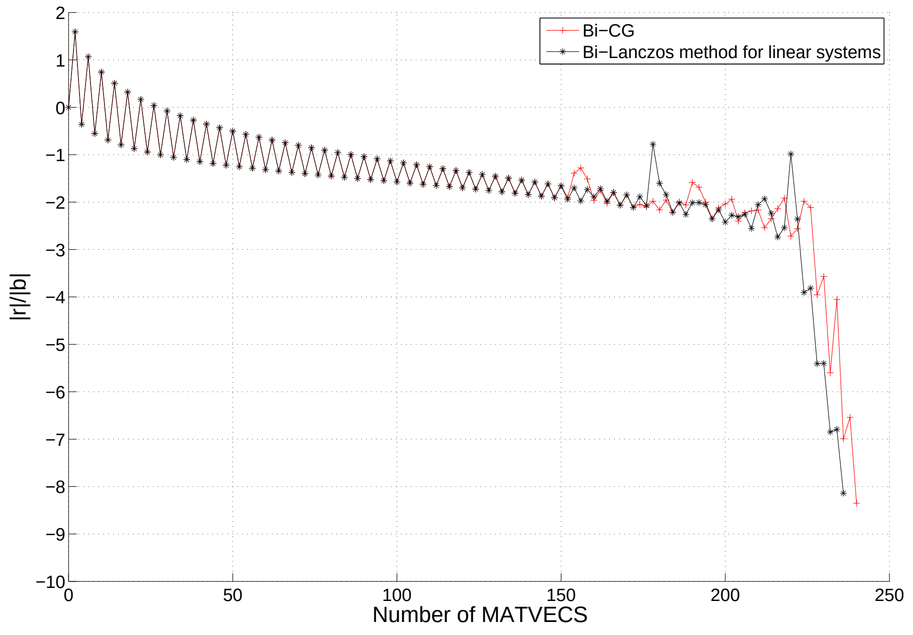

Figure 5.7: Convergence behaviour of Bi-CG and Bi-Lanczos for the Hanowa matrix with $n = 100$

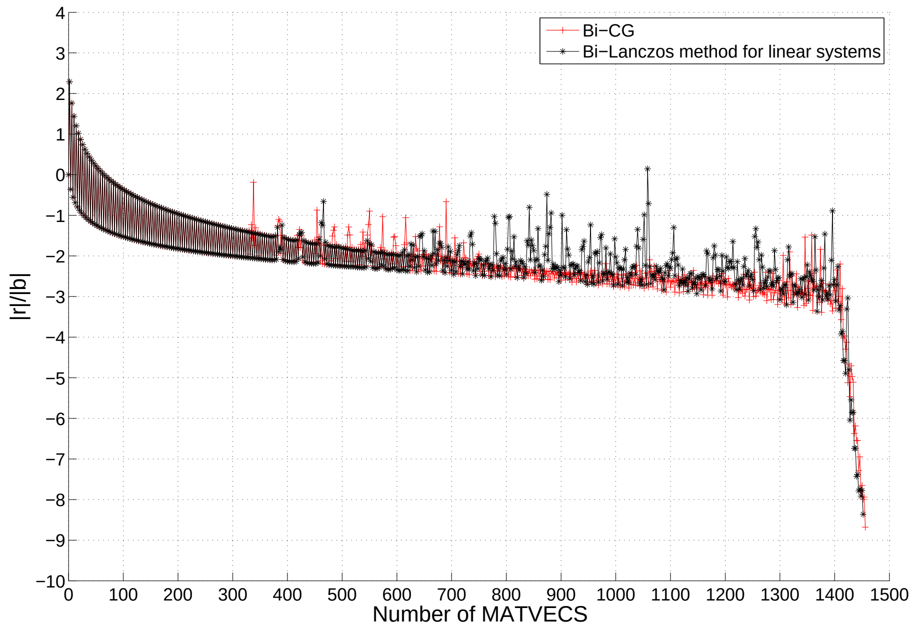

Figure 5.8: Convergence behaviour of Bi-CG and Bi-Lanczos for the Hanowa matrix with $n = 500$

In Figure 5.7 and Figure 5.8 we see the convergence behaviour of Bi-CG and the Bi-Lanczos method for linear systems for the Hanowa matrix for $n = 100$ and $n = 500$. On the horizontal axis we see the number of matrix-vector operations (MATVECS) and on the vertical axis we see the scaled residual norm $||r||/||b||$ in powers of ten. For $n = 100$ we see that both convergence curves coincide until approximately 150 MATVECS. After this, the convergence becomes a little bit more irregular, but both convergence curves follow the same trend and the stopping criterion ($||r|| < ||b|| \cdot 10^{-8}$) is satisfied after 240 MATVECS for Bi-CG and after 236 MATVECS for Bi-Lanczos.

For $n = 500$ we see the same results as for $n = 100$. Both methods have the same residuals until 325 MATVECS. After this, convergence follows the same trend, but the residuals differ slightly. The stopping criterion is satisfied after approximately 1450 MATVECS. The Bi-Lanczos method is finished a few MATVECS earlier. The discrepancies are caused by rounding errors. We can conclude that the residual vectors of both methods are equal.

Matrix	**MATVECS**	**CPU time**
Hanowa 100	240	0.0312s
Hanowa 500	1456	2.8080s
Hanowa 1000	3000	34.882s

Table 5.7: Convergence for Bi-CG

Matrix	**MATVECS**	**CPU time**
Hanowa 100	236	0.0780s
Hanowa 500	1452	2.3868s
Hanowa 1000	2992	28.782s

Table 5.8: Convergence for Bi-Lanczos

In Table 5.3 and Table 5.4 we summarise the results. We see that for $n = 100$ the Bi-CG method is faster than the Lanczos method. For growing n, we see that the Bi-Lanczos method finds a solution quicker. To make this plausible, we included the number of MATVECS and the CPU time for $n = 1000$.

6. The IDR(s) method

Krylov subspace methods are widely used to solve linear systems in the form $Ax = b$. In the case A is symmetric and positive definite, the Conjugate Gradient method (see section 4.2.4 is often preferred. It combines optimal minimisation of the residual with short recurrences. It computes a solution in N steps, where N is the number of matrix-vector multiplications. Unfortunately it is not possible to find a method that uses short recurrences and minimises the residual over some norm for a general matrix A [Faber and Manteuffel, 1984].

The search for such al algorithm for general A has taken two approaches. In the first approach, mathematicians were looking for methods in which the short recurrence requirement was removed. GMRES is the most popular member of this family of methods. In the second approach, mathematicians focussed on short recurrence methods without the optimality condition. The archetype of this method is the Bi-CG method (see section 4.2.7). However Bi-CG requires twice the work as CG. Other method, such as CGS [Sonneveld, 1989] and Bi-CGSTAB [Van der Vorst, 1992] have been developed in order to overcome this problem, but all these methods were more or less based on the Bi-CG method. There is however no reason to believe that a different approach cannot yield faster methods.

This is where the Induced Dimension Reduction (IDR) method comes in. It was first proposed by Peter Sonneveld in 1980 [Wesseling and Sonneveld, 1980]. IDR is a short recurrence method and computes the solution in at most $2N$ matrix-vector multiplications, This makes it at least at fast as the Bi-CG method. Over the years, IDR was completely overshadowed by CGS and Bi-CGSTAB, but in recent years their had been renewed interest in IDR. One of the new family of methods that was developed was IDR(s) with $s \in \mathbb{N}$.

6.1 Derivation of the IDR(s) algorithm

The IDR(s) method is based on the IDR theorem, which was originally published in [Wesseling and Sonneveld, 1980, p. 550]. The following theorem is an extension to complex matrices.

Theorem 6.1 (IDR Theorem).
Let A be any matrix in $\mathbb{C}^{N\times N}$, let v_1 be any nonzero vector in $\mathcal{C}^N$ and let $\mathcal{G}_0$ be the full Krylov space $\mathcal{K}_N(A, v_1)$. Let $\mathcal{S}$ denote any (proper) subspace of $\mathbb{R}^N$ such that $\mathcal{S}$ and $\mathcal{G}_0$ do not share a nontrivial invariant subspace of A, and define the sequence $\mathcal{G}_j,\ j = 1, 2, \ldots$ as

$$\mathcal{G}_j = (I - \omega_j A)(\mathcal{G}_{j-1} \cap \mathcal{S}), \tag{6.1}$$

where the ω_j's are nonzero scalars. The following holds:

(i) $\mathcal{G}_j \subset \mathcal{G}_{j-1}\ \forall j > 0$;

(ii) $\mathcal{G}_j = \{0\}$ for some $j \leq N$.

Proof. See [Sonneveld and Van Gijzen, 2008]

The main idea of the IDR(s) method is to generate residuals r_n that are forced to be in the subspaces $\mathcal{G}_j$, where j is nondecreasing for increasing n. The subspaces $\mathcal{G}_j$ are also called the *Sonneveld spaces*. The first part of the IDR theorem tells us that $\mathcal{G}_j \subset \mathcal{G}_{j-1}$, so that the dimension of the Sonneveld spaces reduces in each iteration. Hence the name of the method. Using the second part of the IDR theorem, the j-th residual must eventually be in $\mathcal{G}_j = \{0\}$ after at most N dimension reduction steps. If the residual is zero, we have found the solution to the system $Ax = b$.

According to Sonneveld and Van Gijzen [Sonneveld and Van Gijzen, 2008] the residuals of the IDR(s) method satisfy

$$r_{m+1} = r_m - \alpha A v_m - \sum_{l=1}^{\hat{l}} \gamma_l \Delta r_{m-l}, \tag{6.2}$$

where the γ_l and α are scalars in $\mathbb{C}$ and v_n is any computable vector in $\mathcal{K}_m(A, r_0) \backslash \mathcal{K}_{m-1}(A, r_0)$. The integer $\hat{l}$ is the depth of the recursion.

Recall that we want the residual to be in the Sonneveld spaces. The residual r_{m+1} is in $\mathcal{G}_{j+1}$ if

$$r_{m+1} = (I - \omega_{m+1} A)\, v_m \qquad \text{with } v_m \in \mathcal{G}_j \cap \mathcal{S}, \tag{6.3}$$

which can be seen directly from the definition of the Sonneveld spaces. Now define the $(\hat{l} \times 1)$ vector $c = (\gamma_1\ \gamma_2\ \ldots\ \gamma_{\hat{l}})$ and the $(N \times s)$ matrix $\Delta R_m = (\Delta r_{m-1}\ \Delta r_{m-2}\ \ldots\ \Delta r_{m-s})$. We choose v_m to be

$$v_m = r_m - \sum_{l=1}^{\hat{l}} \gamma_l \Delta r_{m-l} = r_m - \Delta R_m c, \tag{6.4}$$

so that r_{m+1} satisfies equation (6.2) (with $\alpha = \omega_{j+1}$)

Now define an $(N \times s)$ matrix $P = (p_1\ p_2\ \ldots\ p_s)$ with $p_{ij} \in \mathbb{C}$. Without loss of generality, we assume that the subspace $\mathcal{S}$ is the left nullspace of P, that is $\mathcal{S} = \{x \in \mathbb{C}^N\ : A^* x = 0\}$. Since $v_m \in \mathcal{S}$, we have:

$$P^* v_m = 0. \tag{6.5}$$

Substituting equation (6.4) in (6.5) yields:

$$(P^* \Delta R_m)\, c = P^* r_m. \tag{6.6}$$

Having $\hat{l}$ unknowns and s equations, this system is in general uniquely solvable for c if $\hat{l} = s$ See also Figure 6.1.

With the vector c we can compute v_n and r_{m+1}.We know that $r_{m+1} \in \mathcal{G}_{j+1}$. After updating ΔR_m to ΔR_{m+1}, we start a new iteration in which we calculate v_{m+1} and r_{m+2}. Since we have that $v_{m+1} \in \mathcal{G}_j \cap \mathcal{S}$, we can conclude from equation (6.3) that also $r_{m+2} \in \mathcal{G}_{j+1}$. We repeat these steps $s + 1$ times, until the vectors $r_{m+1}, \ldots, r_{m+s}$ are in $\mathcal{G}_{j+1}$. Since we now have enough vectors in $\mathcal{G}_{j+1}$. the next vector, r_{m+s+1}, will be in $\mathcal{G}_{j+2}$.

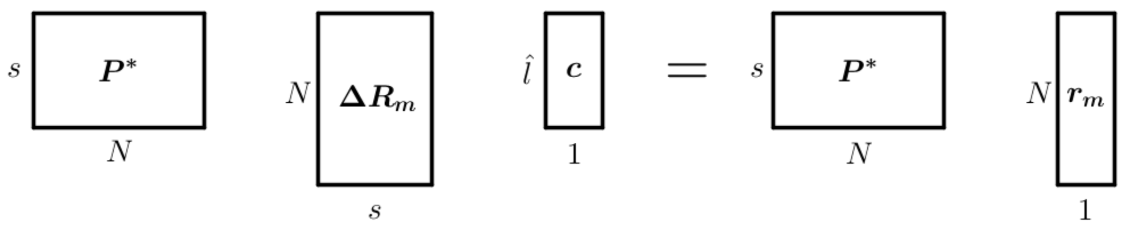

Figure 6.1: Solving $(P^* \Delta R_m)\, c = P^* r_m$

Of course, we also need to find an expression for the solution vector. We can easily find one using equation (6.2) (with $\alpha_m = \omega_{j+1}$):

$$\begin{aligned} r_{m+1} &= r_m - \omega_{j+1} A v_m - \sum_{l=1}^{\hat{l}} \gamma_l \Delta r_{m-l} && (6.7) \\ &= r_m - \omega_{j+1} A v_m - \Delta R_{m-l} c. && (6.8) \end{aligned}$$

Using $r_m = b - Ax_m$, cancelling the b's on both side and multiplying with A^{-1}, we find:

$$\begin{aligned} x_{m+1} &= x_m + \omega_{j+1} v_m - \sum_{l=1}^{\hat{l}} \gamma_l \Delta x_{m-l} && (6.9) \\ &= x_m + \omega_{j+1} v_m - \Delta X_{m-l} c. && (6.10) \end{aligned}$$

Equation (6.7) and (6.9) form the basis of IDR(s). One possible algorithm can be seen in algorithm 6.1.

First we have to initialise the algorithm by choosing a matrix P and computing r_0. Next, we have to build ΔR_{m+1} and Δx_{m+1}, which we need for building the spaces $\mathcal{G}_{j+1}$. The algorithm carries out the loop $s+1$ times in order to find $s+1$ vectors for $\mathcal{G}_{j+1}$. In the calculation of the first residual in $\mathcal{G}_{j+1}$, we can choose ω_{j+1} freely, but often a value is chosen that minimises $||r_{m+1}||$. For the calculation of the subsequent residuals in $\mathcal{G}_{j+1}$, the same value for ω_{j+1} must be used.

By substituting line 25 into line 30 / 34 and line 30 / 34 in line 38, we obtain equation (6.10). In a similar fashion, we can deduct equation (6.8) from the algorithm. For k=0, we substitute line 25 into line 28, line 28 into line 29 (to choose the omega's), line 29 into line 30 and line 30 into line 38. For $0 < k \leq s$, we substitute line 25 into line 34, line 34 into line 35 and line 35 into line 37. When $s+1$ vectors have been computed, the algorithm checks the stopping condition and it starts over again if the stopping condition is not satisfied.

6.2 Performance of the IDR(s) method

The IDR theorem predicts that dimension reduction will take place, but not by how much. The extended IDR theorem gives information about the rate of convergence.

Algorithm 6.1 IDR(s)

```
1: Require $A \in \mathbb{C}^{N\times N}$; $x_0, b \in \mathcal{C}^N$; $P \in \mathbb{C}^{N\times s}$; $TOL \in (0,1)$; $MAXIT > 0$
2: Ensure $x_m$ such that $||b - Ax_m|| < TOL$
3:     $\{Initialisation\}$
4:     Calculate $r_0 = b - Ax_0$;
5:
6:     $\{Apply\ s\ minimum\ norm\ steps\ to\ build\ enough\ vectors\ in\ \mathcal{G}_0\}$
7:     For $m = 0$ to $s-1$ Do
8:         $v = Ar_m$ ;
9:         $\omega = \left(v^H r_m\right) / \left(v^H v\right)$;
10:        $\Delta x_m = \omega r_m$;
11:        $\Delta r_m = -\omega v$;
12:        $r_{m+1} = r_m + \Delta r_m$;
13:        $x_{m+1} = x_m + \Delta x_m$;
14:    EndFor
15:    $\Delta R_{m+1} = (\Delta r_m \ldots \Delta r_0)$;
16:    $\Delta X_{m+1} = (\Delta x_m \ldots \Delta x_0)$;
17:
18:    $\{Building\ \mathcal{G}_j\ spaces\ for\ j = 1, 2, 3, \ldots\}$
19:    $m = s$
20:    $\{Loop\ over\ \mathcal{G}_j\ spaces\}$
21:    While $||r_m|| > TOL$ and $m < MAXIT$ Do
22:        $\{Loop\ inside\ \mathcal{G}_j\ spaces\}$
23:        For $k = 0$ to $s$ Do
24:            Solve $c$ from $P^H \Delta R_m c = P^H r_m$;
25:            $v = r_m - \Delta R_m c$;
26:            If $k = 0$ then
27:                $\{Entering\ \mathcal{G}_{j+1}\}$
28:                $t = Av$;
29:                $\omega = \left(t^H v\right) / \left(t^H t\right)$;
30:                $\Delta x_m = -\Delta X_m c + \omega v$;
31:                $\Delta r_m = -\Delta R_m c - \omega t$;
32:            else
33:                $\{Subsequent\ vectors\ in\ \mathcal{G}_{j+1}\}$
34:                $\Delta x_m = -\Delta X_m c + \omega v$;
35:                $\Delta r_m = -A \Delta x_m$;
36:            End if
37:            $r_{m+1} = r_m + \Delta r_m$;
38:            $x_{m+1} = x_m + \Delta x_m$;
39:            $m = m + 1$;
40:            $\Delta R_m = (\Delta r_{m-1}, \ldots, \Delta r_{m-s})$;
41:            $\Delta X_m = (\Delta x_{m-1}, \ldots, \Delta x_{m-s})$;
42:        End for
43:    End while
44: $x = x_m$
```

Theorem 6.2 (Extended IDR theorem).
Let A be any matrix in $\mathbb{C}^{N\times N}$, let $p_1, p_2, \ldots, p_s \in \mathbb{C}^N$ be linearly independent, let $P = [p_1, p_2, \ldots, p_s]$, let $\mathcal{G}_0 = \mathcal{K}_N(A, r_0)$ be the full Krylov space corresponding to A and the vector r_0 and let the sequence of spaces $\{\mathcal{G}_j,\ j = 1, 2, \ldots\}$ be defined by

$$\mathcal{G}_j = (I - \omega_j A)\left(\mathcal{G}_{j-1} \cap \mathcal{S}\right),$$

where ω_j are nonzero numbers, such that $I - \omega_j A$ is nonsingular. Let $\dim\left(\}_j\right) = d_j$; then the sequence $\{d_j,\ j = 1, 2, \ldots\}$ is monotonically nonincreasing and satisfies

$$0 \leq d_j - d_{j-1} \leq d_{j-1} - d_j \leq s.$$

Proof. See [Sonneveld and Van Gijzen, 2008]

From the extended IDR theorem, it is clear that the dimension reduction per step is between 0 and s. In practice, the reduction is s [Sonneveld and Van Gijzen, 2008]. If this is the case throughout the whole process, we have the so called *generic* case. As a consequence of the extended IDR theorem, we have the following corollary:

Corollary 6.3.
In the generic case IDR(s) requires at most $N + N/s$ matrix-vector multiplications to compute the exact solution.

7. Numerical experiments with IDR(s)

In this section we will give several examples that show the efficiency of the IDR(s) method. These examples are taken from section 6 of the article written by Valeria Simoncini and Daniel B. Szyld [Simoncini and Szyld, 2010]. In the examples we compare the convergence behaviour of Bi-CG, full GMRES, and IDR(s). Bi-CG uses short-recurrences at the cost of having to compute two matrix-vector operations (MATVECS) in each iteration. Full GMRES had the benefit that it minimises the residual in each iteration, at the cost of having to store all the previous orthonormal basis vectors to compute the next one. This makes GMRES very slow compared to Bi-CG and IDR(s). However, it is interesting to see how short-recurrence methods compare to full GMRES. For the plots of the convergence behaviour and the tables we used the implementation in appendix C.3.

7.1 Example 7.1 - The convection-diffusion equation

Consider the centered finite difference discretisation in the unit cube of the operator

$$L(u) = -\Delta u + \beta(u_x + u_y + u_z) = -\Delta u + \beta\nabla u$$

with homogeneous Dirichlet boundary conditions. This a convection-diffusion equation with convection term $\beta\nabla u$ and diffusion term $-\Delta u$. The reaction term ru vanishes, because we set the reaction parameter r to zero. In each direction of the unit cube, we take 20 internal nodes, which gives us a matrix of size $n = 8000$. This gives us a mesh size of $h = 1/(20 + 1)$.

In Figure 7.1 (for $\beta = 100$) we see that the convergence behaviour of IDR(2), IDR(4) and IDR(8) is much better than the convergence behaviour of Bi-CG. IDR(1) does also converge, but it needs more MATVECS and more time. For $\beta = 200$ (see Figure 7.2), we see that IDR(4) and IDR(8) still perform well, while IDR(2) now needs more MATVECS than Bi-CG and IDR(1). IDR(1) does not find a solution after 1000 iterations.

Method	MATVECS	CPU time
GMRES	71	0.6552s
Bi-CG	158	0.0936s
IDR(1)	183	0.0936s
IDR(2)	124	0.0624s
IDR(4)	97	0.1092s
IDR(8)	84	0.0780s

Table 7.1: Example 7.1 with $\beta = 100$

Method	MATVECS	CPU time
GMRES	93	0.9984s
Bi-CG	234	0.1248s
IDR(1)	-	-
IDR(2)	454	0.2496s
IDR(4)	171	0.1092s
IDR(8)	123	0.1092s

Table 7.2: Example 7.1 with $\beta = 200$

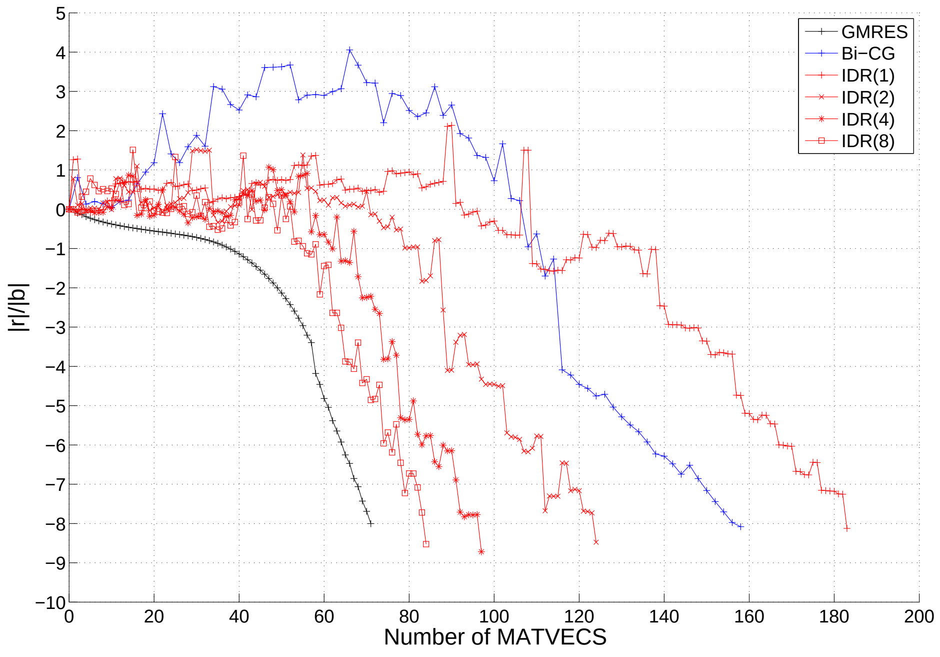

Figure 7.1: Convergence behaviour of Bi-CG, GMRES and IDR(s) with $\beta = 100$

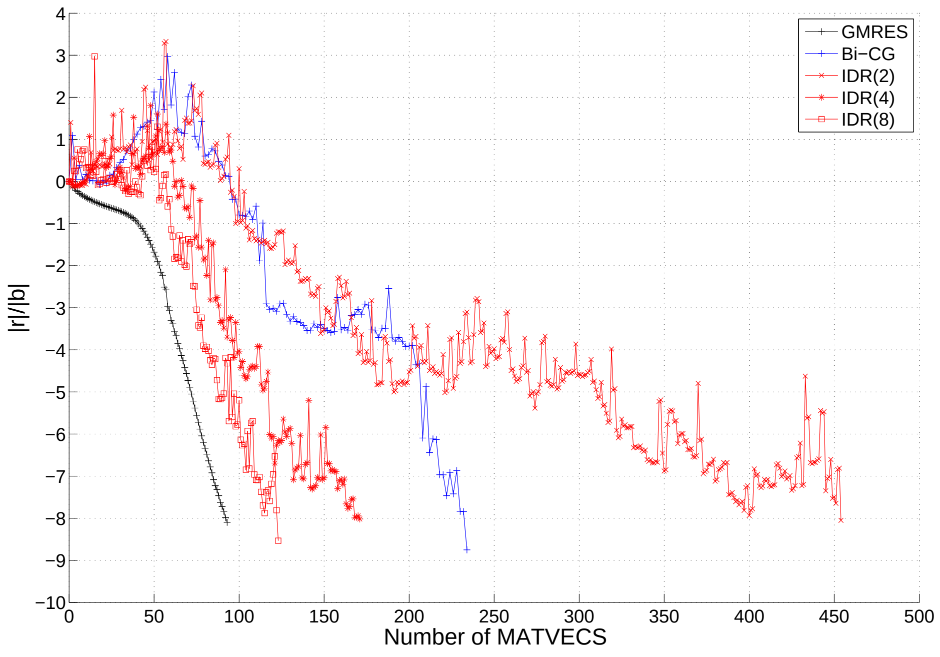

Figure 7.2: Convergence behaviour of Bi-CG, GMRES and IDR(s) with $\beta = 200$

If we take β even larger (e.g. $\beta = 500$), all four IDR(s) methods are outperformed by GMRES and Bi-CG. IDR(s) does not converge for $s = 1, 2, 4$, while IDR(8) needs more time MATVECS to find a solution. This is caused by the larger convection term, which makes the problem asymmetrical [Simoncini and Szyld, 2010, p.11].

In table 7.1 we see the exact number of iterations and the CPU-time for GMRES, Bi-CG and IDR(s) for $s = 1, 2, 4, 8$. Note that GMRES needs the fewest iterations (as it should, since it minimises the residual in each iteration), though it needs much more time to find a solution.

7.2 Example 7.2 - The `Sherman4` matrix

From the Matrix Market[1] we consider the `Sherman4` matrix[2]. This nonsysmmetric, real matrix of size 1104×1104 is used in the simulation of oil reservoirs. It has the following structure:

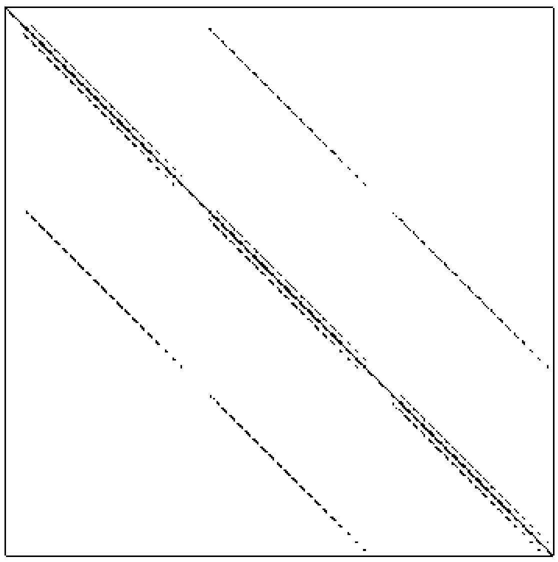

Figure 7.3: Structure of the `Sherman4` matrix

We now solve the system $Ax = b$, where b is A times a vector with ones in all its entries (so the solution will be a vector with ones in all its entries). We use a tolerance of 10^{-8} and we take 1000 as the maximum number of iterations.

In Figure 7.4 we can see the convergence behaviour of GMRES, Bi-CG and IDR(s). In this example IDR(4) and IDR(8) are very beneficial to use, while IDR(2) and IDR(1) also need less MATVECS to compute a solution.

[1]`http://math.nist.gov/MatrixMarket/`

[2]`http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/sherman/sherman4.html`

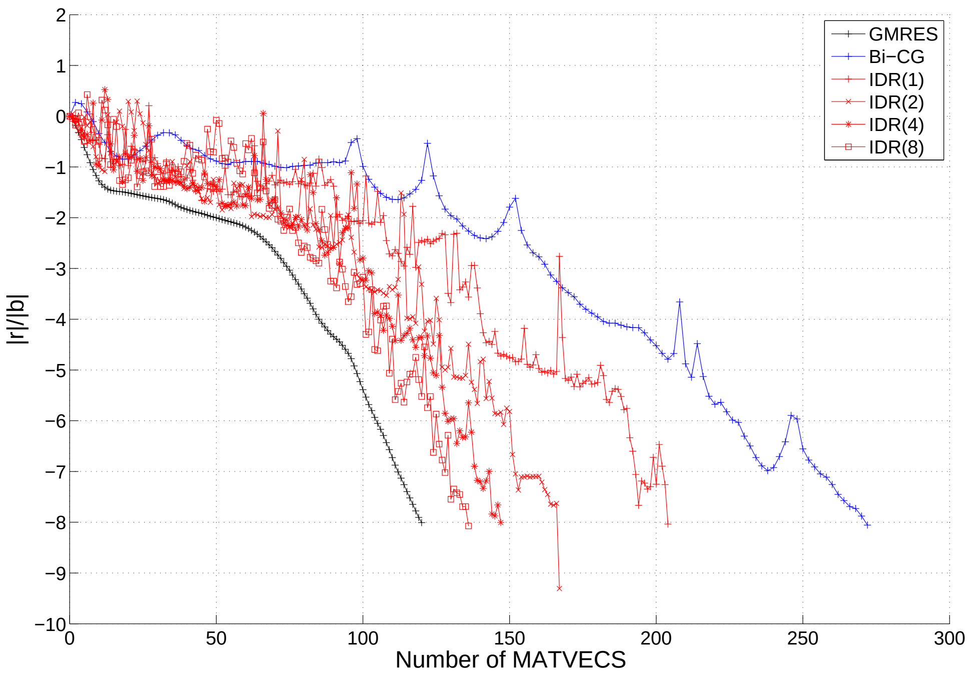

Figure 7.4: Convergence behaviour of the `Sherman4` matrix for GMRES, Bi-CG and IDR(s)

In Table 7.3 we can see the exact results. We see that the Bi-CG methods needs more time to compute a solution than the IDR(2) and the IDR(4) method, while it needs close to twice as much iterations. Furthermore, Bi-CG is as quick as IDR(1), but the latter needs fewer iterations. GMRES needs the fewest iterations, but note that IDR(8) only needs 16 MATVECS more (about 10% more) than full GMRES, while it is roughly four times as fast. For this example, the IDR(s) apparently works very well.

Method	**MATVECS**	**CPU time**
GMRES	120	0.2496s
Bi-CG	272	0.0468s
IDR(1)	204	0.0468s
IDR(2)	167	0.0156s
IDR(4)	147	0.0312s
IDR(8)	136	0.0624s

Table 7.3: Convergence behaviour of the `Sherman4` matrix for GMRES, Bi-CG and IDR(s)

7.3 Example 7.3 - The `add20` matrix

The `add20` matrix[3] is another matrix from the Matrix Market. It is a real nonsymmetric 2395×2395 matrix that is used in electronic circuit design. It has the following pattern:

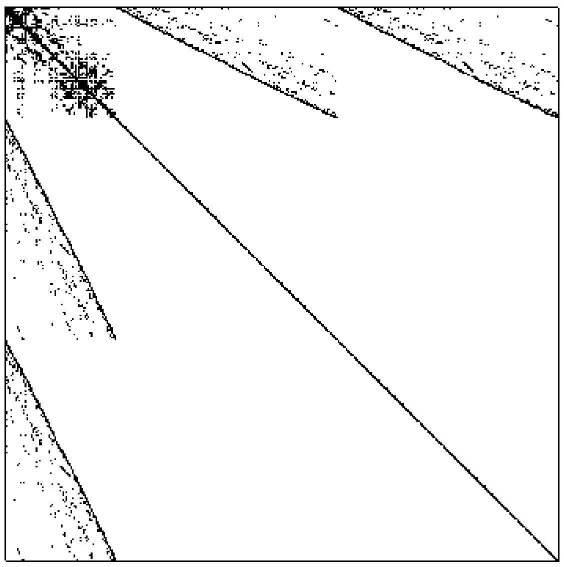

Figure 7.5: Structure of the `add20` matrix

We now solve the system $Ax = b$, where b is A times a vector with ones in all its entries (so the solution will be a vector with ones in all its entries). We use a tolerance of 10^{-8} and we take 1000 as the maximum number of MATVECS.

In Figure 7.6 we see the convergence behaviour for GMRES, Bi-CG and IDR(s) and in Figure 7.7 we see a close-up for IDR(4) and IDR(8). We see that IDR(1) has not converged after 1000 MATVECS and that IDR(2) performs poor compared to Bi-CG.

Method	MATVECS	CPU time
GMRES	295	2.0124s
Bi-CG	638	0.0936s
IDR(1)	-	-
IDR(2)	760	0.1872s
IDR(4)	484	0.1560s
IDR(8)	382	0.1248s

Table 7.4: Convergence behaviour of the `add20` matrix for GMRES, Bi-CG and IDR(s)

[3]`http://math.nist.gov/MatrixMarket/data/misc/hamm/add20.html`

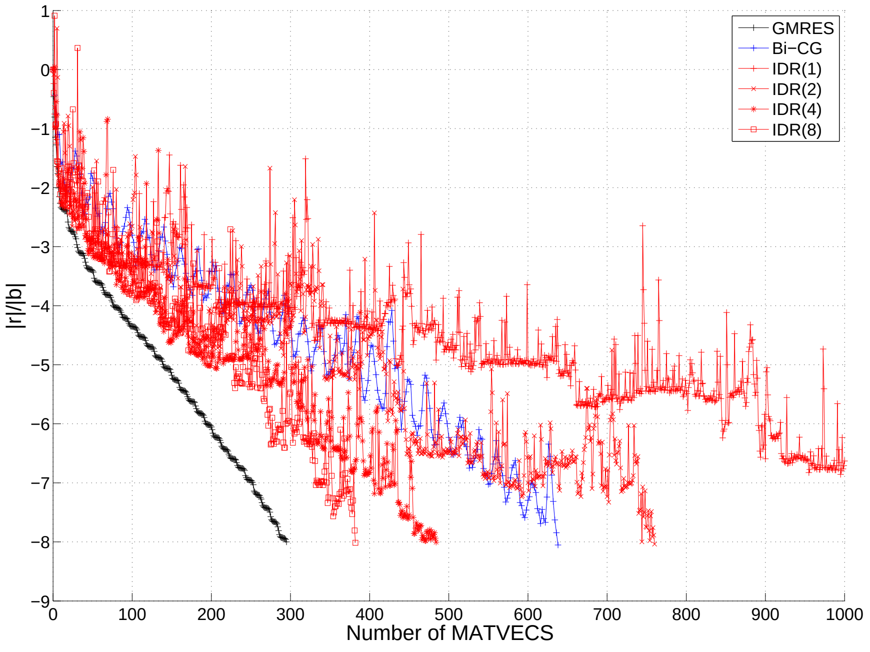

Figure 7.6: Convergence behaviour of the `add20` matrix for GMRES, Bi-CG and IDR(s)

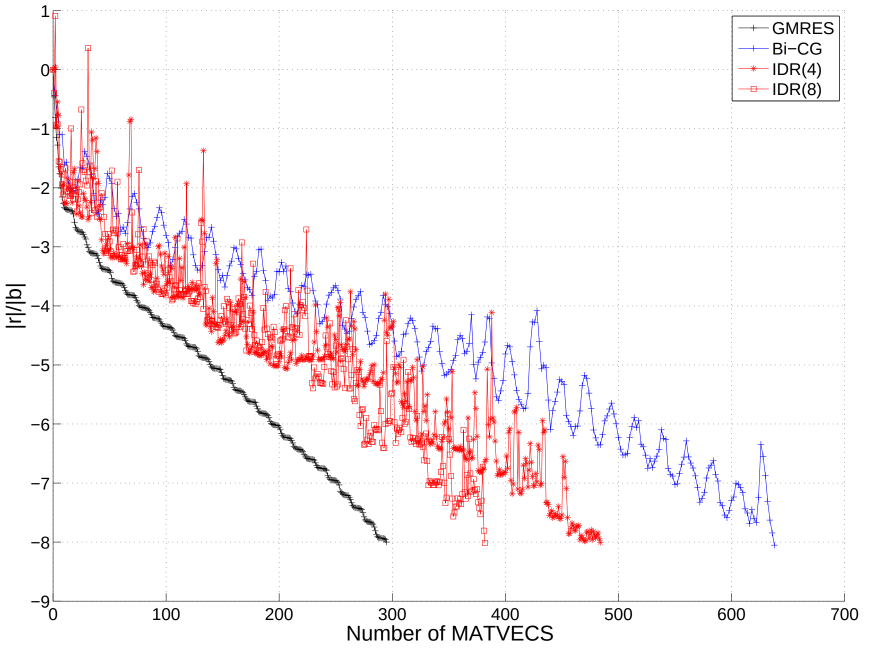

Figure 7.7: Convergence behaviour of the `add20` matrix for GMRES, Bi-CG and IDR(s)

In Table 7.4 we summarise the results. We see that IDR(4) and IDR(8) need fewer MATVECS to find a solution, but it does take more time to do so. We see that for increasing s the performance of IDR(s) also increases. Furthermore we see that GMRES is about 20 times as slow as Bi-CG and 16 times as slow as IDR(8).

7.4 Example 7.4 - The `jpwh_991` matrix

The `jpwh_991` matrix[4] is another matrix from the Matrix Market. It is a real nonsymmetric 991×991 matrix that is used in circuit physics. It has the following pattern:

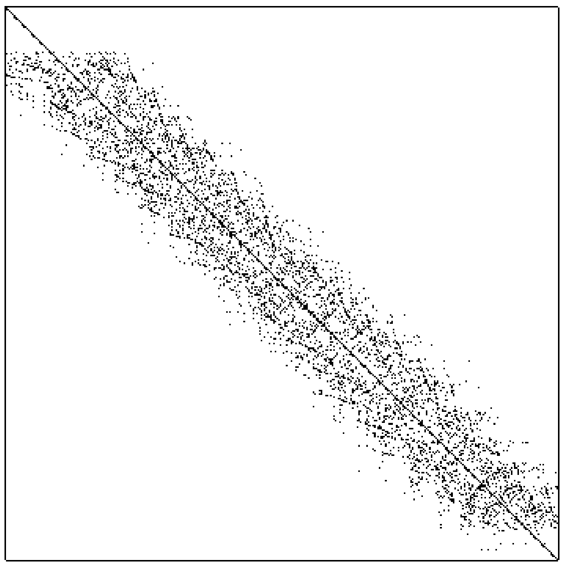

Figure 7.8: Structure of the `jpwh_991` matrix

We now solve the system $Ax = b$, where b is A times a vector with ones in all its entries (so the solution will be a vector with ones in all its entries). We use a tolerance of 10^{-8} and we take 1000 as the maximum number of iterations.

In Figure 7.9 we see the convergence behaviour of the GMRES method, the Bi-CG method and the IDR(s) method. From the plot it immediately becomes clear that the Bi-CG method does not compute a solution. It returns `flag = 4`, which means that 'one of the scalar quantities calculated during Bi-CG became too small or too large to continue computing'. We see that the different IDR(s) methods perform well on this problem. Note that the convergence behaviour of the IDR(2) method is more irregular than the other three IDR(s) methods, especially between ten and thirty MATVECS and during the last ten MATVECS. It even needs more MATVECS than the IDR(1) method.

In Table 7.5 we summarise the results. We see that IDR(1) and IDR(2) need the same amount of CPU time and the same holds for IDR(4) and IDR(8). Furthermore we see that IDR(1) and IDR(2) are approximately three times as fast as GMRES and IDR(4) and IDR(8) are approximately 6 times as fast.

[4]`http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/cirphys/jpwh_991.html`

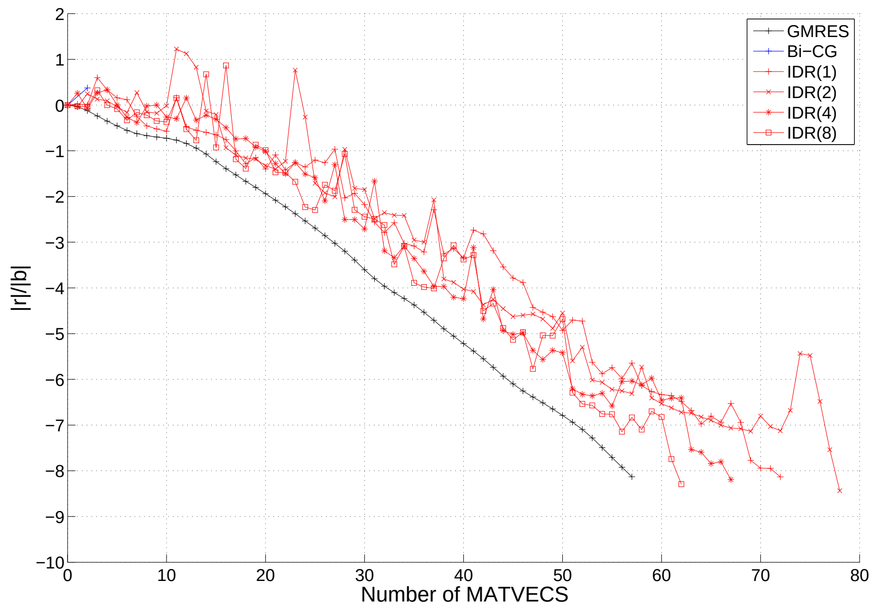

Figure 7.9: Convergence behaviour of the `jpwh_991` matrix for GMRES, Bi-CG and IDR(s)

Method	MATVECS	CPU time
GMRES	57	0.1092s
Bi-CG	-	-
IDR(1)	72	0.0312s
IDR(2)	78	0.0312s
IDR(4)	67	0.0156s
IDR(8)	62	0.0156s

Table 7.5: Convergence behaviour of the `jpwh_991` matrix for GMRES, Bi-CG and IDR(s)

7.5 Conclusions

We have seen in several examples that the IDR(s) method performs well in comparison with the GMRES method and the Bi-CG method. IDR(s) beats the GMRES method when computation time is concerned, although the GMRES method needs fewer iterations because of the minimal residual norm. We have also seen that in some cases the IDR(s) method might find an approximate solution faster than the Bi-CG method. although this is not always the case. In these examples we do see that IDR(s) needs fewer iterations. This makes research in the direction of more efficient IDR(s) algorithms justified.

8. Research goals

Chapter 3 explained the theory behind projection methods. Recall that a projection method wants to find and approximate solution x_m in a Krylov subspace $\mathcal{K}_m$ such that the residual vector $r_m = b - Ax_m$ is in a Krylov subspace $\mathcal{L}_m$. In chapter 4 we explored different Krylov subspace method, either for finding the eigenvalues of a matrix A (general or SPD) or for solving a linear system of equations and we saw how these methods fitted in the framework of projection methods.

In recent years there has been renewed interest in the IDR(s) method. It turns out that the IDR(s) method, which chapter 6 described, can also be seen as a projection method (which are also called Petrov-Galerkin methods). In their paper 'Interpreting IDR as a Petrov-Galerkin method' [Simoncini and Szyld, 2010], Valeria Simoncini and Daniel B. Szyld showed that the IDR(s) method can be seen as a projection method. When the left subspace $\mathcal{L}_m$ is appropriately chosen, we see that the IDR(s) method can be interpreted as a classical Krylov subspace method satisfying the Petrov-Galerkin condition (see section 3.1). One part of this graduation project is to make clear how the IDR(s) method can be seen as a projection method. To do this, we will follow the approach that Simoncini and Szyld have taken.

As P. Sonneveld and M.B.Van Gijzen describe, the IDR(s) algorithm in section 6.1 is a 'direct translation of the IDR theorem into an actual algorithm'. However, there is some freedom in this translation. We obtain mathematically different methods if we make different choices. For instance, we have freedom in choosing the matrix P that defines the subspace $\mathcal{S}$. Secondly, there are different ways to define and calculate the residuals r_j. Lastly, we might try to improve the performance of IDR(s) by looking at different ways in which we can choose the ω's.

In the algorithm described by P. Sonneveld and M.B. Van Gijzen, the value of ω_{n+1} is chosen by minimising the norm of the residual [Sonneveld and Van Gijzen, 2008]. This can be done as follows:

$$\begin{aligned} ||r_{n+1}|| &= ||v_j - \omega_{j+1}Av_j|| \qquad \text{let } t = Av_j && (8.1)\\ &= ||v_j - \omega_{j+1}t|| && (8.2)\\ &= (v_j - \omega_{j+1}t)^T(v_j - \omega_{j+1}t) && (8.3)\\ &= v_j^T v_j - 2\omega_{j+1}t^T v_j + 2\omega_{j+1}t^T t && (8.4) \end{aligned}$$

Note that this is a polynomial in ω_{j+1}. We can differentiate with respect to ω_{j+1} and find that $||r_{j+1}||$ is minimal for

$$\omega_{j+1} = \frac{(t, v_j)}{(t, t)}. \qquad (8.5)$$

However, it is not clear that this approach always works. ω_{j+1} is chosen to minimise $||r_{j+1}||$, but is is also used in the calculation of the subbsequent residuals, which might not be minimised by this particular ω. Furthermore, the algorithm might break down if $\omega \approx 0$. Indeed, there are matrices for which the calculations of A fail systematically [Sonneveld and Van Gijzen, 2008, p. 1044].

In their article, Valeria and Szyld propose a new version of the IDR(s) algorithm, called *Ritz-IDR*. In this new version of IDR(s), they substitute the ω_{j+1}'s with 'the Ritz values obtained by a preliminary generation of a small Krylov subspace of fixed dimension m_0'. For instance, one might use the Arnoldi method, the Lanczos method or the Bi-Lanczos method. They then use a significant portion of the Ritz values that are largest in magnitude and use them in the IDR(s) method for solving linear systems of equations. However, one might wonder if this can be done more efficient, since the IDR(s) algorithm can also be modified to make it suited for calculating Ritz values. This is the second topic that we want to cover in this graduation research.

To conclude, we now summarise the two research questions for this graduation project. The ultimate goal of this research is twofold. We want to have an answer to the following two questions:

1. How can we put IDR in the framework of projection methods?
2. Can we improve the performance of the IDR(s) algorithm by using IDR(s) itself to find the Ritz values?

Bibliography

Arnoldi, W. E. (1951). The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9(17):17–29.

Faber, V. and Manteuffel, T. (1984). Necessary and sufficient conditions for the existence of a Conjugate Gradient method. *SIAM Journal on Numerical Analasys*, 21(2):352–362.

Holub, G. H. and Van Loan, C. F. (1996). *Matrix Computations.* The John Hopkins University Press, Baltimore, MD, USA, 3rd edition. `http://read.pudn.com/downloads96/ebook/390385/%28U.John%20Hopkins%29%20Matrix%20Computations%20%283rd%20Ed.%29.pdf`, accessed on January 24, 2013.

Lanczos, C. (1952). Solution of systems of linear equations by minimized iterations. *Journal of Research of the National Bureau of Standards*, 49:33–53.

Poole, D. (2006). *Linear Algebra: A Modern Introduction.* Thomsom Brooks/Cole, Belmont, CA, USA, 2nd edition.

Saad, Y. (2003). *Iterative methods for sparse linear systems.* Society for Industrial and Applied Mathematics (SIAM), 2nd edition. `http://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf`, accessed on November 15, 2013.

Simoncini, V. and Szyld, D. B. (2010). Interpreting idr as a petrov-galerkin method. *SIAM Journal on Scientific Computing*, 32(4):1898–1912. `https://www.math.temple.edu/~szyld/reports/report_IDR.pdf`, accessed on January 31, 2014.

Sogabe, T., Sugihara, M., and Zhang, S.-L. (2009). An extension of the Conjugate Residual method to nonsymmetric linear systems. *Journal of Computational and Applied Mathematics*, 226(1):101–113. `http://www.sciencedirect.com/science/article/pii/S0377042708002264`, accessed on February 5, 2014.

Sonneveld, P. (1989). CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 10(1):36–52.

Sonneveld, P. and Van Gijzen, M. B. (2008). IDR(s): a family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM Journal on Scientific Computing*, 31(2):1035–1062.

Van der Vorst, A. H. (1992). Bi-CGSTAB: A fast and smoothly convergingvariant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM journal on Scientific and Statistical Computing*, 13(2):631–644.

Vuik, C. and Lahaye, D. J. P. (2010). *Course WI4201 Scientific Computing.* Delft Institue of Applied Mathematics (DIAM), Delft, The Netherlands.

Wesseling, P. and Sonneveld, P. (1980). Numerical experiments with a multiple grid and a preconditioned Lanczos type method. In *Approximation Methods for Navier-Stokes Problems*, volume 771 of *Lecture Notes in Mathematics*, pages 543–562. Springer Berlin Heidelberg.

A. Implentations for eigenvalue problems

A.1 Aanroep_methodes.m

```
clear all
close all
clc

% defaults
maxit = 1000;
n     = 100;

% symmetric matrix A
A = gallery('moler',n,-1);
A = gallery('poisson',n); n=size(A,1);

% nonsymmetric matrix A
A = gallery('hanowa',n);
A = gallery('rando',n,3);
A = gallery('toeppen',n);

% other defaults
b = A * ones(n,1);
x = zeros(n,1);

% execution of symmetric methods (Lanczos type)
[eigenvectors_A,eigenvalues_A,Residual,Iterations] = Lanczos(A)
[Solution, Residual, Iterations] = Lanczos_system(A,b,x,maxit)
[Solution, Residual, Iterations] = CG(A,b,x,maxit)
[Solution, Residual, Iterations] = CR(A,b,x,maxit)

% execution of general methods (Arnoldi type)
[eigenvectors_A,eigenvalues_A,Residual,Iterations] = Arnoldi(A)
[Solution, Residual, Iterations] = FOM(A,b,x,maxit)
[Solution, Residual, Iterations] = GMRES(A,b,x,maxit)

% execution of general methods (Bi-Lanczos type)
[eigenvectors_A,eigenvalues_A,Residual,Iterations] = Bi_Lanczos(A)
[Solution, Residual, Iterations] = Bi_Lanczos_system(A,b,x,maxit)
[Solution, Residual, Iterations] = Bi_CG(A,b,x,maxit)
[Solution, Residual, Iterations] = Bi_CR(A,b,x,maxit)

% execution of general methods (IDR-type)
[x,flag,relres,iter,resvec,H,eigenvaluesH,replacements]=idrs(A,b);
```

A.2 Arnoldi.m

```
function [eigenvectors_A,eigenvalues_A,Residual,Iterations]=Arnoldi(A)

n = length(A);

% Calculations & declarations
V(:,1) = ones(n,1) / norm(ones(n,1));

for j=1:n

    w = A * V(:,j);

    for i = 1:j
        H(i,j)    = w' * V(:,i);
        w = w - H(i,j)*V(:,i);
    end

    H(j+1,j) = norm(w);

    if  j==n || H(j+1,j) <= 1e-15
        break
    end

    V(:,j+1) = w / H(j+1,j);

    % Calculation of eigenvector (s) corresponding to eigenvalue with
    % the largest magnitude
    [EV,EW] = sorteig(H(1:j,1:j));
    s       = EV(:,j);

    % Residual vector and stopping criterion
    R(j) = H(j+1,j) * abs(s(j));
    if R(j) < 10^-8
        break
    end

end

% Calculation of eigenvalues and eigenvectors of H
[eigenvectors_H,eigenvalues_H] = sorteig(H(1:end-1,1:end));

% Approximation of the eigenvectors of A
eigenvalues_A  = diag(eigenvalues_H);
eigenvectors_A = V(1:end,1:j) * eigenvectors_H;

% Number of iterations and residual vector
Residual   = R';
Iterations = j;
```

A.3 Lanczos.m

```
function [eigenvectors_A,eigenvalues_A,Residual,Iterations]=Lanczos(A)

n = length(A);

% Calculations & declarations
V(:,1)   = zeros(n,1);
V(:,2)   = ones(n,1) / norm(ones(n,1));

alpha(1) = 0;
beta(1)  = 0;

for j=1:n

    r = A * V(:,j+1);

    alpha(j)   = V(:,j+1)' * r;
    r          = r - alpha(j) * V(:,j+1) - beta(j)*V(:,j);
    beta(j+1) = norm(r);

    if j==n || beta(j+1) == 0
        break
    end

    V(:,j+2) = r / beta(j+1);

    % Building T
    k = length(alpha);
    T = full(spdiags([beta(2:k+1)',alpha',beta(1:k)'],[-1 0 1],k,k));

    % Calculation of eigenvector (s) corresponding to eigenvalue with
    % the largest magnitude
    [EV,EW] = sorteig(T);
    s       = EV(:,j);

    % Residual vector and stopping criterion
    R(j) = beta(j+1) * abs(s(j));
    if R(j) < 10^-8
        break
    end

end

% Calculation of eigenvalues and eigenvectors of T
[eigenvectors_T,eigenvalues_T] = sorteig(T);

% Approximation of the eigenvectors and eigenvalues of A
eigenvalues_A  = diag(eigenvalues_T);
eigenvectors_A = V(:,2:end-1) * eigenvectors_T;

% Number of iterations and residual vector
Residual   = R';
Iterations = j;
```

A.4 Bi_Lanczos.m

```matlab
function [eigenvectors_A,eigenvalues_A,Residual,Iterations]=Bi_Lanczos(A)

n = length(A);

% Calculations & declarations
V(:,1)    = zeros(n,1);
W(:,1)    = V(:,1);
V(:,2)    = ones(n,1) / norm(ones(n,1));
W(:,2)    = V(:,2);

alpha(1)  = 0;
beta(1)   = 0;
delta(1)  = 0;

for j = 1:n

    Vbar = A  * V(:,j+1);
    Wbar = A' * W(:,j+1);

    alpha(j) =  Vbar' * W(:,j+1);

    Vbar = Vbar - alpha(j) * V(:,j+1) -  beta(j) * V(:,j);
    Wbar = Wbar - alpha(j) * W(:,j+1) - delta(j) * W(:,j);

    Ubar = Vbar' * Wbar;

    delta(j+1) = sqrt(abs(Ubar));

    if delta(j+1) == 0
        break
    end

    beta(j+1) = (Ubar) / delta(j+1);

    W(:,j+2) = Wbar /  beta(j+1);
    V(:,j+2) = Vbar / delta(j+1);

    % Building T
    k = length(alpha);
    T = full(spdiags([delta(2:k+1)',alpha',beta(1:k)'],[-1 0 1],k,k));

    % Calculation of eigenvector (s) corresponding to eigenvalue with
    % the largest magnitude
    [EV,EW] = sorteig(T);
    s       = EV(:,j);

    % Residual vector and stopping criterion
    R(j) = abs(delta(j+1)) * abs(s(j)) * norm(V(:,j+2));
    if R(j) < 10^-8
        break
    end

end
```

```

% Calculation of eigenvalues and eigenvectors of T
[eigenvectors_T,eigenvalues_T] = sorteig(T);

% Approximation of the eigenvectors of A
eigenvalues_A  = diag(eigenvalues_T);
eigenvectors_A = V(:,2:end-1) * eigenvectors_T;

% Number of iterations and residual vector
Residual   = R';
Iterations = j;
```

B. Implementations for linear solvers

B.1 FOM.m

```
function [Solution, Residual, Iterations]=FOM(A,b,x0,maxit)

% Calculations & declarations
r0     = b - A*x0;
R(1)   = norm(r0);
normb  = norm(b);
V(:,1) = r0 / R(1);

for j=1:maxit

    w = A * V(:,j);

    for i = 1:j
        H(i,j)   = w' * V(:,i);
        w = w - H(i,j)*V(:,i);
    end

    H(j+1,j) = norm(w);

    if  H(j+1,j) <= 1e-15
        break
    end

    V(:,j+1) = w / H(j+1,j);

    % Solving a least-square problem for y
    e_1 = zeros(j,1); e_1(1) = 1;
    y   = H(1:end-1,1:j) \ (R(1) * e_1);

    % Residual vector and stopping criterion
    R(j+1) = H(j+1,j) * abs(y(j));
    if R(j+1) / normb < 10^-8
        break
    end

end

Residual   = R';
Solution   = x0 + V(:,1:j)*y;
Iterations = j;
```

B.2 GMRES.m

```matlab
function [Solution, Residual, Iterations]=GMRES(A,b,x0,maxit)

% Calculations & declarations
r0      = b - A*x0;
R(1)    = norm(r0);
normb   = norm(b);
V(:,1)  = r0 / R(1);

for j=1:maxit

    w = A * V(:,j);

    for i = 1:j
        H(i,j)    = w' * V(:,i);
        w = w - H(i,j)*V(:,i);
    end

    H(j+1,j) = norm(w);

    if  H(j+1,j) <= 1e-15
        break
    end

    V(:,j+1) = w / H(j+1,j);

    % Solving a least-square problem for y
    e_1 = zeros(j+1,1); e_1(1) = 1;
    y   = H \ (R(1) * e_1);

    % Residual vector and stopping criterion
    R(j+1) = norm(R(1) * e_1 - H * y );
    if R(j+1) / normb < 10^-8
        break
    end

end

Residual   = R';                     % Residual in each iteration
Solution   = x0 + V(:,1:j)*y;        % Solution
Iterations = j;                      % Number of iterations
```

B.3 Lanczos_system.m

```
function [Solution, Residual, Iterations] = Lanczos_system(A,b,x0,maxit)

n = length(A);

% Calculations & declarations
r0       = b - A*x0;
R(1)     = norm(r0);
normb    = norm(b);

V(:,1)   = zeros(n,1);
V(:,2)   = r0/R(1);

alpha(1) = 0;
beta(1)  = 0;

for j=1:maxit

    r = A * V(:,j+1);

    alpha(j)   = V(:,j+1)' * r;
    r          = r - alpha(j) * V(:,j+1) - beta(j)*V(:,j);
    beta(j+1) = norm(r);

    if beta(j+1) == 0
        break
    end

    V(:,j+2) = r / beta(j+1);

    % Building T
    k = length(alpha);
    T = full(spdiags([beta(2:k+1)',alpha',beta(1:k)'],[-1 0 1],k,k));

    % Solving a least-square problem for y
    e_1 = zeros(j,1); e_1(1) = 1;
    y   = T \ (R(1) * e_1);

    % Residual vector and stopping criterion
    R(j+1) = norm(beta(j+1) * y(j));
    if R(j+1) / normb < 10^-8
        break
    end

end

Residual    = R';
Solution    = x0 + V(:,2:end-1)*y;
Iterations = j;
```

B.4 CG.m

```
function [Solution, Residual, Iterations] = CG(A,b,x,maxit)

% Calculations & declarations
r      = b - A*x;
p      = r;
R(1)   = norm(r);
normb  = norm(b);

for j=1:maxit

    y = r'*r;
    z = A*p;

    alpha = y / (z'*p);
    x     = x + alpha * p;
    r     = r - alpha * z;
    beta  = r'*r / y;
    p     = r + beta*p;

    % Residual vector and stopping criterion
    R(j+1) = norm(r);
    if  R(j+1) / normb < 10^-8
            break
    end
end

Residual   = R';
Solution   = x;
Iterations = j;
```

B.5 CR.m

```
function [Solution, Residual, Iterations] = CR(A,b,x,maxit)

% Calculations & declarations
r     = b - A*x;
p     = r;
R(1)  = norm(r);
normb = norm(b);

for j=1:maxit

    y = r'*A'*r;
    z = A*p;

    alpha = y / (z'*z);
    x     = x + alpha * p;
    r     = r - alpha * z;
    beta  = r'*A*r / y;
    p     = r + beta*p;

    % Residual vector and stopping criterion
    R(j+1) = norm(r);
    if  R(j+1) / normb < 10^-8
            break
    end
end

Residual   = R';
Solution   = x;
Iterations = j;
```

B.6 Bi_Lanczos_system.m

```
function [Solution, Residual, Iterations] = Bi_Lanczos_system(A,b,x0,maxit)

n = length(A);

% Calculations & declarations
r0        = b - A*x0;
normb     = norm(b);
R(1)      = norm(r0);

V(:,1)    = zeros(n,1);
W(:,1)    = zeros(n,1);
V(:,2)    = r0 / R(1);
W(:,2)    = V(:,2);

alpha(1)  = 0;
beta(1)   = 0;
delta(1)  = 0;

for j = 1:maxit

    Vbar = A  * V(:,j+1);
    Wbar = A' * W(:,j+1);

    alpha(j) =  Vbar' * W(:,j+1);

    Vbar = Vbar - alpha(j) * V(:,j+1) -  beta(j) * V(:,j);
    Wbar = Wbar - alpha(j) * W(:,j+1) - delta(j) * W(:,j);

    Ubar = Vbar' * Wbar;

    delta(j+1) = sqrt(abs(Ubar));

    if delta(j+1) == 0
        break
    end

    beta(j+1) = (Ubar) / delta(j+1);

    W(:,j+2) = Wbar /  beta(j+1);
    V(:,j+2) = Vbar / delta(j+1);

    % Building T
    k = length(alpha);
    T = full(spdiags([delta(2:k+1)',alpha',beta(1:k)'],[-1 0 1],k,k));

    % Solving a least-square problem for y
    e_1 = zeros(j,1); e_1(1) = 1;
    y   = T \ (R(1) * e_1);

    % Residual vector and stopping criterion
    R(j+1) = norm(delta(j+1) * y(j) * V(:,j+2));
    if R(j+1) / normb < 10^-8
        break
    end

end

Residual   = R';
Solution   = x0 + V(:,2:end−1)*y;
Iterations = j;
```

B.7 Bi_CG.m

```matlab
function [Solution, Residual, Iterations] = Bi_CG(A,b,x,maxit)

% Calculations & declarations
r     = b - A*x;
p     = r;
rster = r;
pster = p;
normb = norm(b);
R(1)  = norm(r);

for j=1:maxit

    y = r' * rster;
    z = A * p;

    alpha = y / (z' * pster);
    x     = x + alpha * p;
    r     = r - alpha * z;
    rster = rster - alpha * A' * pster;

    beta  = (r' * rster) / y;
    p     = r     + beta * p;
    pster = rster + beta * pster;

    % Residual vector and stopping criterion
    R(j+1)  = norm(r);
    if  R(j+1) < normb * 10^-8
            break
    end

end

Residual   = R';
Solution   = x;
Iterations = j;
```

B.8 Bi_CR.m

```matlab
function [Solution, Residual, Iterations] = Bi_CR(A,b,x,maxit)

% Calculations & declarations
r      = b - A*x;
p      = r;
rster = r;
pster = p;
normb = norm(b);
R(1)   = norm(r);

for j=1:maxit

    y = rster' * A * r;
    z = A * p;

    alpha = y / (pster'*A*z);
    x      = x + alpha * p;
    r      = r - alpha * z;
    rster = rster - alpha * A' * pster;

    beta  = rster'*A*r   /   y;
    p      = r      + beta * p;
    pster = rster + beta * pster;

    % Residual vector and stopping criterion
    R(j+1)  = norm(r);
    if  R(j+1) < normb * 10^-8
            break
    end

end

Residual    = R';
Solution    = x;
Iterations = j;
```

B.9 IDRS.m

```
function [x,flag,relres,iter,resvec,H,replacements]=
               IDRS(A,b,s,tol,maxit,M1,M2,x0,options)
%IDRS Induced Dimension Reduction method
%   X = IDRS(A,B) solves the system of linear equations A*X=B for X.
%   The N-by-N coefficient matrix A must be square and the right-hand
%   side column vector B must have length N. A can also be a struct.
%   The field A.name should contain the name of a function to perform
%   multiplications with A. Other fields can be used to pass parameters
%   to this function.
%
%   X = IDRS(A,B,S) specifies the dimension of the 'shadow space'.
%   If S = [], then IDRS uses the default S = 4. Normally, a higher S
%   gives faster convergence, but also makes the method more expensive.
%
%   X = IDRS(A,B,S,TOL) specifies the tolerance of the method.
%   If TOL is [] then IDR uses the default, 1e-8.
%
%   X = IDRS(A,B,S,TOL,MAXIT) specifies the maximum number of iterations.
%   If MAXIT is [] then IDRS uses the default, min(2*N,1000).
%
%   X = IDRS(A,B,S,TOL,MAXIT,M1) use preconditioner M1. If M1 is [] then no
%   preconditioner is applied.
%   IDRS(A,B,S,TOL,MAXIT,M1,M2) uses a factored preconditioner M = M1 M2.
%   M1 and M2 can be structures. In that case the field M1.name and M2.name
%   should contain function names for M1 and M2.
%
%   X = IDRS(A,B,S,TOL,MAXIT,M1,M2,X0) specifies the initial guess.
%   If X0 is [] then IDR uses the default, an all zero vector.
%
%   X = IDRS(A,B,S,TOL,MAXIT,M1,M2,X0,OPTIONS) specifies additional options.
%      OPTIONS must be a structure
%      OPTIONS.SMOOTHING specifies if residual smoothing must be applied
%         OPTIONS.SMOOTHING = 0: No smoothing
%         OPTIONS.SMOOTHING = 1: Smoothing
%         Default: OPTIONS.SMOOTHING = 0;
%      OPTIONS.OMEGA determines the computation of OMEGA
%         If OPTIONS.OMEGA = 0: a standard minimum residual step is performed
%         If OPTIONS.OMEGA > 0: OMEGA is increased if
%         the cosine of the angle between Ar and r < OPTIONS.OMEGA
%         Default: OPTIONS.OMEGA = 0.7;
%      OPTIONS.P defines the 'shadow' space
%         Default: OPTIONS.P = ORTH(RANDN(N,S));
%      OPTIONS.REPLACE determines the residual replacement strategy
%         If |r| > 1E3 |b| TOL/EPS) (EPS is the machine precision)
%         the recursively computed residual is replaced by the true residual
%         once |r| < |b| (to reduce the effect of large intermediate residuals
%         on the final accuracy)
%         Default: OPTIONS.REPLACE = 0; (No residual replacement)
%
%   [X,FLAG] = IDRS(A,B,S,TOL,MAXIT,M1,M2,X0,OPTIONS)
%   also returns an information flag:
%       FLAG = 0: required tolerance satisfied
%       FLAG = 1: no convergence to the required tolerance within maximum
%                  number of iterations
%       FLAG = 2: check RELRES, possible stagnation above required
%                  tolerance level
%       FLAG = 3: one of the iteration parameters became zero,
%                  causing break down
%
%    [X,FLAG,RELRES] = IDRS(A,B,S,TOL,MAXIT,M1,M2,X0,OPTIONS)
%    also returns the relative residual norm:
%           RELRES = ||B - AX||/||B||
%
%    [X,FLAG,RELRES,ITER] = IDRS(A,B,S,TOL,MAXIT,M1,M2,X0,OPTIONS)
%    also returns the number of iterations.
%
%    [X,FLAG,RELRES,ITER,RESVEC] = IDRS(A,B,S,TOL,MAXIT,M1,M2,X0,OPTIONS)
%    also returns a vector of the residual norms at each matrix-vector
%    multiplication.
%
%    [X,FLAG,RELRES,ITER,RESVEC,REPLACEMENTS] =
%    IDRS(A,B,S,TOL,MAXIT,M1,M2,X0,OPTIONS)
%    also returns the number of residual replacements
%
%    The software is distributed without any warranty.
%
%    Martin van Gijzen and Peter Sonneveld
%    Copyright (c) September 2008
%    Version August 9, 2010
%

if ( nargout == 0 )
   help idrs;
   return
end

% Check for an acceptable number of input arguments
if nargin < 2
   error('Not enough input arguments.');
end

% Check matrix and right hand side vector inputs have appropriate sizes
funA = 0;
if isa(A,'struct')
   funA = 1;
   if isfield(A,'name')
      function_A = A.name;
   else
      error('Use field A.name to specify function name for
             matrix-vector multiplication');
   end
   n = length(b);
else
   [m,n] = size(A);
   if (m ~= n)
      error('Matrix must be square.');
   end
   if ~isequal(size(b),[m,1])
      es = sprintf(['Right hand side must be a column vector of' ...
            ' length %d to match the coefficient matrix.'],m);
      error(es);
   end
end

% Assign default values to unspecified parameters
if nargin < 3 || isempty(s)
   s = 4;
end
if ( s > n )
   s = n;
end
if nargin < 4 || isempty(tol)
   tol = 1e-8;
end
if nargin < 5 || isempty(maxit)
   maxit = min(2*n,1000);
end

if nargin < 6 || isempty(M1)
   precL = 0;
elseif isa(M1,'struct')
   if isfield(M1,'name')
      function_M1 = M1.name;
   else
      error('Use field M1.name to specify function name preconditioner');
   end
   precL = 1;
   funL = 1;
else
   if ~isequal(size(M1),[n,n])
      es = sprintf(['Preconditioner must be a matrix of' ...
            ' size %d times %d to match the problem size.'],n,n);
      error(es);
   end
   precL = 1;
   funL = 0;
end

if nargin < 7 || isempty(M2)
   precU = 0;
elseif isa(M2,'struct')
   if isfield(M2,'name')
      function_M2 = M2.name;
   else
      error('Use field M2.name to specify function name preconditioner');
   end
   precU = 1;
   funU = 1;
else
   if ~isequal(size(M2),[n,n])
      es = sprintf(['Preconditioner must be a matrix of' ...
            ' size %d times %d to match the problem size.'],n,n);
      error(es);
   end
   precU = 1;
   funU = 0;
end
```

```matlab

if nargin < 8 || isempty(x0)
   x0 = zeros(n,1);
else
   if ~isequal(size(x0),[n,1])
      es = sprintf(['Initial guess must be a column vector of' ...
            ' length %d to match the problem size.'],n);
      error(es);
   end
end

% Other parameters
smoothing = 0;
angle = 0.7;
replacement = 0;
replacements = 0;
randn('state', 0);
P = randn(n,s);
P = orth(P);
U = zeros(n,s);
inispace = 0;

if ( nargin > 8  )
% Residual smoothing:
   if isfield(options,'smoothing')
      smoothing = options.smoothing > 0;
   end
% Computation of omega:
   if isfield(options,'omega')
      angle = options.omega;
   end
% Alternative definition of P:
   if isfield(options,'P')
      P = options.P;
      if ~isequal(size(P),[n,s])
         es = sprintf(['P must be a matrix of' ...
         ' size %d times %d to match the problem size.'],n,s);
         error(es);
      end
   end
   if isfield(options,'replace' )
      replacement = options.replace > 0;
   end
   if isfield(options,'U0' )
      U = options.U0;
      if ~isequal(size(U),[n,s])
         es = sprintf(['U0 must be a matrix of' ...
         ' size %d times %d to match the problem size.'],n,s);
         error(es);
      end
      inispace = 1;
   end
end

if nargin > 9
   es = sprintf(['Too many input parameters']);
   error(es);
```

```
end

% END CHECKING INPUT PARAMETERS AND SETTING DEFAULTS

x = zeros(n,1);
% Check for zero rhs:
if (norm(b) == 0)              % Solution is nulvector
   iter = 0;
   resvec = 0;
   flag = 0;
   relres = 0;
   return
end

% Number close to machine precision:
mp = 1e3*eps;

% Initialize output paramater relres
relres = NaN;

% Compute initial residual:
x = x0;
normb = norm(b);
tolb = tol * normb;           % Relative tolerance

if funA
   r = b - feval( function_A, x, A);
else
   r = b - A*x;
end

if smoothing
   x_s = x0;
   r_s = r;
end

normr = norm(r);
resvec=[normr];
trueres = 0;

if (normr <= tolb)           % Initial guess is a good enough solution
   iter = 0;
   flag = 0;
   relres = 0;
   return
end

G = zeros(n,s); M = eye(s,s);
om = 1;

% init coefficients for H
h.a = []; h.b = []; h.g = []; h.o = [];
al = zeros(s,1);

% Main iteration loop, build G-spaces:
iter = 0;
while ( normr > tolb && iter < maxit )
```

```

% New righ-hand size for small system:
   f = (r'*P)';
   for k = 1:s

% Solve small system and make v orthogonal to P:
      c = M(k:s,k:s)\ f(k:s);
      v = r - G(:,k:s)*c;

% Store all gamma
      h.g = [h.g,[zeros(k-1,1);c]];

% Preconditioning:
      if ( precL )
         if funL
            v = feval( function_M1,v,M1 );
         else
            v = M1\v;
         end
      end
      if ( precU )
         if funU
            v = feval( function_M2,v,M2 );
         else
            v = M2\v;
         end
      end
%
% Compute new U(:,k) and G(:,k), G(:,k) is in space G_j
      if ~( iter <= s &&  inispace )
         U(:,k) = U(:,k:s)*c + om*v;
      end
      if ( funA )
         G(:,k) = feval( function_A,U(:,k),A );
      else
         G(:,k) = A*U(:,k);
      end
%
% Bi-Orthogonalise the new basis vectors:
      for i = 1:k-1
         al(i) = ( P(:,i)'*G(:,k) )/M(i,i);
         G(:,k) = G(:,k) - al(i)*G(:,i);
         U(:,k) = U(:,k) - al(i)*U(:,i);
      end

% Store all alpha
      h.a = [h.a,[al(1:k-1);1;zeros(s-k,1)]];

%
% New column of M = P'*G  (first k-1 entries are zero)
      M(k:s,k) = (G(:,k)'*P(:,k:s))';
      if ( M(k,k) == 0 )
         flag = 3;
         return;
      end
%
%  Make r orthogonal to p_i, i = 1..k
```

```
      beta = f(k)/M(k,k);
      r = r - beta*G(:,k);
      x = x + beta*U(:,k);
      normr = norm(r);
      if ( replacement && normr > tolb/mp ) trueres = 1; end;

% Store all beta
      h.b = [h.b;beta];

%
%  Smoothing:
      if ( smoothing )
         t = r_s - r;
         gamma = (t'*r_s)/(t'*t);
         r_s = r_s - gamma*t;
         x_s = x_s - gamma*(x_s - x);
         normr = norm(r_s);
      end
      resvec = [resvec;normr];
      iter = iter + 1;

      if ( normr < tolb | iter == maxit )
         break
      end

%
% New f = P'*r (first k  components are zero)
      if ( k < s )
         f(k+1:s)   = f(k+1:s) - beta*M(k+1:s,k);
      end
   end
%
   if ( normr < tolb | iter == maxit )
      break
   end

%
% Now we have sufficient vectors in G_j to compute residual in G_j+1
% Note: r is already perpendicular to P so v = r

% Preconditioning:
   v = r;
   if ( precL )
      if funL
         v = feval( function_M1,v,M1 );
      else
         v = M1\v;
      end
   end
   if ( precU )
      if funU
         v = feval( function_M2,v,M2 );
      else
         v = M2\v;
      end
   end

```

```
% Matrix-vector multiplication:
   if ( funA )
      t = feval( function_A,v,A );
   else
      t = A*v;
   end

% Computation of a new omaga
   om = omega( t, r, angle );

% Store all omega
   h.o = [h.o;om];

   if ( om == 0 )
      flag = 3;
      return;
   end
%
   r = r - om*t;
   x = x + om*v;
   normr = norm(r);
   if ( replacement && normr > tolb/mp ) trueres = 1; end;
%
%     Residual replacement?
   if ( trueres && normr < normb )
      if funA
         r = b - feval( function_A,x,A );
      else
         r = b - A*x;
      end
      trueres = 0;
      replacements = replacements+1;
   end
%
%     Smoothing:
   if ( smoothing )
      t    = r_s - r;
      gamma_sm = (t'*r_s)/(t'*t);
      r_s = r_s - gamma_sm*t;
      x_s = x_s - gamma_sm*(x_s - x);
      normr = norm(r_s);
   end
%
   resvec = [resvec;normr];
   iter = iter + 1;

end; %while

if ( smoothing )
   x = x_s;
end

if funA
   relres = norm(b - feval( function_A,x,A ))/normb;
else
   relres = norm(b - A*x)/normb;
end
```

```
if ( relres < tol )
   flag = 0;
elseif ( iter == maxit )
   flag = 1;
else
   flag = 2;
end

% Build matrix H
n = length(h.b);
% n = min(length(h.b),size(A,2));
H = spalloc(n+1,n,n*(s+2));

m = length(h.o);
for j=0:m
   for k=1:s
      if j == 0
         H(1:k+1,k) = diff([0;h.a(1:k,k)./h.b(1:k);0])./1; % h.o(0) = 1
      else
         if j*s+k <= n
            H((j-1)*s+(k:s+k+1),j*s+k)=
            diff([0;-h.g(k:s,j*s+k)./h.b((j-1)*s+
            (k:s)); h.a(1:k,j*s+k)./h.b(j*s+(1:k));0])/h.o(j);
         end
      end
   end
end

H = H(1:n,1:n);

return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function om = omega( t, s, angle )

ns = norm(s);
nt = norm(t);
ts = t'*s;
rho = abs(ts/(nt*ns));
om=ts/(nt*nt);
if ( rho < angle )
   om = om*angle/rho;
end

return
```

C. Other Matlab files

C.1 Example_CG_Lanczos.m

```
close all
clear all
clc

% Defaults
n       = 100;
maxit   = 1000;

choice = 1;
hold on;
xlabel('Number of MATVECS');
ylabel('|r|/|b|');
set(gca,'FontSize',16);
xlhand = get(gca,'xlabel');
ylhand = get(gca,'ylabel');
set(xlhand,'fontsize',20);
set(ylhand,'fontsize',20);
grid on;

% Generate the linear system
A = gallery('poisson',n); n = size(A,1);
b = A * ones(n,1);
x = zeros(n,1);

% Plot of the CG method
t = cputime;
disp('CG iteration...');
[Solution, Residual, Iterations] = CG(A,b,x,maxit);
time    = cputime - t;
resvec = log10(Residual/Residual(1));
it      = [0:1:length(resvec)-1];
plot(it,resvec,'r-+');
drawnow;
disp(['Iterations: ',num2str(Iterations)]);
disp(['CPU time: ',num2str(time),'s.']);
disp(' ');

% Plot of the Lanczos method for linear systems
t = cputime;
disp('Lanczos for linear systems iteration...');
[Solution, Residual, Iterations] = Lanczos_system(A,b,x,maxit);
time    = cputime - t;
resvec = log10(Residual/Residual(1));
it      = [0:1:length(resvec)-1];
plot(it,resvec,'k-*');
drawnow;
disp(['Iterations: ',num2str(Iterations)]);
disp(['CPU time: ',num2str(time),'s.']);
disp(' ');
```

```

legend('CG', 'Lanczos method for linear systems');
hold off;
```

C.2 Example_BiCG_BiLanczos.m

```matlab
close all
clear all
clc

% Defaults
n      = 100;
maxit  = 1000;

choice = 1;
hold on;
xlabel('Number of MATVECS');
ylabel('|r|/|b|');
set(gca,'FontSize',16);
xlhand = get(gca,'xlabel');
ylhand = get(gca,'ylabel');
set(xlhand,'fontsize',20);
set(ylhand,'fontsize',20);
grid on;

% Generate the linear system
A = gallery('toeppen',n);
b = A * ones(n,1);
x = zeros(n,1);

% Plot of the Bi_Lanczos method
t = cputime;
disp('Bi-CG for linear systems iteration...');
[Solution, Residual, Iterations] = Bi_CG(A,b,x,maxit);
time   = cputime - t;
resvec = log10(Residual/Residual(1));
it     = [0:2:2*(length(resvec)-1)];
plot(it,resvec,'r-+');
drawnow;
disp(['Iterations: ',num2str(Iterations)])
disp(['CPU time: ',num2str(time),'s.'])
disp(' ')

% Plot of the Bi_Lanczos method for linear systems
t = cputime;
disp('Bi-Lanczos iteration...');
[Solution, Residual, Iterations] = Bi_Lanczos_system(A,b,x,maxit);
time   = cputime - t;
resvec = log10(Residual/Residual(1));
it     = [0:2:2*(length(resvec)-1)];
plot(it,resvec,'k-*');
drawnow;
disp(['Iterations: ',num2str(Iterations)]);
disp(['CPU time: ',num2str(time),'s.']);
disp(' ');

legend('Bi-CG', 'Bi-Lanczos method for linear systems');
hold off;
```

C.3 Example_IDRS.m

```
%
% IDRS parameterised testproblems
%
% The software is distributed without any warranty.
%
% Martin van Gijzen
% Copyright (c) August 2010
%

clear all;
close all;
clc;

% Defaults:
h = 1/21;
eps = -1;
beta(1) = 100;
beta(2) = 100;
beta(3) = 100;
r = 0;

% Generate matrix
m = round(1/h)-1;
if ( m < 1 )
   error('h too small, should be large than 0.5');
end
n = m*m*m;
Sx = gallery('tridiag',m,-eps/h^2-beta(1)/(2*h),2*eps/h^2,
                         -eps/h^2+beta(1)/(2*h));
Sy = gallery('tridiag',m,-eps/h^2-beta(2)/(2*h),2*eps/h^2,
                         -eps/h^2+beta(2)/(2*h));
Sz = gallery('tridiag',m,-eps/h^2-beta(3)/(2*h),2*eps/h^2,
                         -eps/h^2+beta(3)/(2*h));
Is = speye(m,m);
I = speye(n,n);
A = kron(kron(Is,Is),Sx) + kron(kron(Is,Sy),Is)+ kron(kron(Sz,Is),Is) -r*I;

x = linspace(h,1-h,m);
sol = kron(kron(x.*(1-x),x.*(1-x)),x.*(1-x))';
b = A*sol;

% Defaults for the iterative solvers:

tol = 1e-8;
maxit = 1000;

choice = 1;
scrsz = get(0,'ScreenSize');
fig = figure('Position',[scrsz(1) + scrsz(3)/2 scrsz(4)/2
                         scrsz(3)/2 scrsz(4)/2]);
hold on;
xlabel('Number of MATVECS')
ylabel('|r|/|b|')
```

```
set(gca,'FontSize',16)
xlhand = get(gca,'xlabel');
ylhand = get(gca,'ylabel');
set(xlhand,'fontsize',20)
set(ylhand,'fontsize',20)
grid on;

t = cputime;
disp('GMRES iteration...');
[x, flag, relres, iter, resvec] = gmres(A, b, [], tol, 400 );
time = cputime - t;
resvec = log10(resvec/resvec(1));
figure(fig);
it = [0:1:length(resvec)-1];
plot(it,resvec,'k-+');
drawnow;
disp(['Final accuracy: ', num2str(norm(b-A*x)/norm(b))])
disp(['Iterations: ',num2str(iter(2))]);
disp(['CPU time: ',num2str(time),'s.']);
disp(' ');

t = cputime;
disp('Bi-CG iteration...');
[x, flag, relres, iter, resvec] = bicg(A, b, tol, maxit );
time = cputime - t;
resvec = log10(resvec/resvec(1));
figure(fig);
it = [0:2:2*(length(resvec)-1)];
plot(it,resvec,'b-+');
drawnow;
disp(['Final accuracy: ', num2str(norm(b-A*x)/norm(b))])
disp(['Iterations: ',num2str(iter)]);
disp(['CPU time: ',num2str(time),'s.']);
disp(' ');

s = 1;
t = cputime;
disp('IDR(1) iteration...');
[x, flag, relres, iter, resvec] = idrs( A, b, s, tol, maxit );
time = cputime - t;
resvec = log10(resvec/resvec(1));
figure(fig);
it = [0:1:length(resvec)-1];
plot(it,resvec,'r-+');
drawnow;
disp(['Final accuracy: ', num2str(norm(b-A*x)/norm(b))]);
disp(['Iterations: ',num2str(iter)]);
disp(['CPU time: ',num2str(time),'s.']);
disp(' ');

s = 2;
t = cputime;
disp('IDR(2) iteration...');
[x, flag, relres, iter, resvec] = idrs( A, b, s, tol, maxit );
time = cputime - t;
figure(fig);
resvec = log10(resvec/resvec(1));
```

```
it = [0:1:length(resvec)-1];
plot(it,resvec,'r-x');
drawnow;
disp(['Final accuracy: ', num2str(norm(b-A*x)/norm(b))]);
disp(['Iterations: ',num2str(iter)]);
disp(['CPU time: ',num2str(time),'s.']);
disp(' ');

s = 4;
t = cputime;
disp('IDR(4) iteration...');
[x, flag, relres, iter, resvec] = idrs( A, b, s, tol, maxit  );
time = cputime - t;
resvec = log10(resvec/resvec(1));
figure(fig);
it = [0:1:length(resvec)-1];
plot(it,resvec,'r-*');
drawnow;
disp(['Final accuracy: ', num2str(norm(b-A*x)/norm(b))]);
disp(['Iterations: ',num2str(iter)]);
disp(['CPU time: ',num2str(time),'s.']);
disp(' ');

s = 8;
t = cputime;
disp('IDR(8) iteration...');
[x, flag, relres, iter, resvec] = idrs( A, b, s, tol, maxit );
time = cputime - t;
resvec = log10(resvec/resvec(1));
figure(fig);
it = [0:1:length(resvec)-1];
plot(it,resvec,'r-s');
drawnow;
disp(['Final accuracy: ', num2str(norm(b-A*x)/norm(b))]);
disp(['Iterations: ',num2str(iter)]);
disp(['CPU time: ',num2str(time),'s.']);
disp(' ');

legend('GMRES', 'Bi-CG', 'IDR(1)', 'IDR(2)', 'IDR(4)', 'IDR(8)' );
hold off;
```

C.4 Sorteig.m

```
function [EV2,EW2] = sorteig(H)
% takes a square matrix H as input. the output is a diagonal matrix
% EW2 with the eigenvalues on the diagonal from smallest tot largest
% and a matrix EV with the corresponding eigenvectors

[EV EW] = eig(H);

EW2 = diag(sort(diag(EW),'ascend'));
[c, ind]=sort(diag(EW),'ascend');       % store the indices of which columns
                                        % the sorted eigenvalues come from
EV2=EV(:,ind);                          % arrange the columns in this order
```