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

PIDR(s): 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
June 24, 2014

MSc Thesis Science Education and Communication

"PIDR(s): 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, MSc

June 24, 2014

Delft, the Netherlands

Summary

The Induced Dimension Reduction(s) method (or the IDR(s) method) is an example of an iterative method used for solving systems of linear equations. Projection methods are a special type of iterative method. They find an approximate solution in a subspace $\mathcal{K} \in \mathbb{C}^n$ (the right subspace) by requiring that the residual is orthogonal to another subspace $\mathcal{L} \in \mathbb{C}^n$ (the left subspace). In this thesis we investigate how we can implement IDR(s) as a projection method. We call this method PIDR(s), which stands for Projected IDR(s). We present an implementation of PIDR(s) for solving systems of linear equations and for solving eigenvalue problems. These implementations are not meant to be optimal, but they are used to show that IDR(s) can indeed be seen as a projection method.

In chapter 3 we explore the theory of projection methods and Krylov subspace methods: projection methods for which the vectors in the right subspace can be written as a linear combination of the vectors $A^j r_0$ with $j = 0, 1, 2, \ldots$. We derive a general algorithm for projection methods, which we will use as a basis for the PIDR(s) method in chapter 6. In chapter 4 we investigate a selection of Krylov subspace methods and we show how they fit in the framework of projection methods. For each method we give the corresponding left and right subspace.

In chapter 5 we show how we can derive the IDR(s) method from a general Krylov-type solver. In addition, we analyse its performance and we present four numerical experiments that show that for certain problems IDR(s) is a good choice to consider. In chapter 6 we derive the definitions of the left and right subspace of PIDR(s) using information from existing literature. Using the general algorithm for projection methods in chapter 3 and the newly acquired definitions of the left and right subspace, we present two algorithms for PIDR(s): one for solving eigenvalue problems and one for solving systems of linear equations.

The numerical experiments in chapter 7 show that we have a working algorithm for the PIDR(s) method for systems of linear equations and for eigenvalue problems. The algorithm of PIDR(s) for eigenvalue problems is particularly important, since not much is known about IDR(s) as an eigenvalue method. By investigating this problem, we can develop a thorough theoretical framework for IDR(s) as an eigenvalue method, which will give us a better understanding of IDR(s) itself.

Preface

Monday the 6th of September 2004 now lies almost ten years in the past. It was the day I started studying at the Delft University of Technology. In 2005 I switched to studying mathematics. I will not bother you what happened between 2004 and this moment, but right now I'm on the brink of graduating and I must say that I can look back on ten satisfying years. Moreover, I can look forward to what I like most about mathematics, which is teaching it to the generation of tomorrow.

I want to thank a few people for supporting me throughout my graduation project. First of all I would like to thank my thesis committee members (Kees Vuik, Martin van Gijzen, Jeroen Spandaw and Reinaldo Astudillo) for the time they spent reading my literature research and this thesis. Secondly, I want to thank Martin van Gijzen and Reinaldo Astudillo in particular for their support and numerous comments and suggestions.

Marijn Schreuders,
Rotterdam, June 2014

List of Figures

List of Tables

Table of Contents

Chapter 1

Introduction

In secondary school we learn to solve small (two or sometimes three variables) systems of linear equations by hand. However, in scientific computing applications we rarely encounter such small problems. Often we want to solve systems of linear equations with millions of variables. Rather than trying to solve these systems by hand (an impossible task), we let a computer solve such problems. The study of algorithms for performing linear algebra computations on computers is called numerical linear algebra. It has broad applicability in a wide variety of technical areas, such as electrical engineering, aerospace engineering, signal processing, computer science, physics, communication and economics.

1.1 Krylov subspace methods

Within the field of numerical linear algebra, *iterative methods* methods play an important role. Iterative methods generate a sequence of vectors (the approximate solutions) that converges to the exact solution under certain conditions. This is in contrast to direct methods (such as Gaussian elimination), which find an exact solution in a finite number of steps. The field of iterative methods underwent great progress in the 1950s with the introduction of the first computers. Methods such as the Lanczos method (1950) [9], the Arnoldi method (1951) [1] and the Conjugate Gradient (CG) method (1952) [8] made it possible to solve large scale problems on a computer that were impossible to solve before. Later, methods like Bi-CG (1976) [4], GMRES (1986) [11] and Bi-CGSTAB (1992) [18] contributed to the success of numerical linear algebra.

If the matrix corresponding to a system of linear equations is symmetric and positive definite (SPD), we prefer to use the Conjugate Gradient method, since it minimises the residual and uses short recurrences. Here, short recurrences means that we only need a few of the previous approximate solutions to compute a new solution. Unfortunately, there is no such method if the matrix corresponding to a system of linear equations is nonsymmetric. The search for new iterative methods for nonsymmetric problems has taken two approaches. 'GMRES-type' methods minimise the residual in every iteration. Hence, the solution converges in as few iterations as possible. However, the required memory grows for an increasing number of iterations and often we cannot afford to run the full algorithm. 'Bi-CG-type' methods use short recurrences, which means that the amount of work per iteration does not increase if the dimension of the problem grows. However, we have no optimality condition for the residual and hence, convergence may be slower.

Most of the iterative techniques for solving large systems of linear equations are examples of *projection methods*. In a projection method we try to find an approximate solution in a subspace $\mathcal{K}$ of $\mathbb{C}^n$ (the right subspace), such that the residual is orthogonal to another subspace $\mathcal{L}$ of $\mathbb{C}^n$ (the left subspace). Different choices of $\mathcal{K}$ and $\mathcal{L}$ give rise to different projection methods. A projection method for which the right subspace (of size m) equals $span\{r_0, Ar_0, A^2r_0 \ldots, A^{m-1}r_0\}$ is called a *Krylov subspace method*, named after the Russian mathematician Alexei Krylov. Krylov subspace methods work by building a basis for the Krylov subspace and by constructing the approximate solution as a linear combination of the vectors in this basis. Moreover, Krylov subspace methods can be used to solve eigenvalue problems. Hence, they form a broad class of iterative methods (see for example [5] and [10]).

1.2 Research goals

The Induced Dimension Reduction (IDR) method was originally proposed by Peter Sonneveld in 1980 [20]. It is a Krylov subspace method for nonsymmetric matrices that tries to combine an optimality condition for the residual with reasonably short recurrences. IDR generates residuals that are forced to be in certain subspaces of decreasing dimension. In the years after, research focussed on Bi-CG-type methods like Bi-CGSTAB and CGS and hence IDR has always been overshadowed by these methods. This is quite unfortunate, because although there is a clear relation between Bi-CG-type methods and IDR, the underlying mathematical ideas are completely different. By exploiting these differences, better methods can be developed. Fortunately, there has been renewed interest in the IDR method in the past few years. This renewed attention has led to the IDR(s) method, proposed by Peter Sonneveld and Martin van Gijzen [16]. Since IDR(s) is in essence an iterative method, we can ask ourselves if it is possible to see IDR(s) as a projection method. If this is the case, we might be able to better understand the IDR(s) method.

In their paper 'Interpreting IDR as a Petrov-Galerkin method' [12], Valeria Simoncini and Daniel B. Szyld show that the IDR(s) method can indeed be seen as a projection method over the Krylov subspace. When the left subspace and the right subspace are appropriately chosen, we see that the IDR(s) method can be interpreted as a classical Krylov subspace method that fits in the framework of projection methods. However, Simoncini and Szyld do not provide an implementation of IDR(s) as a projection method and in their paper the right subspace is not explicitly defined. Hence, we ask ourselves the following question:

How can we implement IDR(s) as a projection method?

The goal of this project is to develop a working algorithm for IDR(s) as a projection method. We will refer to this method as the PIDR(s) method. We will give an algorithm of PIDR(s) for solving systems of linear equations and for solving eigenvalue problems. To do this, we will first need to identify the left and the right subspace.

1.3 Structure of this thesis

Chapter 2 describes the definitions that we use frequently in 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 algorithm of projection methods. Finally it explains what Krylov subspaces are and how they are related to projection methods.

Chapter 4 will make the reader acquainted with a selection of Krylov subspace methods. These can be classified into methods that solve eigenvalue problems and methods that solve systems of linear equations. We will present the Arnoldi method, the Lanczos method and the Bi-Lanczos method as examples of methods that solve eigenvalue problems. Methods that solve systems of linear equations include the Full Orthogonalisation Method (FOM), the Generalised Minimal Residual (GMRES) method, the Conjugate Gradient (CG) method and the Biconjugate Gradient (Bi-CG) method. Moreover, chapter 4 explains how all these methods can be seen in the framework of projection methods.

In chapter 5 we will explain how we can derive the IDR(s) algorithm. We will describe how the IDR(s) algorithm works. Finally, we will illustrate its performance by conducting four numerical tests. In chapter 6 we arrive at the core of this thesis. We will derive the left and right subspace of PIDR(s) for systems of linear equations and for eigenvalue problems, together with an algorithm and an implementation. In chapter 7 we will present six motivating examples of the performance of PIDR(s). In chapter 8 we will present our findings and we will give recommendations for future research.

Chapter 2

Definitions

In this chapter we will present the relevant definitions for this thesis.

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 $A \in \mathbb{R}^{n \times n}$ if a nonzero vector $u \in \mathbb{C}^n$ exists such that $Au = \lambda u$. The vector u is called an eigenvector of A associated with λ.

Definition 2.5 (Symmetric Positive Definite)**.**

A matrix $A \in \mathbb{R}^{n \times n}$ is said to be Symmetric Positive Definite or SPD if it satisfies

(i) $A^T = A$,

(ii) $u^T A u > 0 \ \forall u \in \mathbb{R}^n$, $u \neq 0$.

Definition 2.6 (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:

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

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

Definition 2.7 (Conjugate transpose)**.**
The conjugate transpose of a matrix $A \in \mathbb{C}^{m\times n}$ is a matrix $A^* \in \mathbb{C}^{n\times m}$ such that

$$A^*_{ij} = \overline{A_{ji}},$$

where $\overline{A}$ denotes complex conjugate of A.

Definition 2.8 (Spectrum)**.**
The spectrum of a matrix $A \in \mathbb{R}^{n\times n}$ is the set of all its eigenvalues λ:

$$\sigma(A) = \{\lambda \in \mathbb{C} \ : \ A - \lambda I = 0\}.$$

Definition 2.9 (Rational Krylov subspace)**.**
Let $q_{m-1}(A)$ be a polynomial in A of degree $(m-1)$ of the form

$$q_{m-1}(A) = \prod_{i=1}^{m-1} \left(I - \frac{1}{\mu_i} A\right),$$

where $A \in \mathbb{R}^{n\times n}$ and $\mu_i \in \mathbb{R}^n \setminus \sigma(A)$.

A rational Krylov subspace [7] corresponding to A and an initial vector $v \in \mathbb{R}^n$ is a Krylov subspace for which

$$\mathcal{K}_m(A, v) = q_{m-1}(A)^{-1} span\{v, Av, \dots, A^{m-1}v\}.$$

Definition 2.10 (Block Krylov subspace)**.**
A block Krylov subspace corresponding to a matrix $A \in \mathbb{R}^{n\times n}$ and initial matrix $P \in \mathbb{R}^{n\times s}$ is a Krylov subspace for which the matrix A operates on P:

$$\mathcal{K}_m(A, P) = span\{P, AP, \dots, A^{m-1}P\}.$$

Chapter 3

Projection methods

In this chapter we will explore the area of projection methods. For the information in this chapter, we rely heavily on the book 'Iterative methods for sparse linear systems', written by Yousef Saad [10], which is considered an influential book in the field numerical linear algebra.

Consider the system of linear equations

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

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

In contrast to direct methods, iterative methods generate a sequence of vectors that under certain conditions converge to the exact solution. If this sequence of solutions converges to the exact solution, an iterative method is said to be `convergent`. 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*, *search subspace* or the *right subspace*. In order to find this approximation, 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 form a basis for another subspace $\mathcal{L}_m$ with dimension m and it is called the *subspace of constraints* or *left subspace*. There are two kinds of projection methods: *orthogonal* projection methods (also called Galerkin methods) and *oblique* projection methods (also called Petrov-Galerkin methods). In orthogonal projection methods the right subspace $\mathcal{K}_m$ is equal to the left subspace $\mathcal{L}_m$ and in oblique projection methods $\mathcal{L}_m$ and $\mathcal{K}_m$ differ. Throughout the rest of this thesis, we use the symbol $\mathcal{K}_m$ to denote the right subspace and $\mathcal{L}_m$ to denote the left subspace.

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 $r_m = b - Ax_m$ as the residual. A projection method onto $\mathcal{K}_m$ 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 the residual, the inner product $(r_0 - A\delta, w) = 0$. The solution to equation (3.1) can now be defined as:

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

We now have to solve $(r_0 - A\delta, w)$ for δ in order to find the approximate solution x_m. In each iteration, the newly calculated residual should be orthogonal to the left subspace $\mathcal{L}_m$. Figure 3.1 illustrates this orthogonality condition [10, p. 134].

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.

$$V_m y_m \quad = \quad [v_1, v_2, \ldots v_m] y_m \quad = \quad y_1 v_1 + y_2 v_2 + \ldots y_m v_m$$

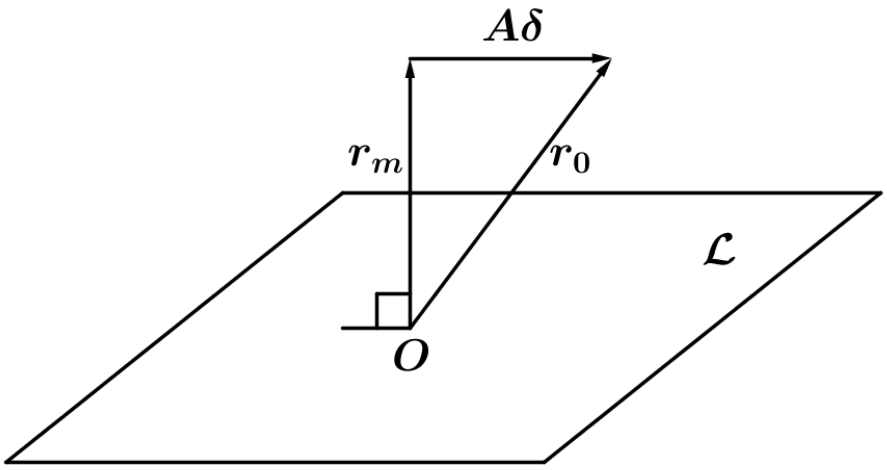

Figure 3.1: Interpretation of the orthogonality condition

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

$$r_m = r_0 - AV_m y_m. \tag{3.7}$$

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 $W_m^T AV_m$ is nonsingular (invertible), then we have an explicit solution for y_m and, since x_m is a function of y_m, also for x_m:

$$y_m = \left(W_m^T AV_m\right)^{-1} W_m^T r_0 \tag{3.8}$$

$$x_m = x_0 + V_m \left(W_m^T AV_m\right)^{-1} W_m^T r_0 \tag{3.9}$$

Equation (3.8) and (3.9) gives rise to algorithm 3.1, which is a general algorithm for projection methods that solve systems of linear equations [10]. We have assumed that the matrix $W_m^T AV_m$ is nonsingular, but this might not always be the case. However, there are two important cases in which the nonsingularity of $W_m^T AV_m$ is guaranteed [10, p. 136]. Theorem 3.1 states these cases:

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 $W^T AV$ is nonsingular for any bases V_m and W_m of $\mathcal{K}_m$ and $\mathcal{L}_m$ respectively.

Proof. See [10, p. 136].

Algorithm 3.1 General projection method for systems of linear equations

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

3.3 The projector of a projection process

For each projection method we can define a projector [10, p. 34]:

Definition 3.2.
A projector $P : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ is any linear mapping which is idempotent, i.e., such that

$$P^2 = P.$$

We can find the projector corresponding to projection methods by rewriting the residual. First we substitute equation (3.8) in equation (3.7). We obtain:

$$r_m = r_0 - A V_m y_m = \left(I - A V_m \left(W_m^T A V_m\right)^{-1} W_m^T\right) r_0.$$

We define P as

$$P = I - A V_m \left(W_m^T A V_m\right)^{-1} W_m^T.$$

Hence, we have $r_m = P r_0$, which means that we can find the m-th residual by projecting r_0. Writing out definition 3.2 yields:

$$\begin{aligned} P^2 &= I^2 - 2 A V_m \left(W_m^T A V_m\right)^{-1} W_m^T + A V_m \left(W_m^T A V_m\right)^{-1} W_m^T A V_m \left(W_m^T A V_m\right)^{-1} W_m^T \\ &= I - 2 A V_m \left(W_m^T A V_m\right)^{-1} W_m^T + A V_m \left(W_m^T A V_m\right)^{-1} W_m^T \\ &= I - A V_m \left(W_m^T A V_m\right)^{-1} W_m^T \\ &= P. \end{aligned}$$

We see that the matrices $W_m^T A V_m$ and $(W_m^T A V_m)^{-1}$ cancel.

3.4 Projection methods for eigenvalue problems

For the eigenvalue problem we want to find the eigenvectors $u^{(i)} \neq 0$ in $\mathbb{C}^n$ and the corresponding eigenvalues $\lambda^{(i)} \in \mathbb{C}$ of a square matrix $A \in \mathbb{R}^{n \times n}$ such that $Au^{(i)} = \lambda^{(i)} u^{(i)}$ for $i = 1, \ldots, n$. For an $n \times n$ matrix, there are n eigenvalues (not necessarily distinct). However, most of the times we are only interested in (a few of) the largest or smallest eigenvalues.

In a projection method for eigenvalue problems we want to find (in the m-th iteration) the eigenvectors $u_m^{(i)}$ in the right subspace (for $i = 1, \ldots, m$) and their corresponding eigenvalues $\lambda_m^{(i)}$ such that the left subspace is orthogonal to the residual $r_m^{(i)}$:

$$\text{Find} \quad \lambda_m^{(i)} \in \mathbb{R} \text{ and } u_m^{(i)} \in \mathcal{K}_m \qquad \text{such that} \qquad r_m^{(i)} \perp \mathcal{L}_m. \tag{3.10}$$

Let V_m be a basis for $\mathcal{K}_m$ and let W_m be a basis for $\mathcal{L}_m$. Since $u_m^{(i)} \in \mathcal{K}_m$ for $1 \leq i \leq m$, we can write $u_m^{(i)} = V_m y_m^{(i)}$. Here, $y_m^{(i)}$ contains the coefficients for the column vectors of V_m. If we write out the orthogonality condition, we obtain:

$$\begin{aligned} W_m^T r_m^{(i)} = 0 \quad &\Leftrightarrow \quad W_m^T \left(A u_m^{(i)} - \theta_m^{(i)} u_m^{(i)} \right) = 0 \\ &\Leftrightarrow \quad W_m^T A V_m y_m^{(i)} - \theta_m^{(i)} V_m y_m^{(i)} = 0 \\ &\Leftrightarrow \quad \left(W_m^T A V_m \right) y_m^{(i)} = \theta_m^{(i)} \left(W_m^T V_m \right) y_m^{(i)} \end{aligned}$$

Here the $\theta_m^{(i)}$ denote the eigenvalues of $W_m^T A V_m$. The corresponding eigenvectors are $W_m^T V y_m^{(i)}$.

In the last line we have a generalised eigenvalue problem for the matrices $W_m^T A V_m$ and $W_m^T V_m$. As we will see in chapter 4, the generalised eigenvalue problem reduces to an ordinary eigenvalue problem if V_m and W_m are biorthogonal. The matrices $W_m^T A V_m$ and $W_m^T V_m$ have the same dimension, which is generally much smaller than the dimension of A. Hence, computing the eigenvalues of $W_m^T A V_m$ has a relative low cost. As we will explain in more detail in section 4.1, the Ritz values of $W_m^T A V_m$ are approximations of the eigenvalues of A and the eigenvectors of A can be approximated by the Ritz vectors $V_m y_m$, where y_m is the eigenvector of $W_m^T A V_m$. In algorithm 3.2 we see a pseudoformal code of a general projection method for eigenvalue problems.

Algorithm 3.2 General projection method for eigenvalue problems

1: Select a pair of subspaces $\mathcal{K}_m$ and $\mathcal{L}_m$
2: Until convergence; **Do**
3: Build bases $V_m = [v_1, v_2, \ldots, v_m]$ for $\mathcal{K}_m$ and $W_m = [w_1, w_2, \ldots, w_m]$ for $\mathcal{L}_m$
4: Solve $W_m^T A V_m y_m = \theta_m W_m^T V_m y_m$ and approximate the eigenvalues with θ_m
5: and the eigenvectors with $V_m y_m$.
6: Check stopping criterion
7: **EndDo**

3.5 Definition of the Krylov subspace

Section 3.1 explained that a projection method searches for an approximate solution in the right subspace $\mathcal{K}_m$ such that the residual is orthogonal to all vectors in the left subspace $\mathcal{L}_m$. In Krylov subspace methods, the subspace $\mathcal{K}_m$ is a Krylov subspace corresponding to the matrix A and the initial residual r_0:

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, A^2r_0, \ldots, A^{m-1}r_0\}, \tag{3.11}$$

r_0 is the initial residual and the vectors $r_0, Ar_0, A^2r_0, \ldots, A^{m-1}r_0$ are 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 Generalised Minimum Residual method (GMRES, see section 4.2.2) and the Conjugate Gradient method (CG, see section 4.2.3). 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 $\mathcal{L}_m = \mathcal{K}_m(A^T, r_0)$.

Recall that we can write $x_m = x_0 + \mathcal{K}_m$ and we can write x_m as x_0 plus a linear combination of the first m Krylov vectors or simply as x_0 plus a polynomial p_{m-1} in A of degree $(m-1)$, multiplied by r_0:

$$x_m = x_0 + \left(\sum_{j=0}^{m-1} \alpha_j A^j \right) r_0 = x_0 + p^{m-1}(A) r_0,$$

with $\alpha \in \mathbb{R}$ and $A^0 = I$. If we have $x_0 = 0$, $r_0 = b$ and we obtain: $x_m = p^{m-1}(A)b$.

We can obtain a similar expression for the residuals. From 3.7 we have $r_m = r_0 - A(x_m - x_0)$. 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, we know $r_m \in \mathcal{K}_{m+1}$ and we can write r_m as a linear combination of the first $(m+1)$ Krylov vectors or simply as a polynomial q_m of degree m in A multiplied by r_0:

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

Chapter 4

Krylov subspace methods

Numerical linear algebra is often concerned with two kinds of problems: finding the eigenvalues of a matrix (eigenvalue problems) and solving a system of (linear) equations.

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 system of linear equations. There is a wide variety of methods available to solve either problem. Krylov subspace methods are one of those. Krylov subspace methods can be classified, depending on the kind of problem we want to solve and the characteristics of A. Chapter 4 will discuss several of these methods. Section 4.1 will discuss different Krylov subspace methods to solve eigenvalue problems and section 4.2 will discuss several Krylov subspace methods to solve systems of linear equations. Figure 4.1 shows the methods that will be discussed and shows their classification according to the problem and matrix properties.

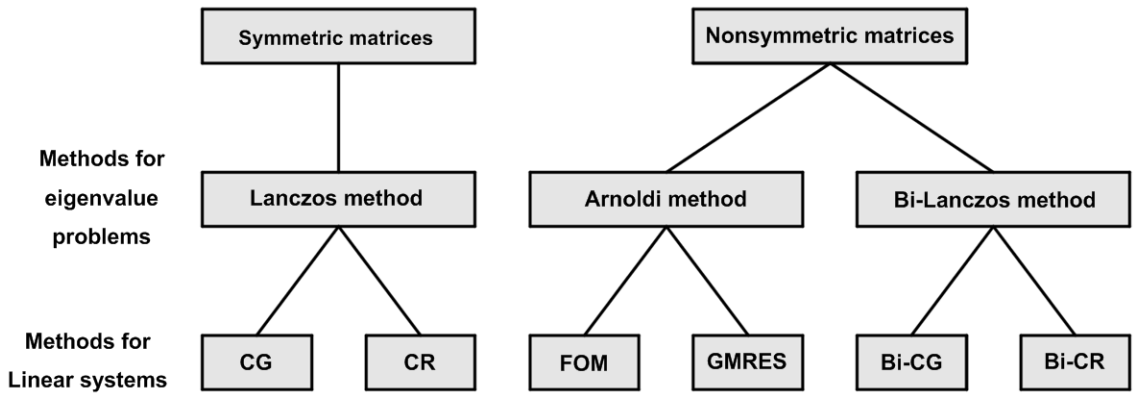

Figure 4.1: Krylov subspace methods

4.1 Krylov subspace methods for eigenvalue problems

In an eigenvalue problem, we want to find the eigenvalues $\lambda^{(i)} \in \mathbb{C}$ and corresponding eigenvectors $u^{(i)} \neq 0$ in $\mathbb{C}^n$ of a matrix $A \in \mathbb{R}^{n \times n}$ for $i = 1, \ldots, n$. The eigenvalues can be found by

solving the system of linear equations $\det(A - \lambda^{(i)} I) = 0$. Here I denotes the $(n \times n)$ identity matrix. The eigenvector $u^{(k)}$ corresponding to a particular eigenvalue $\lambda^{(k)}$ can be found by solving the equation $(A - \lambda^{(k)} I)u^{(k)} = 0$. However, this is often an expensive calculation, since A can be a large matrix. Moreover, we are often interested in only a few of the extreme eigenvalues. 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 is an example of a Krylov subspace method. It approximates the eigenvalues of a general matrix $A \in \mathbb{R}^{n \times n}$. It was proposed by Walter Edwin Arnoldi in 1951 [1]. 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 approximations to a subset of the eigenvalues of A. This is accomplished by building an orthonormal basis of vectors $V_m = [v_1, \ldots v_m]$ for the Krylov subspace $\mathcal{K}_m$, with

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

In iteration j, an extra vector v_j is added to the basis. Since H_m is smaller than A, the eigenvalues are inexpensive to compute. Algorithm shows one possible implementation of the Arnoldi method (see also appendix A.1.

Algorithm 4.1 The Arnoldi Method

1: Choose an initial vector v_1 such that $||v_1|| = 1$
2: For $j = 1, 2, \ldots, m$ **Do**
3: $\quad w_j := Av_j$
4: $\quad$ For $i = 1, 2, \ldots, j$ **Do**
5: $\quad\quad h_{ij} = (w_j, v_i)$
6: $\quad\quad w_j := w_j - h_{ij} v_i$
7: $\quad$ **EndDo**
8: $\quad h_{j+1,j} = ||w_j||_2$
9: $\quad$ **If** $h_{j+1,j} = 0$
10: $\quad\quad$ Stop
11: $\quad$ **EndIf**
12: $\quad v_{j+1} = w_j / h_{j+1,j}$
13: $\quad$ Build the Hessenberg matrix H_j and calculate its eigenvectors
14: $\quad$ Check stopping criterion
15: **EndDo**
16: Approximate the eigenvalues and eigenvectors of A using the upper
17: 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. Each subsequent basis vector v_{j+1} $(j = 1, \ldots, m)$ is calculated by multiplying the previous vector v_j with A (line 3) and orthogonalising it with

respect to all the previous basis vectors using the modified Gram-Schmidt process (lines 5-6). Finally we normalise the resulting vector (line 12) [10, p. 12]. When the stopping criterion is satisfied, the algorithm calculates the Ritz values $\theta_m^{(i)}$ and the eigenvectors $y_m^{(i)}$ of H_m with $i = 1, \ldots, m$.

For the stopping criterion, we use $||r_j|| < TOL$, where $||r_j|| := ||Au_j^{(i)} - \lambda_j^{(i)} u_j^{(i)}||,\ i = 1, \ldots, j$, is the residual in the j-th iteration. Note that this stopping criterion is expensive to compute, since we have to compute a matrix-vector product in every 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:

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

where $V_m \in \mathbb{C}^{n \times m}$ is a matrix with orthonormal columns that form a basis for $\mathcal{K}_m$, $H_m \in \mathbb{R}^{m \times m}$ is an upper Hessenberg matrix, $\overline{H}_m \in \mathbb{R}^{(m+1) \times j}$ is an upper Hessenberg matrix with one extra row 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 $i = 1, \ldots, j$):

$$\begin{aligned} ||r_j|| &= ||Au_j^{(i)} - \lambda_j^{(i)} u_j^{(i)}|| \\ &= ||AV_j y_j^{(i)} - \theta_j^{(i)} V_j y_j^{(i)}|| \\ &= ||V_j H_j y_j^{(i)} + h_{j+1,j} v_{j+1} e_j^T y_j^{(i)} - \theta_j^{(i)} V_j y_j^{(i)}|| \\ &= ||V_j (H_j y_j^{(i)} - \theta_j^{(i)} y_j^{(i)}) + h_{j+1,j} v_{j+1} y_j^{(i)}(j)|| \\ &= ||h_{j+1,j} v_{j+1} y_j^{(i)}(j)|| \\ &= |h_{j+1,j}| \cdot |y_j^{(i)}(j)|. \end{aligned}$$

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

$$|h_{j+1,j}| \cdot |y_j^{(i)}(j)| < TOL. \tag{4.6}$$

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 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.

The Ritz values of H_m are approximations to a subset of the eigenvalues of A. They converge to the extreme eigenvalues. The eigenvectors $u_m^{(i)}$ of A can be approximated by the Ritz vector $V_m y_m^{(i)}$. We can show this by rewriting equation (4.4) to $V_m H_m + r_m e_m^T$. Here we have used that $h_{m+1} = ||r_m||$ and $v_{m+1} = r_m/||r_m||$. If the Ritz values are close to the eigenvalues, the residual is small and hence $AV_m \approx V_m H_m$. In the m-th iteration we have the following relation

$$Au_m^{(i)} = AV_m y_m^{(i)} \approx V_m H_m y_m^{(i)} = V_m \theta_m^{(i)} y_m^{(i)} = \theta_m^{(i)} V_m y_m^{(i)} = \theta_m^{(i)} u_m^{(i)}. \tag{4.7}$$

This shows that the Ritz values are approximations to the eigenvalues of A. 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 symmetric.

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:

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

When we transpose both sides, we get

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

Since A is symmetric, we have $A = A^T$. Hence, equation (4.9) can be written as

$$V_m^T A V_m = H_m^T. \tag{4.10}$$

Since the right-hand sides of equations (4.8) and (4.10) are the same, we have $H_m^T = H_m$. Since H_m is an upper Hessenberg matrix and $H_m T$ a lower Hessenberg matrix, we must have that H_m is a tridiagonal matrix.

■

4.1.2 The Lanczos method

The Lanczos method [9][10] 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 and positive definite (SPD). The Lanczos method was named after Cornelius Lanczos, a Hungarian mathematician. Since A is an SPD matrix, the eigenvalues of A are real and positive. The Lanczos algorithm is useful in situations where a few of A's largest or smallest eigenvalues are desired. Just as the Arnoldi method, it builds an orthonormal basis V_m for the Krylov subspace $\mathcal{K}_m$ (defined in equation (3.11)). The Lanczos method also produces a tridiagonal matrix T_m. Algorithm 4.2 shows one possible implementation of the Arnoldi method (see also appendix A.2).

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: $\quad w_j := Av_j - \beta_j v_{j-1}$
5: $\quad \alpha_j := (w_j, v_j)$
6: $\quad w_j := w_j - \alpha_j v_j$
7: $\quad \beta_{j+1} := ||w_j||_2$
8: $\quad$ **If** $\beta_{j+1} = 0$
9: $\quad\quad$ Stop
10: $\quad$ **EndIf**
11: $\quad v_{j+1} := w_j / \beta_{j+1}$
12: $\quad$ 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: $\quad$ Check stopping criterion
14: **EndDo**
15: Approximate the eigenvalues and eigenvectors of A using the tridiagonal
16: matrix T_m and the orthonormal basis V_m.

In line 4-6 the algorithm finds a new search direction orthogonal to all search directions of the previous vectors v and in line 11 normalisation 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 tridiagonal matrix $T_j \in \mathbb{R}^{j \times j}$ in every iteration. T_j takes the following form:

$$T_j = \begin{pmatrix} \alpha_1 & \beta_2 & & & & \\ \beta_2 & \alpha_2 & \beta_3 & & O & \\ & \beta_3 & \alpha_3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & O & & \ddots & \ddots & \beta_j \\ & & & & \beta_j & \alpha_j \end{pmatrix}. \tag{4.11}$$

When the stopping criterion is satisfied (after m iterations), the Lanczos algorithm calculates

the Ritz values $\theta_m^{(i)}$ and the eigenvectors $y_m^{(i)}$ of T_m with $i = 1, \ldots, m$. The Ritz values of T_m are approximations to a subset of the eigenvalues of A (see section 4.1.1). The eigenvectors $u_m^{(i)}$ of A can be approximated by the Ritz vector $V_m y_m^{(i)}$ for $i = 1, \ldots, m$.

We can formulate a low-cost 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

$$Av_j = \beta_j v_{j-1} + a_j v_j + \beta_{j+1} v_{j+1} \qquad \text{for } j = 1, \ldots, m. \tag{4.12}$$

From this expression it is easy to see that an orthonormal basis can be build using only three vectors in every 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 different from 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 (4.12)). However, we will need all the basis vectors to approximate the eigenvectors of A. Equation (4.12) can be written in matrix-vector notation as

$$AV_m = V_m T_m + \beta_{m+1} v_{m+1} e_m^T. \tag{4.13}$$

Note that equation (4.13) 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:

$$|\beta_{j+1}| \cdot |y_j^{(i)}(j)| < TOL.$$

4.1.3 The Lanczos Biorthogonalisation method

Although the Arnoldi method has good properties (it is a stable method with respect to rounding errors and breakdown does not occur, [19]), it does have 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 every 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 [19, p.107]

The Lanczos Biorthogonalisation method [9][10], also called the Bi-Lanczos method or non-symmetric 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 the bases for the two subspaces $\mathcal{K}_m(A, v_1)$ and $\mathcal{L}_m(A^T, w_1)$:

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

Algorithm 4.3 shows the Lanczos Biorthogonalisation method in pseudoformal notation. The implementation of Bi-Lanczos can be found in appendix A.3. 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 search directions of the previous vectors v and in line 12 and 13 normalisation takes place. Next, the algorithm builds a tridiagonal matrix $T_j \in \mathbb{R}^{j \times j}$ in each iteration. T_j has the following form:

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: $\quad \alpha_j := (w_j, v_j)$
5: $\quad \hat{v}_{j+1} = Av_j - \alpha_j v_j - \beta_j v_{j-1}$
6: $\quad \hat{w}_{j+1} = A^T w_j - \alpha_j w_j - \delta_j w_{j-1}$
7: $\quad \delta_{j+1} := |(\hat{v}_{j+1}, \hat{w}_{j+1})|^{1/2}$
8: $\quad$ **If** $\delta_{j+1} = 0$
9: $\quad\quad$ Stop
10: $\quad$ **EndIf**
11: $\quad \beta_{j+1} := (\hat{v}_{j+1}, \hat{w}_{j+1})/\delta_{j+1}$
12: $\quad v_{j+1} = \hat{v}_{j+1}/\delta_{j+1}$
13: $\quad w_{j+1} = \hat{w}_{j+1}/\beta_{j+1}$
14: $\quad$ 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: $\quad$ Check stopping criterion
16: **EndDo**
17: Approximate the eigenvalues and eigenvectors of A using the tridiagonal
18: matrix T_m and the basis V_m.

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

Note that $\beta_{j+1} = \pm\delta_{j+1}$. This is easily deduced from 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 $\theta_m^{(i)}$ and the eigenvectors $y_m^{(i)}$ of T_m with $i = 1, \ldots, m$. The Ritz values of T_m are approximations to a subset of the eigenvalues of A. The eigenvectors $u_m^{(i)}$ of A can be approximated by the Ritz vectors $V_m y_m^{(i)}$ (see section 4.1.1).

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 an inexpensive stopping criterion for the Bi-Lanczos method in a similar fashion as in the Arnoldi and Lanczos method. However, in the Bi-Lanczos method, the vectors $v_1, \ldots v_j$ are not orthonormal ($||v_{j+1}|| \neq 1$). Hence, we obtain:

$$|\delta_{j+1}| \cdot |y_j^{(i)}(j)| \cdot ||v_{j+1}|| < TOL. \tag{4.16}$$

4.2 Krylov subspace methods for solving systems of linear equations

Suppose we are interested in solving the system of linear equations $Ax = b$ with $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$ and initial guess x_0. Let $V_j = [v_1, \ldots, v_j]$ and $W_j = [w_1, \ldots, w_j]$ be orthonormal bases for the Krylov subspaces $\mathcal{K}_j$ and $\mathcal{L}_j$ respectively. We can write the solution in the j-th iteration as $x_j = x_0 + V_j y_j$, where y_j is a vector with coefficients for the vector v_j. In section 3.2 we found the following relations for y_m and x_m:

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

Since $v_1 = r_0/||r_0||$ and $||r_0|| = \beta$ (and hence $r_0 = \beta v_1$), we have:

$$W_m^T r_0 = \beta W_m^T \cdot v_1 = \beta e_1, \tag{4.17}$$

where e_1 is the first unit vector. In the last equality sign we used that $W_m^T v_1 = e_1$. If $W_m = V_m$ (in orthogonal projection methods) we obtain $V_m^T v_1$ and this is clearly equal to e_1. In case V_m and W_m are biorthogonal, we obtain the same relation. We now have:

$$y_m = \left(W_m^T A V_m\right)^{-1} \beta e_1 \tag{4.18}$$

$$x_m = x_0 + V_m \left(W_m^T A V_m\right)^{-1} \beta e_1. \tag{4.19}$$

Krylov subspace methods for systems of linear equations are classified in two categories. First, the 'GMRES-type' category, which contains methods based on the Arnoldi method. Hence they are long recurrence algorithms. Secondly, we have the 'BI-CG-type' category, which contains short recurrence methods based on the Bi-Lanczos method. In the next sections we present an overview of the most commonly used Krylov subspace methods for solving systems of linear equations.

4.2.1 The Full Orthogonalisation Method (FOM)

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

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

We can find a solution to a system of linear equations $Ax = b$ by using equation (4.18) and (4.19). Algorithm 4.4 shows one possible implementation of the FOM 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: Check stopping criterion
15: **EndDo**
16: $x = x_0 + V_m y_m$

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 finishes (after m iterations), both V_m and y_m are used to compute the approximate solution x_m in line 16. For the stopping criterion, we use equation (4.20) with $\epsilon = TOL$ [19, p. 56]:

$$\frac{||r_j||}{||b||} = \frac{||b - Ax_j||}{||b||} < TOL. \tag{4.20}$$

As in the Arnoldi method, it is not efficient to compute the residual directly. Instead we substitute the approximate solution into equation (4.20) and using equation (4.4) in the second line and equation (4.18) in the third line with $W_m^T A V_m = H_j$, we find:

$$
\begin{aligned}
||r_j|| &= ||r_0 - AV_j y_j|| \\
&= ||\beta v_1 - V_j H_j y_j - h_{j+1,j} v_{j+1} e_j^T y_j|| \\
&= ||\beta V_j e_1 - \beta V_j e_1 - h_{j+1,j} v_{j+1} y_j(j)|| \\
&= ||h_{j+1,j} v_{j+1} y_j(j)|| \\
&= |h_{j+1,j}| \cdot |y_j(j)|.
\end{aligned}
$$

Substituting this into equation (4.20) and rewriting it, we find the following stopping criterion:

$$|h_{j+1,j}| \cdot |y_j(j)| < ||b|| \cdot TOL.$$

4.2.2 The Generalised Minimal RESidual (GMRES) method

The Generalised Minimal Residual (GMRES) method [10][11] is a Krylov subspace method for solving systems of linear equations. It minimises the residual norm in every step. GMRES is an oblique projection method, so $\mathcal{L}_m \neq \mathcal{K}_m$. Instead, we have:

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

Just as FOM, GMRES is based on the Arnoldi method. However, there are some differences. We use equation (4.5) instead of equation (4.4) to build an orthogonal basis for the Krylov subspace. The particular selection of the right and left subspace in GMRES ensures the minimisation of the residual in the Euclidian norm. Assume that $V_j \in \mathbb{R}^{n \times j}$ represents an orthogonal basis for $\mathcal{K}_j(A, r_0)$. The approximate solution is given by $x_j = x_0 + V_j y_j$. We can rewrite the residual in a similar fashion as in the Arnoldi method. This yields:

$$||r_j|| = ||\beta e_1 - \overline{H}_j y_j||. \tag{4.21}$$

We have to select the y_j in order to minimise the residual. We define:

$$y_j = \min_y ||\beta e_1 - \overline{H}_j y||. \tag{4.22}$$

Algorithm 4.5 shows the GMRES method in pseudoformal notation. An 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 least squares problem. In line 16 the approximate solution x_m is calculated. Using equation (4.21) we obtain the following stopping criterion:

$$||\beta e_1 - \overline{H}_j y_j|| < ||b|| \cdot TOL. \tag{4.23}$$

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: $\quad w_j := Av_j$
4: $\quad$ For $i = 1, 2, \ldots, j$ **Do**
5: $\quad\quad h_{ij} = (w_j, v_i)$
6: $\quad\quad w_j := w_j - h_{ij}v_i$
7: $\quad$ **EndDo**
8: $\quad h_{j+1,j} = ||w_j||_2$
9: $\quad$ **If** $h_{j+1,j} = 0$
10: $\quad\quad$ Stop
11: $\quad$ **EndIf**
12: $\quad v_{j+1} = w_j/h_{j+1,j}$
13: $\quad y_j = min_y||\beta e_1 - \overline{H}_j y||$
14: $\quad$ Check stopping criterion
15: **EndDo**
16: $x = x_0 + V_m y_m$

4.2.3 The Conjugate Gradient (CG) method

The Conjugate Gradient (CG) [8][10] method is one of the best-known iterative methods for solving systems of linear equations with a symmetric positive definite matrix A (see definition 2.5). It is an orthogonal projection method onto $\mathcal{K}_m$ and orthogonal to $\mathcal{L}_m$ with the following left and right subspace:

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

$$\mathcal{L}_m(A, v_1) = \mathcal{K}_m(A, v_1).$$

Let x be the exact solution of the system of linear equations $Ax = b$. The idea of CG is to construct a vector $x_j \in \mathcal{K}_j$ in every 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: $||y||_A = \sqrt{y^T Ay}$ [19, p. 66]. In every 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 4.6. An implementation can be found in appendix B.3. In every 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 (4.20) 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

Algorithm 4.6 The Conjugate Gradient method (CG)

1: Compute $r_0 = b - Ax_0$ and $p_0 := r_0$
2: For $j = 0, 1, 2, \ldots$ m **Do**
3: $\quad \alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4: $\quad x_{j+1} := x_j + \alpha_j p_j$
5: $\quad r_{j+1} := r_j - \alpha_j A p_j$
6: $\quad$ Check stopping criterion
7: $\quad \beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
8: $\quad p_{j+1} := r_{j+1} + \beta_j p_j$
9: **EndDo**
10: $x = x_m$

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:

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

4.2.4 The Conjugate Residual (CR) method)

In section 4.2.3 we found that CG was an analogue of FOM for symmetric positive definite (SPD) matrices. Something similar holds for the Conjugate Residual (CR) method [10]. It is an analogue of GMRES for Hermitian SPD matrices. Because of this, the CR method is an oblique projection method onto $\mathcal{K}_m$ and orthogonal to $\mathcal{L}_m$, where

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

Algorithm 4.7 shows in pseudoformal notation the CR method. The implementation of the CR method can be found in appendix B.4.

Algorithm 4.7 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: $\quad \alpha_j := (r_j, Ar_j)/(Ap_j, Ap_j)$
4: $\quad x_{j+1} := x_j + \alpha_j p_j$
5: $\quad r_{j+1} := r_j - \alpha_j A p_j$
6: $\quad$ Check stopping criterion
7: $\quad \beta_j := (r_{j+1}, Ar_{j+1})/(r_j, Ar_j)$
8: $\quad p_{j+1} := r_{j+1} + \beta_j p_j$
9: $\quad Ap_{j+1} = Ar_{j+1} + \beta_j A p_j$
10: **EndDo**
11: $x = x_{m+1}$

The algorithm is similar to the algorithm of CG. In every 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 (4.20).

In the CR method, the residual vectors r_i are A-orthogonal (conjugate): $r_i A r_j = 0$ 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 [10, pp. 203-204].

4.2.5 The Biconjugate Gradient (Bi-CG) method

The Conjugate Gradient algorithm is one of the most widely used algorithms to solve a system of linear equations $Ax = b$. The Biconjugate Gradient method (Bi-CG) [4][10] can be viewed as an extension of CG for general matrices. It is an oblique projection method onto $\mathcal{K}_m$ and orthogonal to $\mathcal{L}_m$ with

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

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.

Algorithm 4.8 Biconjugate 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: $\quad \alpha_j := (r_j, r_j^*)/(p_j, A^T p_j^*)$
5: $\quad x_{j+1} := x_j + \alpha_j p_j$
6: $\quad r_{j+1} := r_j - \alpha_j A p_j$
7: $\quad r_{j+1}^* := r_j^* - \alpha_j A^T p_j^*$
8: $\quad$ Check stopping criterion
9: $\quad \beta_j := (r_{j+1}, r_{j+1}^*)/(r_j, r_j^*)$
10: $\quad p_{j+1} := r_{j+1} + \beta_j p_j$
11: $\quad p_{j+1}^* := r_{j+1}^* + \beta_j p_j^*$
12: **EndDo**
13: $x = x_{m+1}$

A possible algorithm for the Bi-CG method is given in algorithm 4.8 and an implementation is given in appendix B.5. First we define the dual system of $Ax = b$ as the system $A^T x^* = b^*$. Next, we define $r_j^* = b^* - A^T x_j^*$ as the j-th residual of the dual system. r_j^* is often called the 'shadow residual'. In every 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 (4.20) as a stopping criterion, where $||r_j||$ is computed directly. The Bi-CG method can be used to solve the dual system as well. To do this, we need to insert $x^*_{j+1} = x^*_j + \alpha_j p^*_j$ after line 5 of the algorithm.

4.2.6 The Biconjugate Residual (Bi-CR) method

The Bi-CG method is an extension of CG for nonsymmetric matrices. Similarly, the Biconjugate Residual method (Bi-CR) was recently suggested as an extension to CR for nonsymmetric matrices [14]. Bi-CR is an oblique projection method with the following right and left subspace:

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

with $v_1 = r_0/||r_0||$ and $w_1 = r^*_0/||r^*_0||$ as usual. An implementation for Bi-CR can be found in algorithm 4.9

Algorithm 4.9 Biconjugate 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: $\quad \alpha_j := (r_j, A^T r^*_j)/(p_j, A^T(A^T p^*_j))$
4: $\quad x_{j+1} := x_j + \alpha_j p_j$
5: $\quad r_{j+1} := r_j - \alpha_j A p_j$
6: $\quad r^*_{j+1} := r^*_j - \alpha_j A^T p^*_j$
7: $\quad$ Check stopping criterion
8: $\quad \beta_j := (r_{j+1}, A^T r^*_{j+1})/(r_j, A^T r^*_j)$
9: $\quad p_{j+1} := r_{j+1} + \beta_j p_j$
10: $\quad p^*_{j+1} := r^*_{j+1} + \beta_j p_j *$
11: **EndDo**
12: $x = x_m$

When we look closely at the algorithm, we find that it is 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 3.5) 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 we have:

$$r_j^* = s_{j+1}(A^T)A^T r_0^*. \tag{4.24}$$

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:

$$p_j^* = t_{j+1}(A^T)A^T p_0^*. \tag{4.25}$$

From equation (4.24) and (4.25), 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:

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

Chapter 5

IDR(s): Induced Dimension Reduction(s)

If we are interested in solving a system of linear equations $Ax = b$ with A a an SPD matrix, the Conjugate Gradient method (see section 4.2.3) is often preferred. It combines optimal minimisation of the residual with short recurrences.

Unfortunately it is not possible to find a Krylov subspace method for general nonsymmetric matrices that combines these properties [3]. When the matrix A is not symmetric, we can follow two approaches. In the first approach, research focused on methods in which the requirement for short recurrences was removed. GMRES (see section 4.2.2) is the most popular member of this family of methods. In the second approach, research focused on short recurrence methods without the optimality condition. The archetype of this method is the Bi-CG method (see section 4.2.5). However Bi-CG requires twice the work as CG. Other methods, such as CGS [15] and Bi-CGSTAB [18] have been developed in order to overcome this problem, but all these methods are based on Bi-CG.

This is where the Induced Dimension Reduction (IDR) method comes in. It was first proposed by Peter Sonneveld in 1980 [20]. IDR is based on reasonably short recurrences 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 has been completely overshadowed by CGS and Bi-CGSTAB, but in recent years there has been renewed interest in IDR. One of the new family of methods that was developed in 2008 is the IDR(s) method (see [16] and [17]).

5.1 Derivation of the IDR(s) algorithm

The IDR(s) method is based on the IDR theorem, which was originally published in [20].

Theorem 5.1 (IDR Theorem)**.**
Let A be any matrix in $\mathbb{R}^{n\times n}$, let v_1 be any nonzero vector in $\mathbb{R}^n$ 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

$$\mathcal{G}_j = (I - \omega_j A)\,(\mathcal{G}_{j-1} \cap \mathcal{S})\,, \tag{5.1}$$

where the ω_j's are nonzero scalars.

Then 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 [16, p. 1037]

The main idea of the IDR(s) method is to generate residuals r_m that are forced to be in the subspaces $\mathcal{G}_j$, where j is nondecreasing for increasing m. 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 [16], the residuals of a general Krylov-type solver can be described by

$$r_{m+1} = r_m - \alpha A v_m - \sum_{k=1}^{K} \gamma_k \Delta r_{m-k}, \tag{5.2}$$

where $\gamma_k, \alpha \in \mathbb{R}$ and $v_m \in \mathcal{K}_m(A, r_0) \setminus \mathcal{K}_{m-1}(A, r_0)$. The integer K is the depth of the recursion. Recall that we want the residual r_{m+1} to be in the Sonneveld spaces $\mathcal{G}_{j+1}$. This is the case if

$$r_{m+1} = (I - \omega_{m+1} A)\, v_m \qquad \text{with } v_m \in \mathcal{G}_j \cap \mathcal{S}, \tag{5.3}$$

which can be seen directly from the definition of the Sonneveld spaces. Now define the $(K \times 1)$ vector $c = [\gamma_1\ \gamma_2\ \ldots\ \gamma_K]^T$ and the $(n \times s)$ matrix $\Delta R_m = [\Delta r_{m-1}\ \Delta r_{m-2}\ \ldots\ \Delta r_{m-s}]$. If we choose v_m to be

$$v_m = r_m - \sum_{k=1}^{K} \gamma_k \Delta r_{m-k} = r_m - \Delta R_m c, \tag{5.4}$$

then r_{m+1} satisfies equation (5.2) (with $\alpha = \omega_{j+1}$).

Now define an $(n \times s)$ matrix $P = [p_1\ p_2\ \ldots\ p_s]$ with $p_{ij} \in \mathbb{R}$. Without loss of generality, we assume that the subspace $\mathcal{S}$ is the left nullspace of P. Letting $v_m \in \mathcal{S}$, we obtain:

$$P^T v_m = 0. \tag{5.5}$$

Substituting equation (5.4) in (5.5) yields:

$$\left(P^T \Delta R_m\right) c = P^T r_m. \tag{5.6}$$

Since we know P, ΔR_m and r_m, we can solve this system of linear equations. Having K unknowns and s equations, this system is in general uniquely solvable for c if $K = s$. In Figure 5.1 we illustrate this.

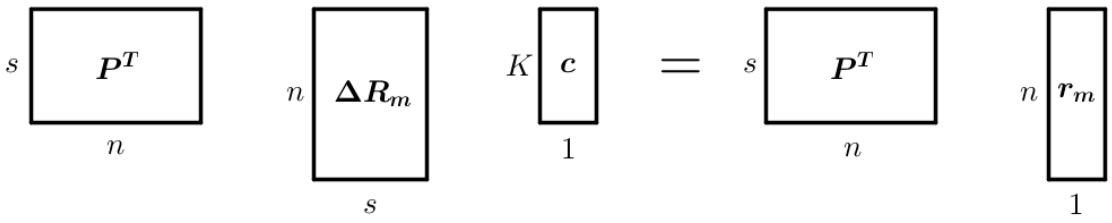

Figure 5.1: Solving $\left(P^T \Delta R_m\right) c = P^T r_m$

With the vector c we can compute v_m and hence r_{m+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 know $v_{m+1} \in \mathcal{G}_j \cap \mathcal{S}$, we can conclude from equation (5.3) 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 $\mathcal{G}_{j+2}$.

Of course, we also need to find an expression for the solution vector. We can easily find one using equation (5.2) (with $\alpha_m = \omega_{j+1}$):

$$r_{m+1} \;=\; r_m - \omega_{j+1} A v_m - \sum_{k=1}^{K} \gamma_k \Delta r_{m-k} \;=\; r_m - \omega_{j+1} A v_m - \Delta R_{m-k} c. \tag{5.7}$$

Using $r_m = b - A x_m$, cancelling the b's on both sides and multiplying with A^{-1} yields:

$$x_{m+1} \;=\; x_m + \omega_{j+1} v_m - \sum_{k=1}^{K} \gamma_k \Delta x_{m-k} \;=\; x_m + \omega_{j+1} v_m - \Delta X_{m-k} c. \tag{5.8}$$

Equation (5.7) and (5.8) form the basis of IDR(s). An algorithm for IDR(s) can be seen in algorithm 5.1.

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 have to find ω_{j+1}. We can use an approach that minimises the residual (see e.g. [17]). We obtain

$$\omega_{j+1} = \frac{v^T A v}{v^T A^T A v}.$$

For the calculation of the subsequent residuals in $\mathcal{G}_{j+1}$, the same value for ω_{j+1} must be used. This is a drawback, since it is not guaranteed that this value of ω_{j+1} also minimises the subsequent residuals in $\mathcal{G}_{j+1}$. Other selections for ω_j are proposed in [12] and [17].

Algorithm 5.1 IDR(s)

Require $A \in \mathbb{R}^{N\times N}$; $x_0, b \in \mathcal{R}^N$; $P \in \mathbb{R}^{N\times s}$; $TOL \in (0,1)$; $MAXIT > 0$
Ensure x_m such that $||b - Ax_m|| < TOL$
    $\{Initialisation\}$
    Calculate $r_0 = b - Ax_0$;

    $\{Apply\ s\ minimum\ norm\ steps to\ build\ enough\ vectors\ in\ \mathcal{G}_0\}$
    For $m = 0$ to $s - 1$ **Do**
        $v = Ar_m$;
        $\omega = \left(v^T r_m\right) / \left(v^T v\right)$;
        $\Delta x_m = \omega r_m$;
        $\Delta r_m = -\omega v$;
        $r_{m+1} = r_m + \Delta r_m$;
        $x_{m+1} = x_m + \Delta x_m$;
    EndFor
    $\Delta R_{m+1} = (\Delta r_m \ldots \Delta r_0)$;
    $\Delta X_{m+1} = (\Delta x_m \ldots \Delta x_0)$;

    $\{Building\ \mathcal{G}_j\ spaces\ for\ j = 1, 2, 3, \ldots\}$
    $m = s$
    $\{Loop\ over\ \mathcal{G}_j\ spaces\}$
    While $||r_m|| > TOL$ **and** $m < MAXIT$ **Do**
        $\{Loop\ inside\ \mathcal{G}_j\ spaces\}$
        For $k = 0$ to s **Do**
            Solve c from $P^T \Delta R_m c = P^T r_m$;
            $v = r_m - \Delta R_m c$;
            If $k = 0$ **then**
                $\{Entering\ \mathcal{G}_{j+1}\}$
                $t = Av$;
                $\omega = \left(t^T v\right) / \left(t^T t\right)$;
                $\Delta x_m = -\Delta X_m c + \omega v$;
                $\Delta r_m = -\Delta R_m c - \omega t$;
            else
                $\{Subsequent\ vectors\ in\ \mathcal{G}_{j+1}\}$
                $\Delta x_m = -\Delta X_m c + \omega v$;
                $\Delta r_m = -A\Delta x_m$;
            End if
            $r_{m+1} = r_m + \Delta r_m$;
            $x_{m+1} = x_m + \Delta x_m$;
            $m = m + 1$;
            $\Delta R_m = (\Delta r_{m-1}, \ldots, \Delta r_{m-s})$;
            $\Delta X_m = (\Delta x_{m-1}, \ldots, \Delta x_{m-s})$;
        End for
    End while
    $x = x_m$

5.2 Performance of the IDR(s) method

The IDR theorem predicts that dimension reduction will take place, but it does not provide information about the speed of convergence. The extended IDR theorem gives information about the rate of convergence.

Theorem 5.2 (Extended IDR theorem)**.**
Let A be any matrix in $\mathbb{R}^{n\times n}$, let $p_1, p_2, \ldots, p_s \in \mathbb{R}^n$ 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

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

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

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

Proof. See [16]

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 [16]. 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 (see [16, p. 1041–1042]):

Corollary 5.3.
In the generic case IDR(s) requires at most $n + n/s$ matrix-vector multiplications to compute the exact solution.

As a result, the IDR(s) method is a finite method, meaning that it will find a solution in a finite number of iterations.

5.3 Numerical experiments with IDR(s)

In this section we will give four examples that compare the convergence behaviour of Bi-CG, full GMRES and IDR(s) for $s = 1, 2, 4$ and 8. We take $b = A \cdot \text{ones(n, 1)}$, $TOL = 10^{-8}$ and a random starting vector v_1. We use the build-in routines in Matlab for GMRES and Bi-CG and use the `idrs.m` algorithm from the website of Martin van Gijzen[1].

[1] http://ta.twi.tudelft.nl/nw/users/gijzen/IDR.html

5.3.1 Example 5.3.1 - the convection-diffusion equation

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

$$L(u) = -\Delta u + \beta(u_x + u_y + u_z) = -\Delta u + \beta \nabla u, \tag{5.9}$$

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

In Figure 5.2 (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 5.3), 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.

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 behaviour is caused by the larger convection term, which makes the problem asymmetrical [12, p.11]. In Table 5.1 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, though it needs much more time to find a solution.

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 5.1: Example 5.3.1 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 5.2: Example 5.3.1 with $\beta = 200$

5.3.2 Example 5.3.2 - the `Sherman4` matrix

From the Matrix Market[2] we consider the `Sherman4` matrix[3]. This nonsymmetric, real matrix of size 1104×1104 is used in the simulation of oil reservoirs.

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 take 1000 as the maximum number of iterations. In Figure 5.4 we can see the convergence behaviour of GMRES, Bi-CG and IDR(s).

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

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

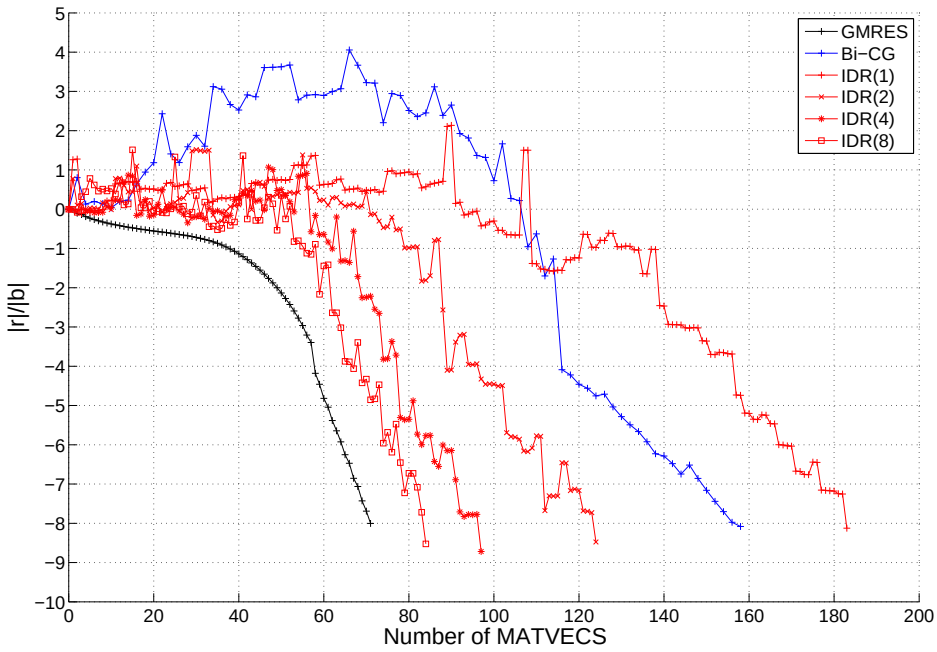

Figure 5.2: Convergence behaviour of the convection diffusion matrix with $\beta = 100$

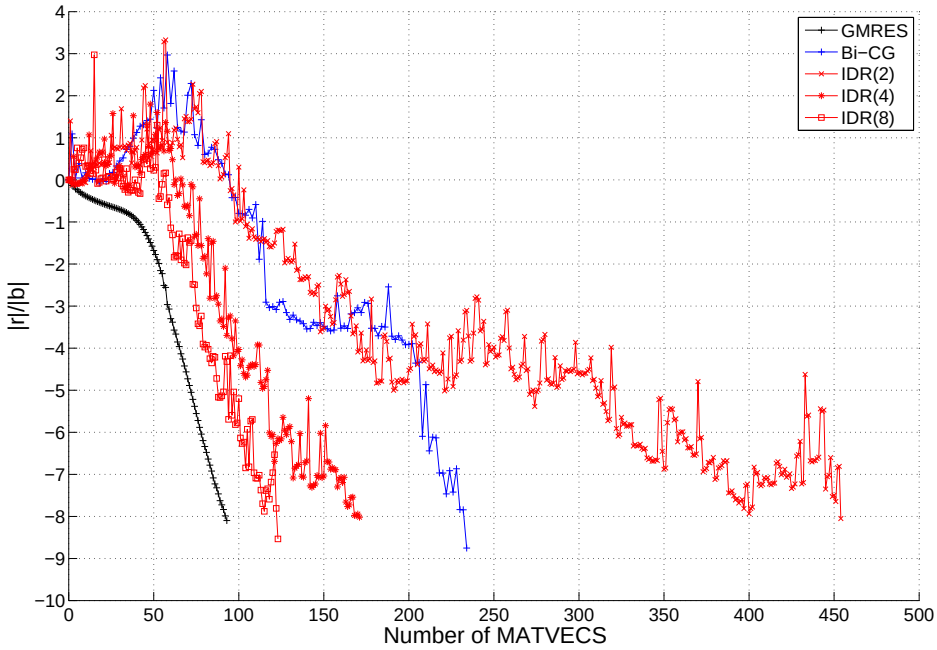

Figure 5.3: Convergence behaviour of the convection-diffusion matrix with $\beta = 200$

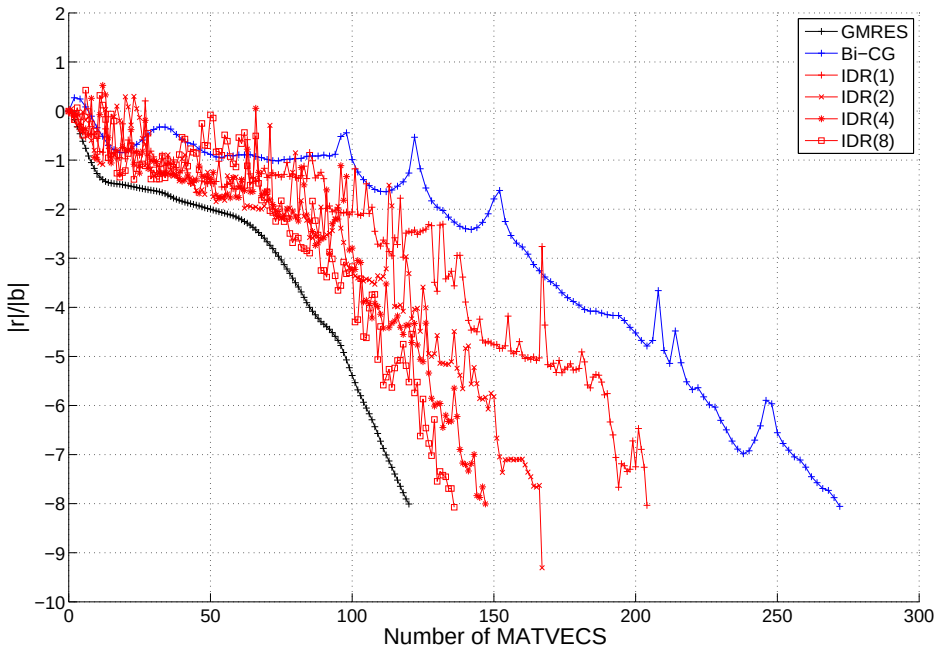

Figure 5.4: Convergence behaviour of the `Sherman4` matrix

In this example IDR(s) with $s = 1, 2, 4, 8$ outperforms Bi-CG. In Table 5.3 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 four times as fast. For this example, the IDR(s) apparently works 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 5.3: Convergence behaviour of the `Sherman4` matrix

5.3.3 Example 5.3.3 - the `add20` matrix

The `add20` matrix[4] is another matrix from the Matrix Market. It is a real nonsymmetric 2395×2395 matrix that is used in electronic circuit design.

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 take 1000 as the maximum number of MATVECS. In Figure 5.5 we see the convergence behaviour for GMRES, Bi-CG and IDR(s) and in Figure 5.6 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.

In Table 5.4 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).

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 5.4: Convergence behaviour of the `add20` matrix

5.3.4 Example 5.3.4 - the `jpwh_991` matrix

The `jpwh_991` matrix[5] is another matrix from the Matrix Market. It is a real nonsymmetric 991×991 matrix that is used in circuit physics.

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 take 1000 as the maximum number of iterations.

In Figure 5.7 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. 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.

[4] `http://math.nist.gov/MatrixMarket/data/misc/hamm/add20.html`
[5] `http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/cirphys/jpwh_991.html`

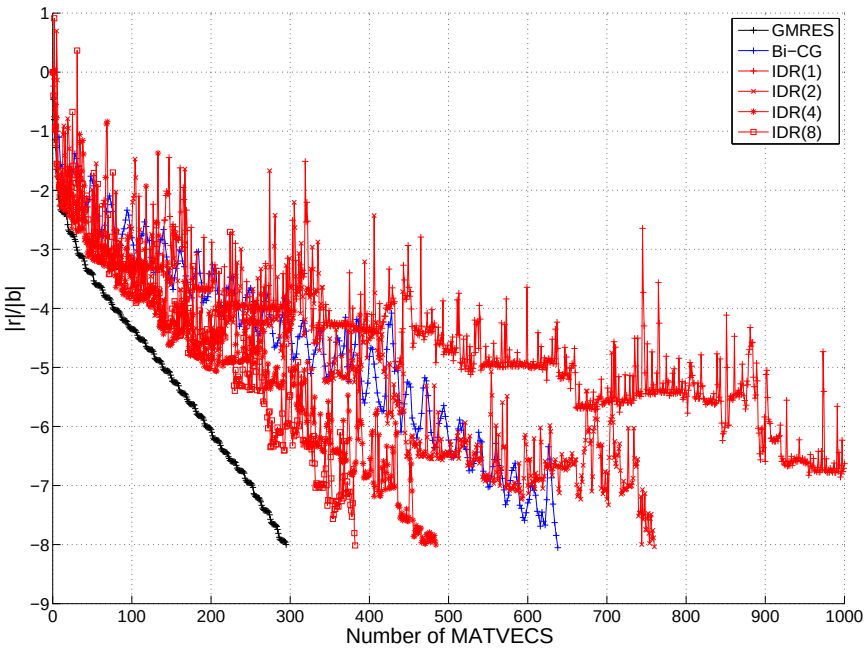

Figure 5.5: Convergence behaviour of the `add20` matrix

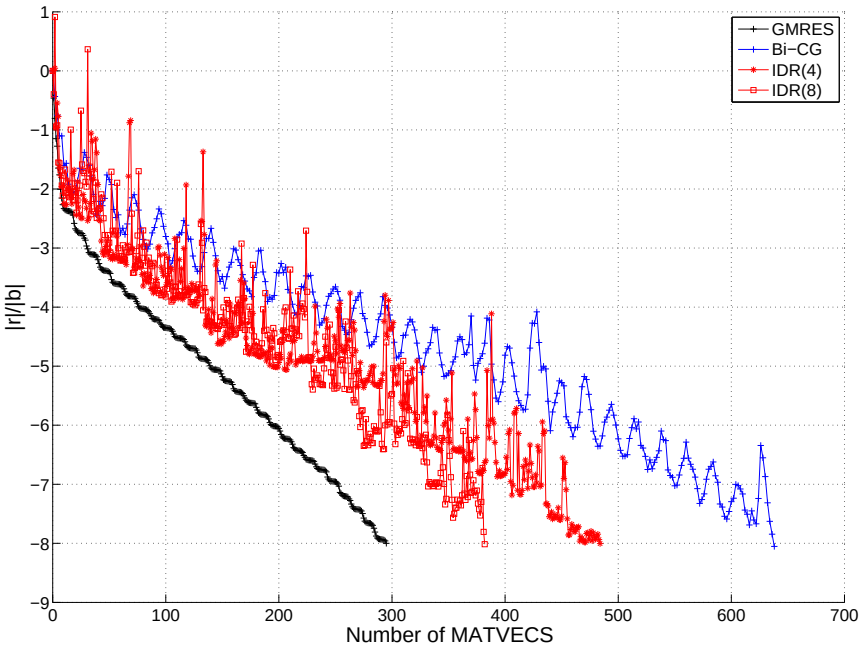

Figure 5.6: Zoomed in convergence behaviour of the `add20` matrix

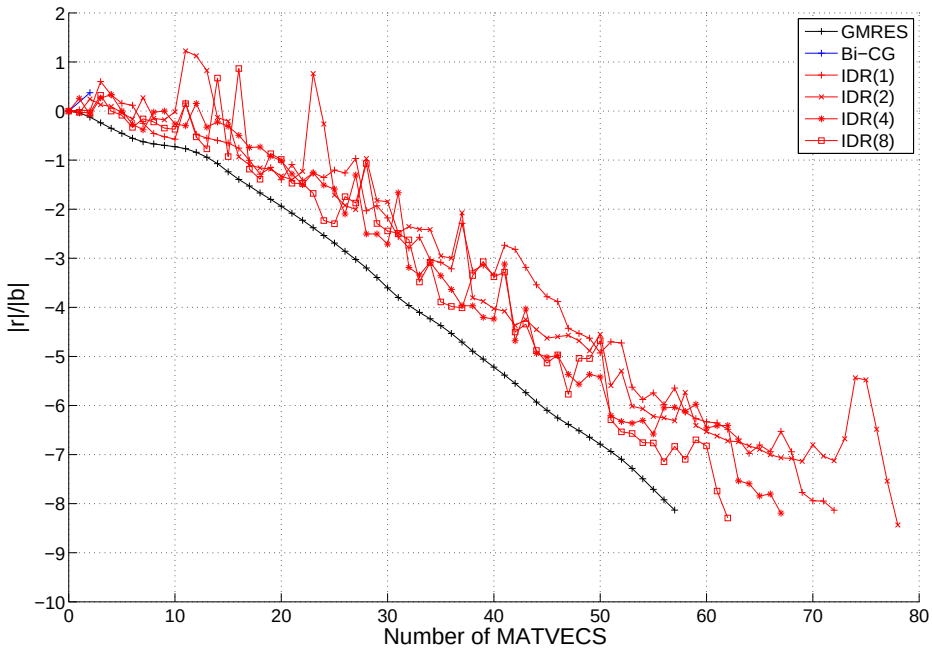

Figure 5.7: Convergence behaviour of the `jpwh_991` matrix

In Table 5.5 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.

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 5.5: Convergence behaviour of the `jpwh_991` matrix

Chapter 6

PIDR(s): Projected Induced Dimension Reduction(s)

Recall that in a projection method, we can find the approximate solution x_m to a system of linear equations $Ax = b$ with $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$ in the following manner:

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

where $\mathcal{K}_m$ is defined as:

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

Section 3.5 said that different choices of $\mathcal{L}_m$ lead to different projection methods. It turns out that the IDR(s) method can also be seen in the framework of projection methods when we consider a special choice for $\mathcal{L}_m$ [12]. We will also explicitly give a formula for the right subspace $\mathcal{K}_m$. Hereafter, we will refer to IDR(s) as a projection method as PIDR(s), which stands for Projected IDR(s).

6.1 Analysis of the right subspace

Simoncini and Szyld [12] claim that the right subspace is equal to a Krylov subspace $\mathcal{K}_m(A, r_0)$ generated by A and r_0. However, they do not explicitly give a definition for the right subspace. In their definition of the left subspace we see a factor $(\Omega_j(A))^{-T}$. However, in a projection process we avoid these inverses, since they can make a program unstable. This seems to support the idea that we should have a right subspace with a factor $\Omega_j(A)$ in its definition in order to avoid the explicit computation of $(\Omega_j(A))^{-T}$. We will now derive an expression for the right subspace.

Section 3.5 tells that we can always write a residual as a polynomial in A multiplied with r_0:

$$r_m = \Phi_m(A)r_0, \tag{6.1}$$

where $\Phi_m(A)$ is a polynomial in A of degree m. Now suppose that $r_m \in \mathcal{G}_j$ for some $j > 0$.

From [16] we have:

$$r_m = (I - \omega_j A)\, r' \qquad \text{with } r' \in \mathcal{G}_{j-1} \cap \mathcal{S}$$

Since $r' \in \mathcal{G}_{j-1}$, we can write $r' = (I - \omega_j A))\, r''$ with $r'' \in \mathcal{G}_{j-2} \cap S$. Repeating this yields:

$$r_m = \Omega_j(A) r''' \qquad \text{with } r''' \in \mathcal{G}_0 \cap \mathcal{S}, \tag{6.2}$$

where we define $\Omega_j(A)$ as a matrix of degree j:

$$\Omega_j(A) = \begin{cases} I & \text{if } j = 0; \\ \prod_{i=1}^{j} (I - \omega_i A) & \text{if } j \geq 1. \end{cases} \tag{6.3}$$

Here, I denotes the $(n \times n)$ identity matrix and $\omega_i \in \mathbb{R}$. By definition, $\Omega_j(A) \in \mathbb{R}^{n \times n}$. Equating (6.1) and (6.2) yields:

$$\Omega_j(A) r''' \;=\; \Phi_m(A) r_0 \qquad \Longleftrightarrow \qquad r''' \;=\; \Psi_{m-j}(A) r_0, \tag{6.4}$$

where $\Psi_{m-j}(A) = (\Omega_j(A))^{-1}\, \Phi_m(A)$. Note that $(\Omega_j(A))$ is of degree j and $\Phi_m(A)$ is of degree m. Hence, $\Psi(A)$ is of degree $m - j$. When we substitute equation (6.4) in (6.2) we obtain:

$$r_m = \Omega_j(A) \Psi_{m-j}(A) r_0.$$

If we define $r_{m-j} = \Psi_{m-j}(A) r_0$, then we know that $r_{m-j} = span\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-j} r_0\}$, which is the same as saying that $r_{m-j} \in \mathcal{K}_{m-j+1}$. We know have:

$$r_m = \Omega_j(A) \mathcal{K}_{m-j+1}.$$

Using section 3.5, we obtain:

$$x_m = \Omega_j(A) \mathcal{K}_{m-j}.$$

We now claim that the subspace in equation (6.5) is the correct search subspace of PIDR(s):

$$\widetilde{\mathcal{K}}_m(A, \widetilde{r}_0) \;=\; \Omega_j(A)\; \mathcal{K}_{m-j}(A, r_0), \tag{6.5}$$

with $\Omega_j(A)$ as in equation (6.3). Here, $\mathcal{K}_{m-j}(A, r_0)$ is a regular Krylov subspace (defined in definition (3.11)) of dimension $m - j$.

There are a few important points to be noted. First, note that we could write $\widetilde{\mathcal{K}}_m(A, \widetilde{r}_0) = \mathcal{K}_{m-j}(A,\ \Omega_j(A) r_0)$ [16, p. 1046] and hence $\widetilde{r}_0 = \Omega_j(A) r_0$ by definition. Furthermore, note that $\mathcal{K}_{m-j}$ is dependent on $\Omega_j(A)$ instead of $\Omega_m(A)$. This is no coincidence, as we will show in section 6.3.

6.2 Analysis of the left subspace

Let $\sigma(A)$ denote the spectrum of A (see definition 2.8) and define the block Krylov subspace $\mathcal{K}_j(A^T, P)$ (see definition 2.10). Here, $P = [p_1, p_2, \ldots, p_s]$ is a matrix in $\mathbb{R}^{n\times s}$ (see [16, p. 1044] for a discussion on how to choose P) and A^T denotes the conjugate transpose of A. We could also write $\mathcal{K}_j(A^T, P)$ in the following way:

$$\mathcal{K}_j(A^T, P) = \bigcup_{i=1}^{s} \mathcal{K}_j(A^T, p_i)$$

According to Simoncini and Szyld [12], the left subspace for the IDR(s) algorithm is equal to:

$$\widetilde{\mathcal{L}}_j(A^T, \widetilde{P}) \quad = \quad (\Omega_j(A))^{-T}\, \mathcal{K}_j(A^T, P). \tag{6.6}$$

The subspace $\widetilde{\mathcal{L}}_j$ is a *rational block Krylov subspace* (see definitions 2.9 and 2.10), in which we use $(\Omega_j(A))^{-T}$: the inverse of the matrix $\Omega_j(A)^T$. A necessary condition for the invertibility of this matrix is that $\frac{1}{\omega_j} \notin \sigma(A)$. This can be seen by rewriting equation 6.3 (for $j \geq 1$) as

$$\Omega_j(A) = c \cdot \left(A - \frac{1}{\omega_j} I\right) \cdot \ldots \cdot \left(A - \frac{1}{\omega_2} I\right) \left(A - \frac{1}{\omega_1} I\right),$$

with $c = (-1)^j \cdot \omega_j \cdot \ldots \cdot \omega_2 \cdot \omega_1$. If $\frac{1}{w_k} \in \sigma(A)$ for some $1 < k \leq j$, we have that $\left(A - \frac{1}{\omega_k} I\right) = 0$ and hence $\det \Omega_j(A) = 0$. Therefore we should have $\frac{1}{\omega_j} \notin \sigma(A)$.

We can prove that $\widetilde{\mathcal{L}}_j$ gives rise to a projection method. Recall that the IDR(s) method generates residuals that are forced to be in subspaces $\mathcal{G}_j$ of decreasing dimension, where $\mathcal{G}_j$ is defined in equation (5.1).We can write $\mathcal{G}_j$ in the following way [6, p. 24] [13, p. 1104]:

$$\mathcal{G}_j = \Omega_j(A)\ \mathcal{K}_j(A^T, P)^{\perp}. \tag{6.7}$$

When $s + 1$ residuals have been computed in $\mathcal{G}_j$, the next residual will be in $\mathcal{G}_{j+1}$. This can be written as

$$r_m \in \mathcal{G}_j, \quad \text{with } j = \lfloor m/(s+1) \rfloor,$$

where $\lfloor m/(s+1) \rfloor$ denotes the largest integer not greater than $m/(s+1)$. In a projection method, these residuals should be orthogonal to the left subspace. Theorem 6.1 (see [12]) states that $\widetilde{\mathcal{L}}_j$ is indeed the left subspace of a projection method.

Theorem 6.1.
$r_m \perp \widetilde{\mathcal{L}}_j$ and hence $\widetilde{\mathcal{L}}_j$ is the left subspace of the IDR(s) method.

Proof. Let $r_m \in \mathcal{G}_j$. This implies that $r_m \perp \mathcal{G}_j^{\perp}$. We show that $\mathcal{G}_j^{\perp} = \widetilde{\mathcal{L}}_j$. Let $B = \Omega_j(A)$ and $\mathcal{L}_j = \mathcal{K}_j(A^T, P)$. Using (6.7), we find $\mathcal{G}_j = B\mathcal{L}_j^{\perp} = \{Bw \mid w \in \mathcal{L}_j^{\perp}\}$. We now have:

$$\begin{aligned}
\mathcal{G}_j^{\perp} \overset{(6.7)}{=} \left(B\mathcal{L}_j^{\perp}\right)^{\perp} &= \{v \mid v^T Bw = 0,\ w \in \mathcal{L}_j^{\perp}\} \\
&= \{v \mid \left(B^T v\right)^T w = 0,\ w \in \mathcal{L}_j^{\perp}\} \\
&= \{B^{-T} y \mid y^T w = 0,\ w \in \mathcal{L}_j^{\perp}\} \qquad \left(\text{using } y = B^T v\right) \\
&= B^{-T}\{y \mid y^T w = 0,\ w \in \mathcal{L}_j^{\perp}\} = B^{-T}\mathcal{W} \overset{(6.6)}{=} \widetilde{\mathcal{L}}_j
\end{aligned}$$

Since $\mathcal{G}_j^{\perp} = \widetilde{\mathcal{L}}_j$, we have that $r_m \perp \widetilde{\mathcal{L}}_j$ if $r_m \in \mathcal{G}_j$.

■

6.3 Definition of the approximate solution

It is now time to show how IDR(s) fits into the framework of projection methods. For this we use algorithm 3.1, which shows the general framework for a projection method. In this algorithm, we first have to select the right subspace $\mathcal{K}_m$ and the left subspace $\mathcal{L}_m$. Sections 6.1 and 6.2 tell that the right and left subspace can be written in the following way:

$$\widetilde{\mathcal{K}}_m(A, \widetilde{r}_0) = \Omega_j(A)\ \mathcal{K}_{m-j}(A, r_0) = \mathcal{K}_{m-j}(A,\ \Omega_j(A) r_0) \tag{6.8}$$

$$\widetilde{\mathcal{L}}_j(A^T, \widetilde{P}) = (\Omega_j(A))^{-T}\, \mathcal{K}_j(A^T, P) = \mathcal{K}_j(A^T,\ (\Omega_j(A))^{-T}\, P) \tag{6.9}$$

Next, the algorithm builds a basis for these two subspaces. Denote $\widetilde{V}_m$, $\widetilde{W}_j$, V_{m-j} and W_j as the bases for the right and left subspace, the Krylov subspace $\mathcal{K}_{m-j}(A, r_0)$ and the block Krylov subspace $\mathcal{K}_j(A^T, P)$ respectively. We obtain:

$$\widetilde{V}_m = \Omega_j(A)\ V_{m-j}; \tag{6.10}$$

$$\widetilde{W}_j = (\Omega_j(A))^{-T}\, W_j, \tag{6.11}$$

In line 4 of algorithm 3.1, we compute the coefficient vector y_m, which contains the coefficients for the basis vectors $\left[\widetilde{v}_1, \ldots \widetilde{v}_m\right]$. Using the orthogonality condition and (3.7) yields:

$$\begin{aligned}
\widetilde{W}_j^T r_m = 0 &\Leftrightarrow \widetilde{W}_j^T r_0 - \widetilde{W}_j^T A \widetilde{V}_m y_m = 0 \\
&\Leftrightarrow \widetilde{W}_j^T \widetilde{r}_0 = \widetilde{W}_j^T A \widetilde{V}_m y_m
\end{aligned}$$

Solving this expression for y_m, we obtain:

$$y_m = \left(\widetilde{W}_j^T A \widetilde{V}_m \right)^{-1} \widetilde{W}_j^T r_0. \tag{6.12}$$

We can simplify equation (6.12) by using the following proposition:

Proposition 6.2.
The matrix polynomial $\Omega_j(A)$ is commutative with any matrix $A \in \mathbb{R}^{n\times n}$ for $j \geq 0$.

Proof. It is clear that $\Omega_j(A)$ is commutative with A if $j = 0$, since any matrix is commutative with the identity matrix. For $j \geq 1$ we use induction. For $j = 1$ we have:

$$A\Omega_1(A) = A\left(I - \omega_1 A\right) = AI - A\omega_1 A = IA - \omega_1 A^2 = \left(I - \omega_1 A\right) A = \Omega_1(A)A$$

Now suppose that $A\Omega_j(A) = \Omega_j(A)A$ for some $j > 1$. For $j + 1$ we find:

$$\begin{aligned} A\Omega_{j+1}(A) &= A\Omega_j(A)\left(I - \omega_{j+1}A\right) \\ &\overset{I.H.}{=} \Omega_j(A)A\left(I - \omega_{j+1}A\right) \\ &= \Omega_j(A)\left(I - \omega_{j+1}A\right)A \quad = \quad \Omega_{j+1}(A)A. \end{aligned}$$

■

Substituting (6.10) and (6.11) into equation (6.12) and using proposition 6.2 yields:

$$y_m = \left(W_j^T A V_{m-j}\right)^{-1} \widetilde{W}_j^T r_0. \tag{6.13}$$

Note that we have to compute the inverse of the matrix $W_j^T A V_{m-j}$. This can only be done if this matrix is square. In other words, the number of vectors in W_j should be equal to the number of vectors in V_{m-j}. This is not true for general m. Recall that $j = \lfloor m/(s+1) \rfloor$. Now suppose that we are in an iteration that is a multiple of $s + 1$. Writing $m = k(s + 1)$ with $k = 1, 2, \ldots$, we have $j = k$. Then it is clear that both bases have the same dimension, since:

$$\begin{aligned} \dim\left(V_{m-j}\right) &= n \times (m - j) \quad = \quad n \times (k(s+1) - k) \quad = \quad n \times ks \quad = \quad n \times js \\ \dim\left(W_j\right) &= n \times js \end{aligned}$$

Here we have used that W_j is a block basis with blocks of size s. From this dimension analysis it is clear that we can only compute a unique solution in iterations that are multiples of $s+1$. Since the approximate solution x_m can be written as $x_0 + \widetilde{V}_m y_m$, we can use the coefficient vector in equation 6.12 to compute x_m in line 5 of algorithm 3.1:

$$x_m = x_0 + \widetilde{V}_m \left(W_j^T A V_m\right)^{-1} \widetilde{W}_j^T r_0. \tag{6.14}$$

In the sixth line of algorithm 3.1, we use the norm of the residual to check if the stopping criterion is satisfied.

6.4 The PIDR(s) algorithm

Simoncini and Szyld describe that PIDR(s) tries to find at the m-th iteration an approximate solution $x_m \in x_0 + \widetilde{\mathcal{K}}_m(A, \widetilde{r_0})$ such that $r_m \perp \widetilde{\mathcal{L}}_j$. This only holds if $m > s$, since $\widetilde{\mathcal{L}}_j$ is not defined for $j = 0$. Hence, in order for the method to work, the algorithm needs to compute the first s approximate solutions and residuals using any other projection method. An algorithm of PIDR(s) is given in Algorithm 6.1.

Algorithm 6.1 PIDR(s): Projected IDR(s) for systems of linear equations

1: Do s steps of a projection method to compute s approximations and residuals
2: Select the Krylov subspaces $\mathcal{K}_{m-j}(A, r_0)$ with basis V_{m-j}
3: and the block Krylov subspace $\mathcal{L}_j(A^T, P)$ with basis W_j.
4: Until convergence; **Do**
5: Extend V_{m-j} with s vectors
6: Extend W_j with s vectors
7: Extend $\Omega_j(A)$ to $\Omega_{j+1}(A)$
8: Build basis $\widetilde{V}_m = \Omega_j(A) V_{m-j}$
9: Build basis $\widetilde{W}_m = (\Omega_j(A))^{-1} W_j$
10: $y_m := \left(W_j^T A V_{m-j} \right)^{-1} \widetilde{W}_j^T r_0$
11: $x_m := x_0 + \widetilde{V}_m y_m$
12: Check stopping criterion
13: **EndDo**

First we compute s basis vectors and we define the two bases V_{m-j} and W_m. Next, we look at a cycle of $(s + 1)$ iterations. In s of these iterations, we update V_{m-j}, which means that we compute s extra basis vectors. In the first iteration of each cycle, iterations of the form $j = k(s + 1)$ with $k = 1, 2, \ldots$, we update the polynomial and the bases $\widetilde{V}_m$, W_j and $\widetilde{W}_j$. Since W is a block basis, the number of extra vectors computed in each cycle is equal to s. Hence, the number of vectors in all of the bases V_{m-j}, $\widetilde{V}_m$, W_j and $\widetilde{W}_j$ after one cycle is equal to $j \times s$. Therefore we are able to compute the inverse of $W^T AV$. Using this inverse, we can compute the coefficient vector in line 9 and a new approximate solution x in line 10. It is clear that we can only compute a solution every $(s + 1)$st iteration. In line 11 we compute the residual, which is used to check if the stopping criterion is satisfied.

Two different implementation of the PIDR(s) algorithm can be found in appendix C.2 and appendix C.3. In `pidrs.m` we only compute x_m in iterations that are multiples of $s + 1$. As a result we update V_{m-j} with s vectors at a time. In `pidrs_eachiter.m` we update V_{m-j} in each iteration. Both algorithms are mathematically equivalent. Note that this algorithm should be seen from a theoretical point of view. Just as the Arnoldi method is not a one-to-one copy of algorithm 3.1, neither is the algorithm of PIDR(s). In an efficient implementation,

we would never try to compute the inverse of the matrix $W_m^T A V_{m-j}$. Instead we might be able to use a Hessenberg relation similar to equation 4.2 and 4.3 (for more information on we refer to [2] and [6]). Furthermore, we would also try to prevent the use of the matrix $(\Omega_j(A))^{-T}$ and find an alternate expression for the residual instead of calculating it directly.

6.5 PIDR(s) as an eigenvalue method

Recall that in an eigenvalue problem for projection methods we want to find in the m^{th} iteration for $i = 1, \ldots, m$ the eigenvectors $u_m^{(i)} \neq 0$ in $\mathbb{C}^n$ and the corresponding eigenvalues $\lambda_m^{(i)} \in \mathbb{C}$ of a matrix $A \in \mathbb{R}^{n \times n}$ such that $A u_m^{(i)} = \lambda_m^{(i)} u_m^{(i)}$. We define the m^{th} residual for the i-th eigenvector and corresponding eigenvale as $r_m^{(i)} = A u_m^{(i)} - \lambda_m^{(i)} u_m^{(i)}$. In a projection method for eigenvalue problems we want to find an eigenvector in the right subspace and an eigenvalue such that the left subspace is orthogonal to the residual:

$$\text{Find} \quad \lambda_m^{(i)} \in \mathbb{C} \text{ and } u_m^{(i)} \in \widetilde{\mathcal{K}}_m \qquad \text{such that} \qquad r_m^{(i)} \perp \widetilde{\mathcal{W}}_j. \tag{6.15}$$

Proceeding as in the case of IDR(s) for solving systems of linear equations, we select $\widetilde{V}_m$ be a basis for $\widetilde{\mathcal{K}}_m(A, \widetilde{r}_0)$ and let $\widetilde{W}_m$ be a basis for $\widetilde{\mathcal{W}}_m(A^T, \widetilde{P})$, where $\widetilde{V}_m$ and $\widetilde{W}_m$ are defined as in equations (6.10) and (6.11). Since $u_m^{(i)} \in \widetilde{\mathcal{K}}_m(A, \widetilde{r}_0)$, we can write $u_m^{(i)} = \widetilde{V}_m y_m^{(i)} = \Omega_j(A) V_{m-j} y_m^{(i)}$. Let the $\theta_m^{(i)}$ be the Ritz values of the matrix $\widetilde{W}_j^T A$ corresponding to the eigenvector $\widetilde{W}_j^T u_m^{(i)}$. For iterations that are multiples of $s+1$ we have:

$$\begin{aligned} \widetilde{W}_j^T r_m^{(i)} = 0 \quad &\Leftrightarrow \quad \widetilde{W}_j^T A u_m^{(i)} - \theta_m^{(i)} \widetilde{W}_j^T u_m^{(i)} = 0 \\ &\Leftrightarrow \quad \widetilde{W}_j^T A \widetilde{V}_m y_m^{(i)} = \theta_m^{(i)} \widetilde{W}_j^T \widetilde{V}_m y_m^{(i)} \end{aligned}$$

If we denote $Q_m = \widetilde{W}_j^T A \widetilde{V}_m$ and $C_m = \widetilde{W}_j^T \widetilde{V}_m$, then we have the following relation

$$Q_m y_m^{(i)} = \theta_m^{(i)} C_m y_m^{(i)}. \tag{6.16}$$

Equation (6.16) is an example of a *generalised eigenvalue problem*. In a generalised eigenvalue problem for two $n \times n$ matrices A and B we want to find (for $1 \leq i \leq n$ the eigenvalues $\lambda^{(i)}$ corresponding to the eigenvectors $u^{(i)} \neq 0$ such that

$$A u^{(i)} = \lambda B u^{(i)} \tag{6.17}$$

If we want so solve problem 6.16, the matrices Q_m and C_m must have the same dimension. This condition is satisfied, since both matrix are of size $n \times js$. We can compute the eigenvalues and eigenvectors by using the `eig(Q,C)` command in Matlab. Note that computing these is more economical than computing the eigenvalues of A directly, since the size of Q_m is generally much smaller than the size of A.

Similar to a standard eigenvalue problem, the eigenvalues are approximations to a subset of the eigenvalues of A. Which eigenvalues are approximated, depends on which eigenvector is

used for calculating the residual. In general, the extreme eigenvalues are approximated first. The eigenvectors of A can be approximated by the Ritz vector $\widetilde{V}_m y_m$.

In order to find the approximate eigenvalues and eigenvectors, we need to follow the same steps as in algorithm 6.1. The ultimate goal is to build the matrices Q_m and C_m, so that we can compute the approximated eigenvalues and eigenvectors. Algorithm 6.2 shows an algorithm for PIDR(s) for solving eigenvalue problems. An implementation of this algorithm can be found in Appendix C.4. Note that we don't have to use $\widetilde{W}_m$ in the algorithm, since we can write $\widetilde{V} = \Omega_j(A)V$ and $\widetilde{W} = (\Omega_j(A))^{-1} W$.

Algorithm 6.2 PIDR(s): Projected IDR(s) for eigenvalue problems

1: Select the Krylov subspaces $\mathcal{K}_{m-j}(A, r_0)$ with basis V_{m-j}
2: and the block Krylov subspace $\mathcal{L}_j(A^T, P)$ with basis W_j.
3: Until convergence; **Do**
4: Extend V_{m-j} with s vectors
5: Extend W_j with s vectors
6: Extend $\Omega_j(A)$ to $\Omega_{j+1}(A)$
7: Build basis $\widetilde{V}_m = \Omega_j(A)V_{m-j}$
8: Build basis $\widetilde{W}_j = (\Omega_j(A))^{-1} W_j$
9: Solve $\widetilde{W}_j^T A \widetilde{V}_m y_m = \theta_m \widetilde{W}_j^T \widetilde{V}_m y_m$ and approximate the eigenvalues with θ_m
10: and the eigenvectors with $\widetilde{V}_m y_m$.
11: Check stopping criterion
12: **EndDo**

Chapter 7

Numerical examples PIDR(s)

In this chapter we conducted numerical experiments to test the equivalence of IDR(s) and PIDR(s). In section 7.1 we will look at three examples in which PIDR(s) is used to solve a system of linear equations and in section 7.2 we will look at three examples for PIDR(s) as an eigenvalue problems.

7.1 PIDR(s) for solving systems of linear equations

In this section we will look at three examples in which we solve a system of linear equations using PIDR(s). First we will consider a matrix corresponding to the convection-diffusion equation. This matrix was also used in section 5.3.1. Next we consider the `sherman1` matrix, which is related to the `sherman4` matrix discussed in section 5.3.2. Lastly, we discuss the `jpwh_991` matrix, which was also used as a test example for IDR(s) in section 5.3.4. In all three examples we use a random initial guess x_0, a random vector with omega's and a random right hand side, all three drawn uniformly on the open interval $(0, 1)$. The maximum number of iterations is equal to 1000 and the tolerance (TOL) for the scaled norm ($||r||/||b||$) is 10^{-8}. Finally, the matrix P is an $n \times s$ random matrix whose elements are drawn from the uniform distribution on the open interval $(0, 1)$ and whose columns are pairwise orthogonal.

In these experiments we will compare the convergence behaviour of PIDR(s) (with fixed s) with four other methods: IDR(s), IDR(4), Bi-CG and full GMRES. In each figure we have plotted the number of matrix-vector multiplications (MATVECS) against the scaled norm $||r||/||b||$. On the vertical axis we see the exponents of this norm with base 10. As section 6.4 told, we build the PIDR(s) algorithm from a theoretical point of view. The algorithm is not stable, since we have to compute several inverses in each iteration. In order for the program to work, we often need to choose large values of s, since this decreases the number of iterations and so the possibilities for a breakdown. Each time we need to consider the stability of the method and the number of necessary MATVECS. It is not always true that the smallest value of s yields the best results. The algorithm produces more stable results for larger values of s, since the algorithm needs fewer iterations. Hence, for the sake of reliability of the results, we prefer higher values of s to values of s that result in the fewest iterations. We often use a trial-and-error approach for finding a correct value of s.

Recall that we only compute a solution every $(s+1)$st iteration. Since we use such large values for s, we only need a few iterations. Using that PIDR(s) uses $s + 1$ MATVECS per iteration,

we can find the correct number of MATVECS by multiplying the number of iterations with $s+1$. In a similar fashion, we have to multiply the number of iterations for Bi-CG with two, since Bi-CG uses two MATVECS per iteration (one for A and one for A^T).

Lastly, it is not relevant to include the computation times of the different methods, since our goal is not to create an efficient implementation of PIDR(s). However, we will still include the computation times to give an idea of how the methods perform. An implementation of the program that invokes these three examples can be found in appendix C.1.

7.1.1 Example 1: the convection-diffusion equation

In section 5.3 we considered the matrix corresponding to the centered finite difference discretisation in the unit cube with homogeneous Dirichlet boundary conditions of the operator

$$L(u) = -\Delta u + \beta(u_x + u_y + u_z) = -\Delta u + \beta \nabla u. \tag{7.1}$$

We will discuss two examples. In the first example we take 8 internal points (resulting in a matrix of size 512×512) with the convection parameters $\beta = 100$ and $\beta = 200$. In the second example we take 10 internal points, resulting in a 1000×1000 matrix. We use the same values for β. In Table 7.1 to Table 7.4 we have summarised the results.

Method	**MATVECS**	**CPU time**
GMRES	72	0.1716s
Bi-CG	164	0.0936s
IDR(4)	111	0.0936s
IDR(s)	84	0.0468s
PIDR(s)	108	0.3432s

Table 7.1: Example 7.1 with $\beta = 100$, $s = 35$

Method	**MATVECS**	**CPU time**
GMRES	107	0.3744s
Bi-CG	226	0.0936s
IDR(4)	223	0.1560s
IDR(s)	124	0.1248s
PIDR(s)	123	0.3432s

Table 7.2: Example 7.1 with $\beta = 200$, $s = 40$

Method	**MATVECS**	**CPU time**
GMRES	74	0.5148s
Bi-CG	160	0.4680s
IDR(4)	107	0.4056s
IDR(s)	85	0.3744s
PIDR(s)	123	1.7628s

Table 7.3: Example 7.1 with $\beta = 100$, $s = 40$

Method	**MATVECS**	**CPU time**
GMRES	124	0.9580s
Bi-CG	274	0.7956s
IDR(4)	289	0.9360s
IDR(s)	146	0.8424s
PIDR(s)	192	1.9344s

Table 7.4: Example 7.1 with $\beta = 200$, $s = 95$

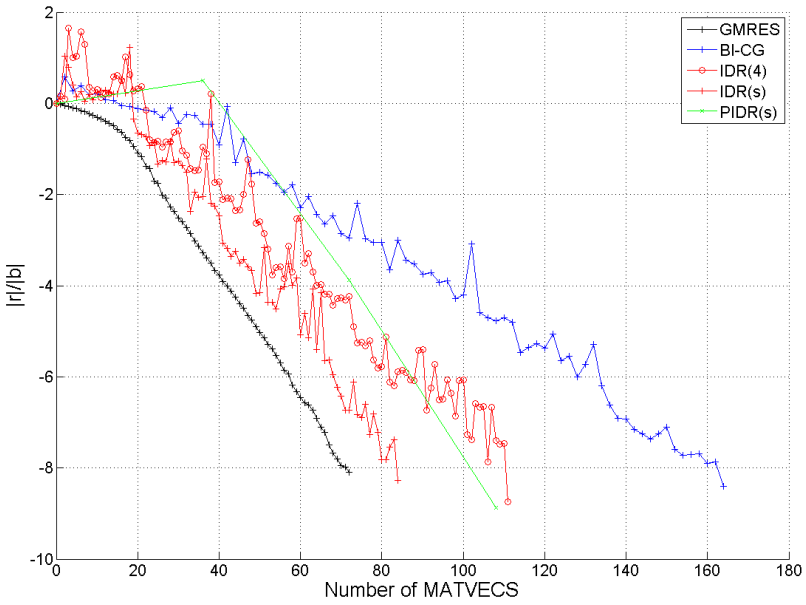

Figure 7.1: Convergence behaviour of the convection-diffusion matrix, $\beta = 100$ and $s = 35$

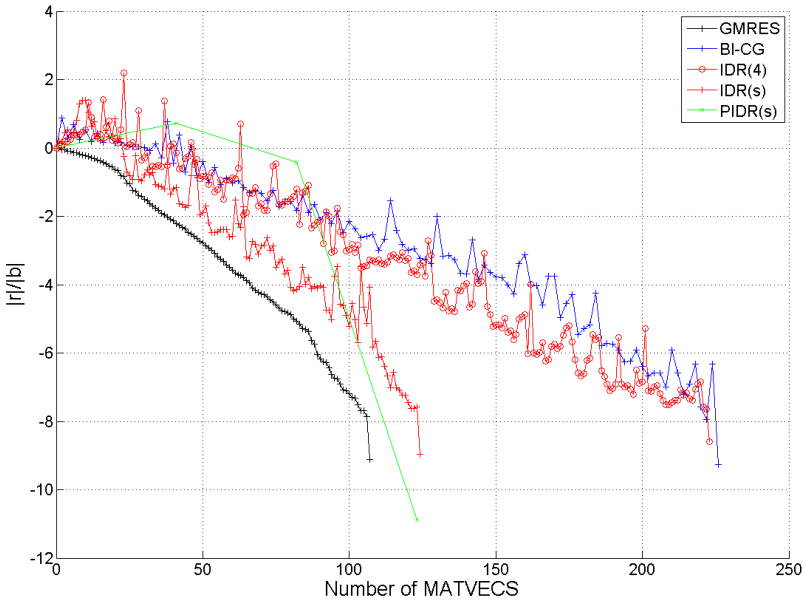

Figure 7.2: Convergence behaviour of the convection-diffusion matrix, $\beta = 200$ and $s = 40$

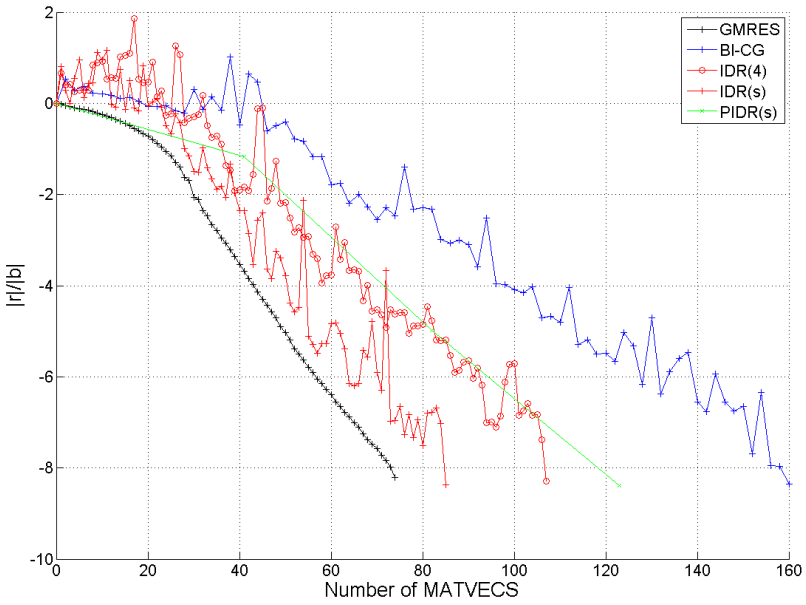

Figure 7.3: Convergence behaviour of the convection-diffusion matrix, $\beta = 100$ and $s = 40$

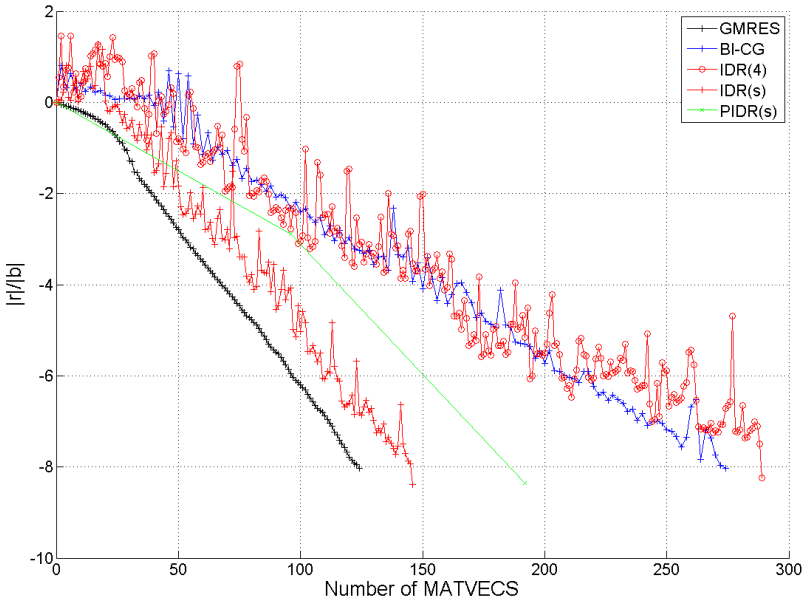

Figure 7.4: Convergence behaviour of the convection-diffusion matrix, $\beta = 200$ and $s = 95$

In Figure 7.1 we see the convergence plot for $n = 512$ with $\beta = 100$ and $s = 35$. The PIDR(s) algorithm behaves slightly worse than IDR(4) when we look at the number of iterations. However, it needs fewer MATVECS than the Bi-CG method, which seems an encouraging result. As predicted, the GMRES method needs the fewest iterations. Note that PIDR(s) only computes a solution and a residual every 36th iteration, so the points of the graph belonging to PIDR(s) are connected with straight line segments.

If we increase the value of β to 200, of which the plot can be seen in Figure 7.2, we see that the PIDR(s) method performs similar to the IDR(s) method. In order to get stable results, we need to increase the value of s to 40. If we look at the convergence curves, we see that the PIDR(s) behaves better for $\beta = 200$. We see that the convergence curves of GMRES and PIDR(s) lie much closer together. This is logical, since the short-recurrence properties of the PIDR(s) algorithm are lost for large s. It seems that in this particular problem, the IDR(s) algorithm seems to behave better for higher s. Just as in section 5.3.1, we see that the IDR(4) method start to behave relatively bad compared to Bi-CG if we increase the value of β.

In Figure 7.3 we have plotted the convergence curve for the five methods for $n = 1000$ and $\beta = 100$. With trial and error, we found that $s = 40$ finds an approximate solution in as few MATVECS as possible, while also being stable. We might find a solution for smaller s, but the PIDR(s) behaves less stable. In Figure 7.4 we see that the PIDR(s) algorithm outperforms the other methods. It is interesting to see that the convergence curve of PIDR(s) lies below the convergence curve of GMRES, which should not be possible from a theoretical point of view, since GMRES minimises the residual in each iteration. However, keep in mind that the points in which PIDR(s) calculates the residual are connected by straight line segments.

7.1.2 Example 2: the `sherman1` matrix

From the Matrix Market we consider the `sherman1` matrix[1], which is related to the `sherman4` matrix from section 5.3.2. This nonsymmetric, real matrix of size 1000×1000 is used in the simulation of oil reservoirs. Its smallest eigenvalue is approximately -5, its largest eigenvalue is approximately 0.

Method	**MATVECS**	**CPU time**
GMRES	363	2.4531s
Bi-CG	1082	0.1406s
IDR(4)	728	0.1562s
IDR(s)	401	2.5625s
PIDR(s)	402	3.1094s

Table 7.5: `sherman1` matrix with $s = 200$

[1] `http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/sherman/sherman1.html`

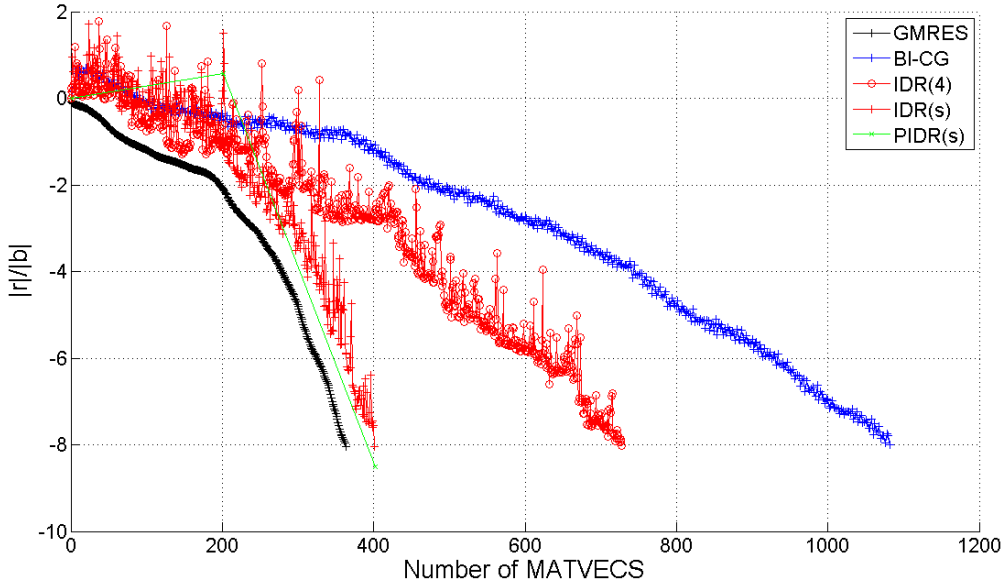

Figure 7.5: Convergence behaviour of the `sherman1` matrix with $s = 200$

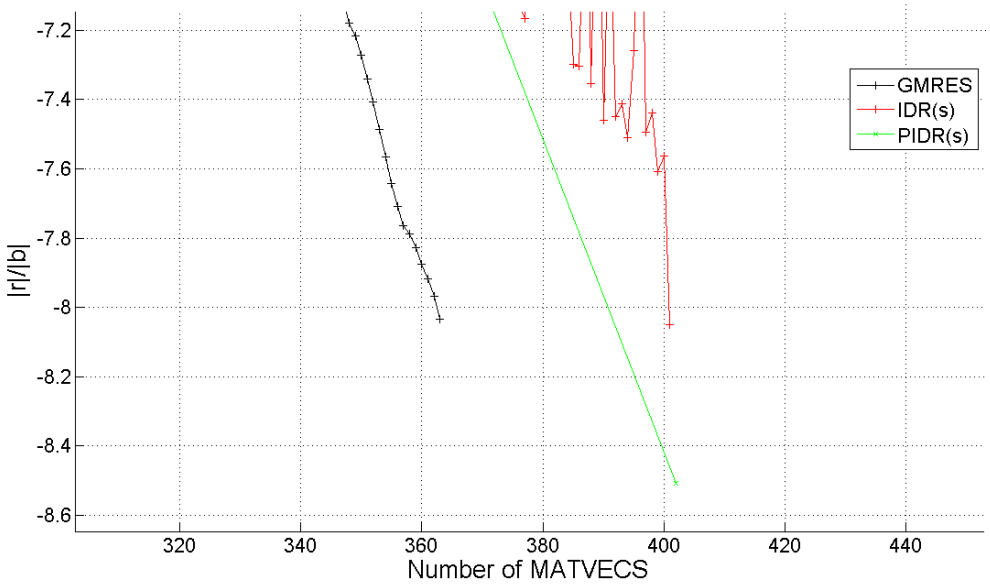

Figure 7.6: Zoomed in convergence behaviour of the `sherman1` matrix with $s = 200$

In Figure 7.5 we see the convergence curves of GMRES, Bi-CG, IDR(4), IDR(s) and PIDR(s) and Figure 7.6 shows a close-up of the last few iterations. We see that PIDR(s) behaves good compared to IDR(4) and Bi-CG and its convergence curve matches that of IDR(s). Also note that the number of MATVECS for IDR(s) and PIDR(s) is only slightly higher than the number of MATVECS for full GMRES. In Table 7.5 we can see the exact number of MATVECS and the required computing times.

7.1.3 Example 3: the `jpwh_991` matrix

The `jpwh_991` matrix also served as a test example for IDR(s) in section 5.3.4. It is a real nonsymmetric 991×991 matrix that is used in circuit physics. The eigenvalues approximately lie between -16 and 0.

Method	**MATVECS**	**CPU time**
GMRES	56	0.1092s
Bi-CG	118	0.0156s
IDR(4)	59	0.1248s
IDR(s)	61	0.1872s
PIDR(s)	72	1.2012s

Table 7.6: `jpwh_991` matrix with $s = 35$

In Figure 7.7 we see the convergence curves of the five projection methods. In Figure 7.8 we have included a zoomed in Figure of the last few iterations of GMRES, IDR(4), IDR(s) and PIDR(s). We see that the number of MATVECS of these four methods lie close together. The number of iterations of the Bi-CG method is approximately the same as the other methods, but since we have to compute two MATVECS per iteration, the total number of MATVECS is twice as large as the number of MATVECS of the other four methods. In Table 7.6 we summarise the results.

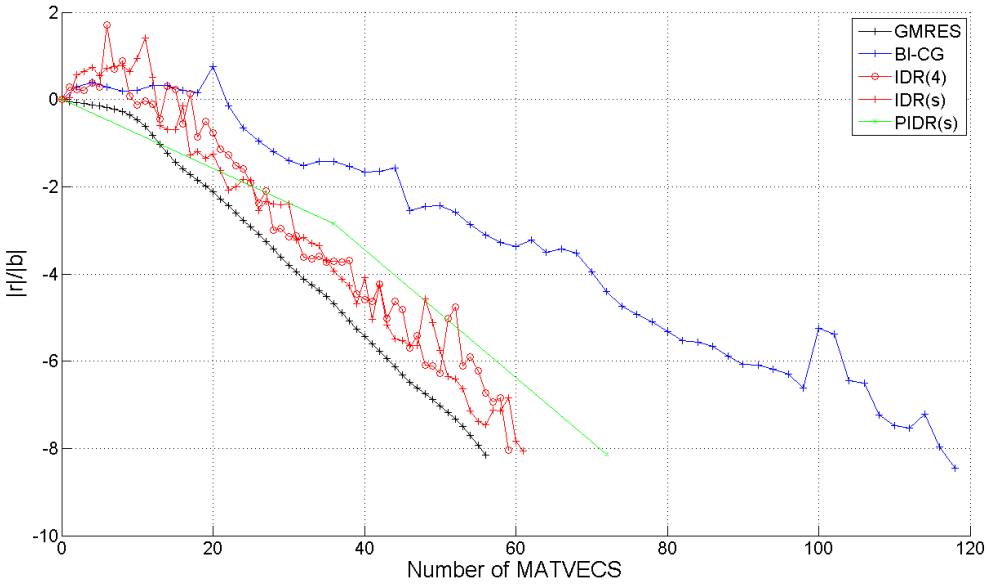

Figure 7.7: Convergence behaviour of the `jpwh_991` matrix with $s = 35$

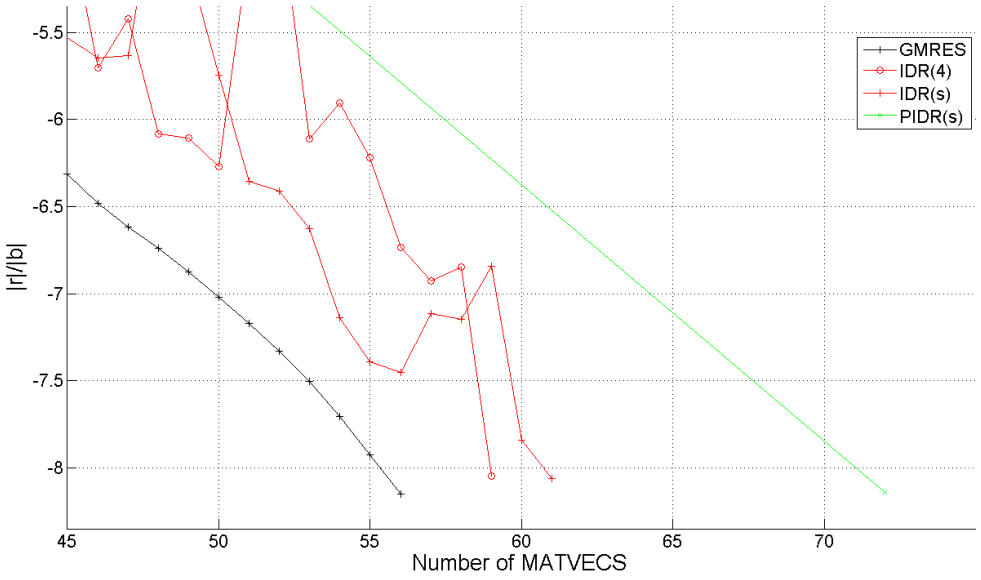

Figure 7.8: Zoomed in convergence behaviour of the `jpwh_991` matrix with $s = 35$

7.2 PIDR(s) for solving eigenvalue problems

In this section we consider three examples of matrices of which we compute the eigenvalues using PIDR(s): the symmetric positive definite Poisson matrix, a nonsymmetric random matrix and the nonsymmetric 'Kahan' matrix. In all three examples we use a random starting vector and a random list of ω_j's, both drawn uniformly on the open interval $(0, 1)$. The maximum number of iterations is equal to 500 and the tolerance (TOL) for the norm is 10^{-8}. Finally, the matrix P is an $n \times s$ random matrix whose elements are drawn from the uniform distribution on the open interval $(0, 1)$ and whose columns are pairwise orthogonal. An implementation of the program that invokes these three examples can be found in C.1

7.2.1 Example 1: the Poisson matrix

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

$$-\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\varphi(x, y) = f(x, y) \tag{7.2}$$

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. (7.3) gives an example for $n = 3$:

$$A = \begin{pmatrix} 4 & -1 & 0 & -1 & & & & & \\ -1 & 4 & -1 & 0 & -1 & & & O & \\ 0 & -1 & 4 & 0 & 0 & -1 & & & \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & & \\ & -1 & 0 & -1 & 4 & -1 & 0 & -1 & \\ & & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ & & & -1 & 0 & 0 & 4 & -1 & 0 \\ & O & & & -1 & 0 & -1 & 4 & -1 \\ & & & & & -1 & 0 & -1 & 4 \end{pmatrix}. \tag{7.3}$$

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 $j = 1 : n$. Since the Poisson matrix is a symmetric matrix, the eigenvalues should be real. For $n \to \infty$, the eigenvalues will lie on the open interval $(0, 8)$. 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.

In Figure 7.9 and Figure 7.10 we have plotted the real part and the imaginary part of each eigenvalue and Ritz value of the Poisson(10) matrix with $s = 4$ and $s = 8$ respectively. For $s = 4$ we see that in most of the runs, PIDR(s) generates the full space (100 basis vectors). Moreover, it is not guaranteed that the algorithm correctly approximates the largest eigenvalue(s), although a few of the extreme eigenvalues are approximated correctly. This is clearly visible in Figure 7.9, in which we can see several spurious (fake) eigenvalues (the eigenvalues outside the open interval $(0, 8)$) on the real axis). Hence, running the algorithm for $s = 4$ has no advantage over using the `eig(A)` command in Matlab.

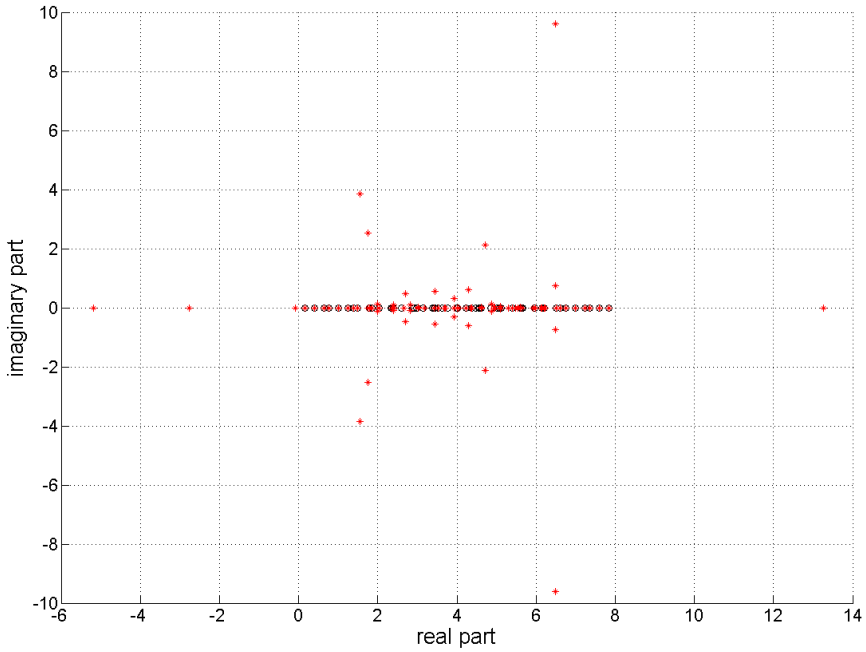

Figure 7.9: Eigenvalues (black) and Ritz values (red) of the Poisson(10) matrix with $s = 4$

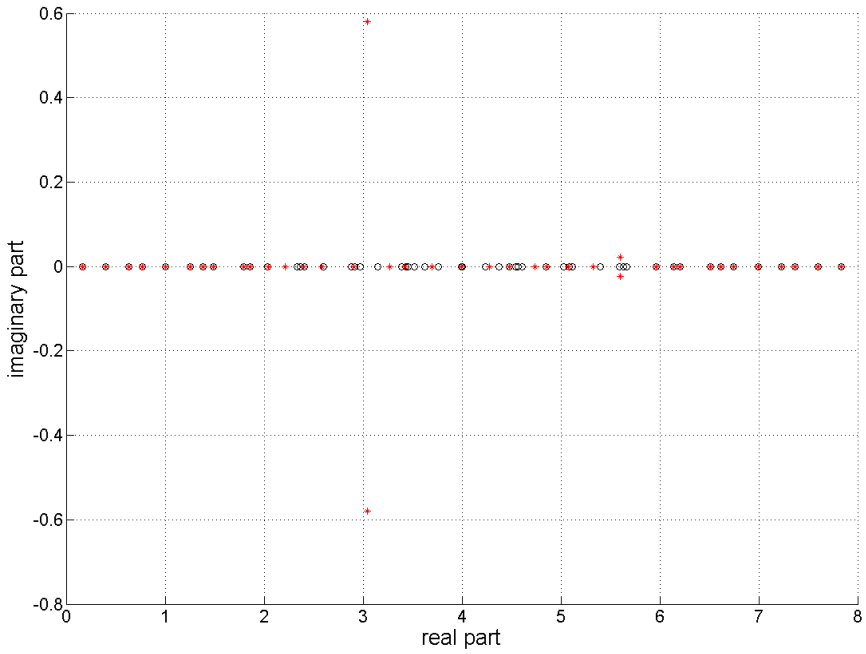

Figure 7.10: Eigenvalues (black) and Ritz values (red) of the Poisson(10) matrix with $s = 8$

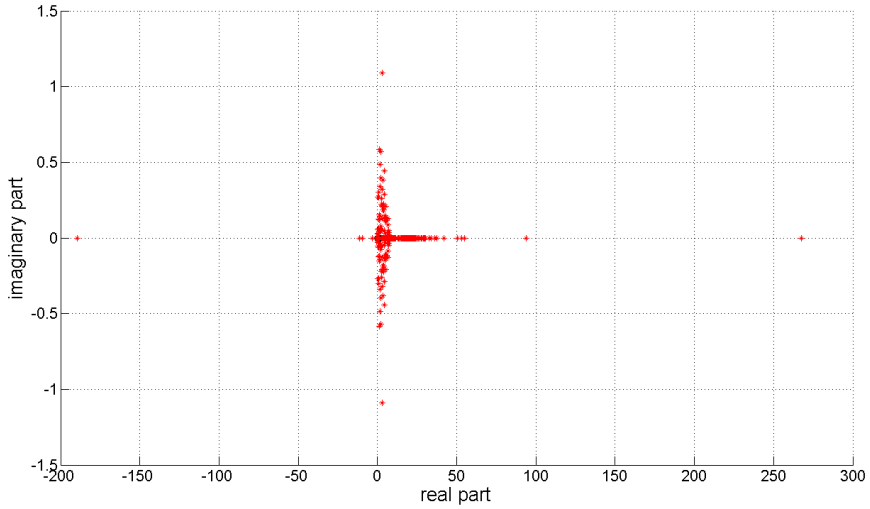

Figure 7.11: Eigenvalues (black) and Ritz values (red) of the Poisson(25) matrix with $s = 8$

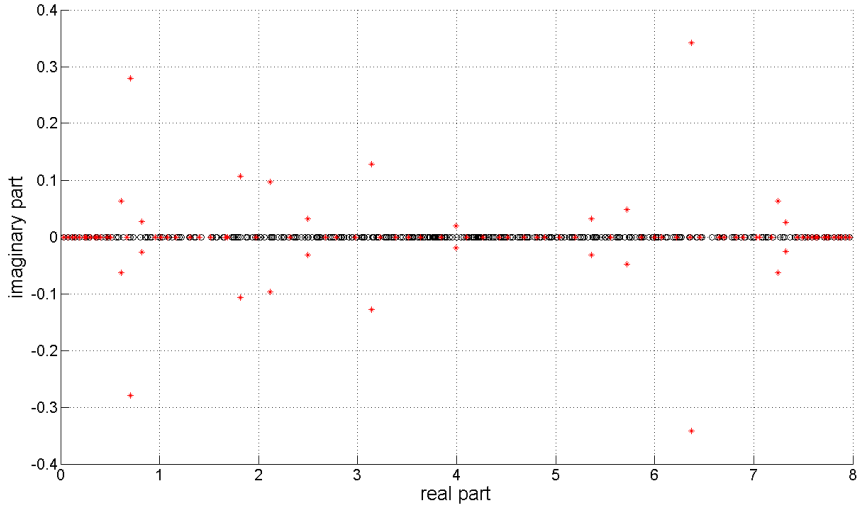

Figure 7.12: Eigenvalues (black) and Ritz values (red) of the Poisson(25) matrix with $s = 32$

For $s = 8$ (and $s > 8$) the algorithm does give advantage over the `eig(A)` command. In most runs, the algorithm generates approximately 40 - 48 basis vectors. The extreme eigenvalues are approximated correctly and only a few spurious eigenvalues are computed. We see that the real part of every Ritz value lies on the open interval $(0, 8)$.

In Figure 7.11 and Figure 7.12 we have plotted the real part and the imaginary part of each eigenvalue and Ritz value of the Poisson(25) matrix with $s = 8$ and $s = 32$ respectively. For $s = 8$ we see the same behaviour as for the Poisson(10) matrix with $s = 4$. The algorithm generates the full space and most of the approximated eigenvalues are spurious. For $s = 32$ (and consequently $s > 16$) the algorithm seems stable and it generates approximately 96 to 160 basis vectors, depending on the first basis vector and the values of ω_j. The extreme eigenvalues are approximated correctly and we see that the real part of each Ritz value lies on the open interval $(0, 8)$.

7.2.2 Example 2: the rand(n) matrix

The `rand(n)` command returns a random matrix of size $n \times n$ which contains pseudorandom values drawn from the standard uniform distribution on the open interval(0,1). Since the rand(n) matrix is nonsymmetric, the eigenvalues can be real as well as imaginary and they are all close to 0. Note that the largest eigenvalue is approximately equal to $n/2$.

In Figure 7.13 and Figure 7.14 we have plotted the real part and the imaginary part of each eigenvalue and Ritz value of the rand(100) matrix with $s = 2$ and $s = 4$.

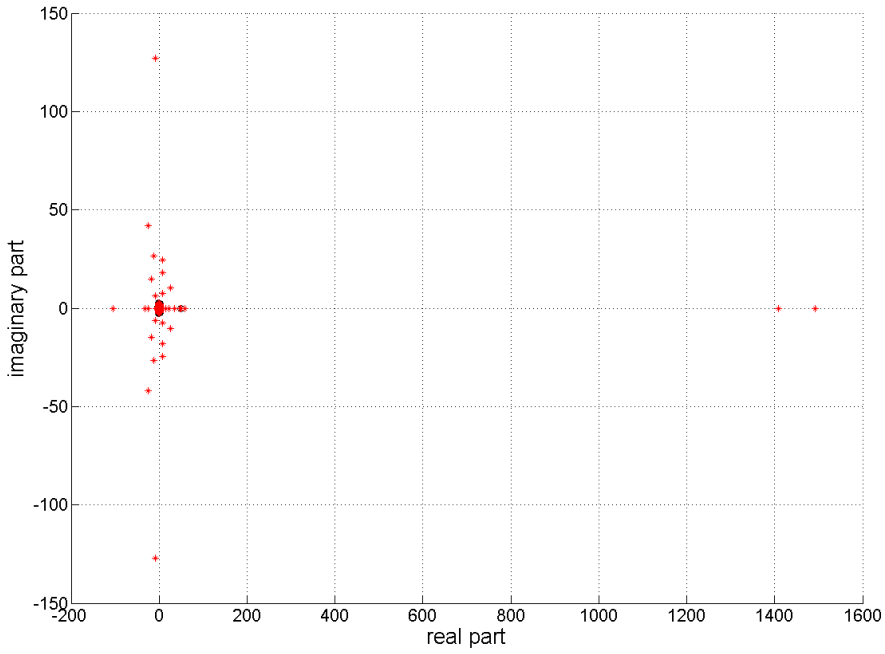

Figure 7.13: Eigenvalues (black) and Ritz values (red) of the rand(100) matrix with $s = 2$

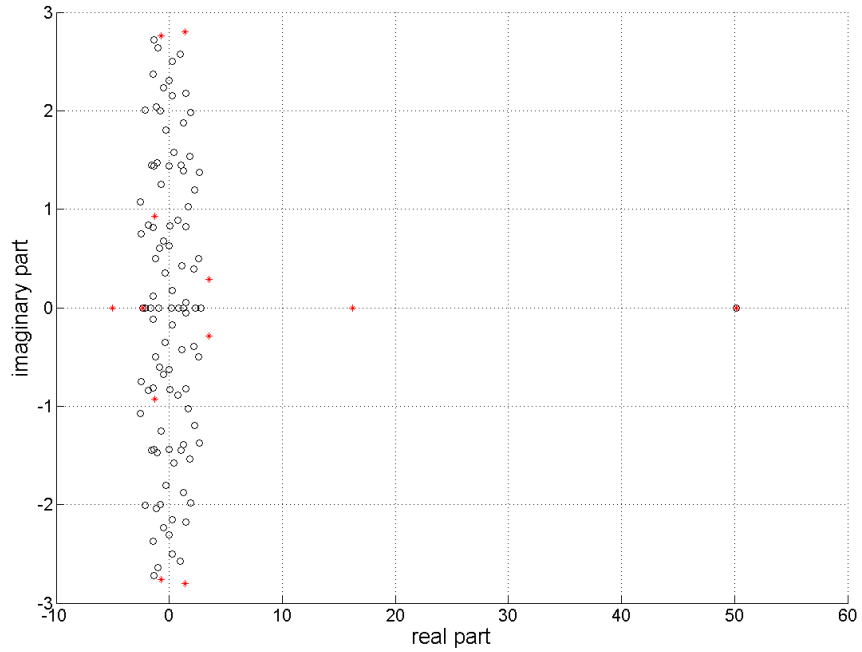

Figure 7.14: Eigenvalues (black) and Ritz values (red) of the rand(100) matrix with $s = 4$

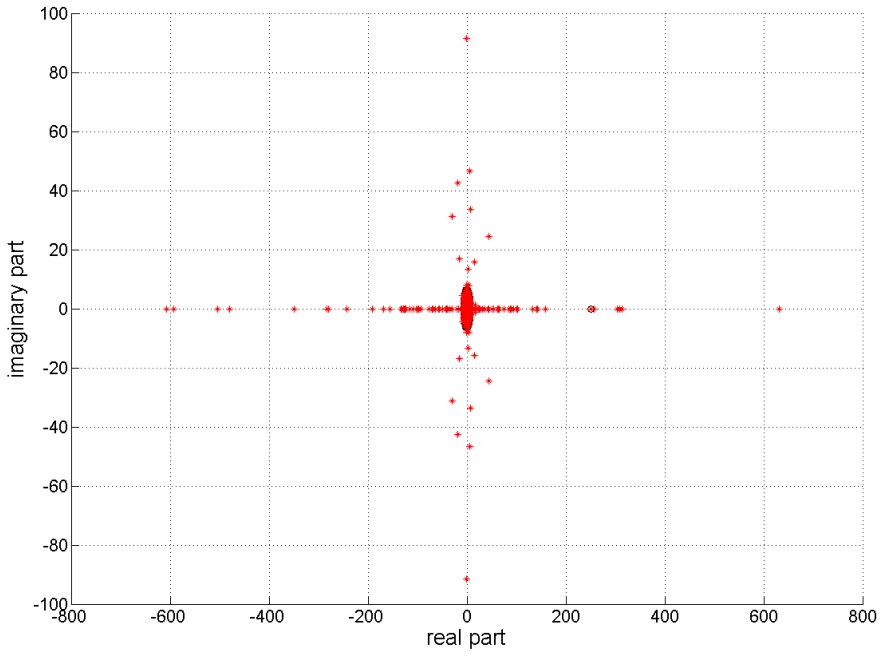

Figure 7.15: Eigenvalues (black) and Ritz values (red) of the rand(500) matrix with $s = 4$

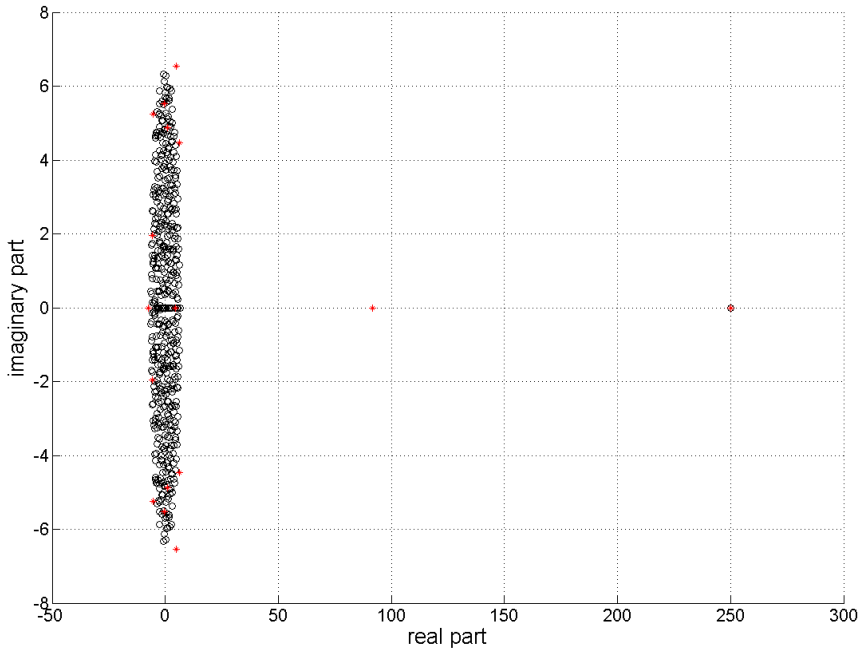

Figure 7.16: Eigenvalues (black) and Ritz values (red) of the rand(500) matrix with $s = 8$

For $s = 2$, the largest eigenvalue is approximately 50 and hence we immediately see in Figure 7.13 that we have a few outliers for the approximated eigenvalues. When we take a closer look, we see that the PIDR(s) algorithm generates the full Krylov basis and hence the results are unreliable. For $s = 4$ the algorithm gives more stable results. In most of the runs, the algorithm generates a Krylov basis of size 12, meaning that we have 12 approximated eigenvalues. As can be seen in Figure 7.14, the largest eigenvalue is approximated correctly. Running the algorithm with larger values of s does not yield better results. No extra eigenvalues are approximated and the algorithm computes more basis vectors.

In Figure 7.15 and Figure 7.16 we have plotted the real part and the imaginary part of each eigenvalue and Ritz value of the rand(500) matrix with $s = 4$ and $s = 8$. For $s = 4$ the PIDR(s) algorithm generates the full Krylov base. None of the eigenvalues are approximated correctly and there are several spurious eigenvalues. For $s = 8$ the results improve, and we see that the PIDR(s) algorithm correctly approximates the largest eigenvalue. The Krylov basis consists of approximately 16 vectors. If we choose s wisely, we might generate a slightly smaller basis and still approximate the largest eigenvalue correctly.

7.2.3 Example 3: the Kahan(n,θ,ϵ) matrix

The Kahan matrix is an upper triangular matrix that has interesting properties regarding estimation of condition and rank. The Matlab command `gallery('kahan',n,`θ`,`ϵ`)`[2] invokes this matrix. In our test problem, we use the default settings $\theta = 1.2$ and $\epsilon = 25$. An example for $n = 6$ is given in 7.4 (all elements are rounded to four decimals):

[2]see http://math.berkeley.edu/ mgu/MA221/kahan.m for information about θ and ϵ

$$A = \begin{pmatrix} 1.0000 & -0.3624 & -0.3624 & -0.3624 & -0.3624 & -0.3624 \\ & 0.9320 & -0.3377 & -0.3377 & -0.3377 & -0.3377 \\ & & 0.8687 & -0.3148 & -0.3148 & -0.3148 \\ & & & 0.8097 & -0.2934 & -0.2934 \\ & & O & & 0.7546 & -0.2734 \\ & & & & & 0.7033 \end{pmatrix}.$$

Note that the elements on the main diagonal are monotonically decreasing and that the non-zero elements in each column are monotonically increasing. Since the eigenvalues of any triangular matrix are on the main diagonal, most eigenvalues are approximately 0 for large values of n. Moreover, all the eigenvalues are real and lie on the half open interval $(0, 1 + \xi]$, where $\xi = \epsilon \cdot \text{eps} \cdot n$ (eps is the machine precision).

In Figure 7.17 and Figure 7.18 we have plotted the real part and the imaginary part of each eigenvalue and Ritz value of the Kahan(500) matrix with $s = 2$ and $s = 4$. For $s = 2$ the PIDR(s) algorithm generates the full Krylov space (of size 500) and the extreme eigenvalues are not approximated correctly. The eigenvalues should lie on the half open interval $(0, 1 + \xi]$ and this is clearly not the case. For $s = 4$ the PIDR(s) algorithm does approximate the extreme eigenvalues correctly. It is clearly visible that the eigenvalues cluster around 0. The size of the Krylov basis is approximately 24 to 28. For larger values of s the algorithm works fine, but we expect that the dimension of the Krylov basis is a little greater than the dimension of the basis corresponding to $s = 4$.

In Figure 7.19 we have plotted the real part and the imaginary part of each eigenvalue and Ritz value of the Kahan(1000) matrix with $s = 4$. For $s = 2$ the PIDR(s) algorithm doesn't generate any output. In each iteration we have to compute the eigenvalues and eigenvectors in order to compute the norm. The algorithm may become slow if the dimension of the basis grows large.

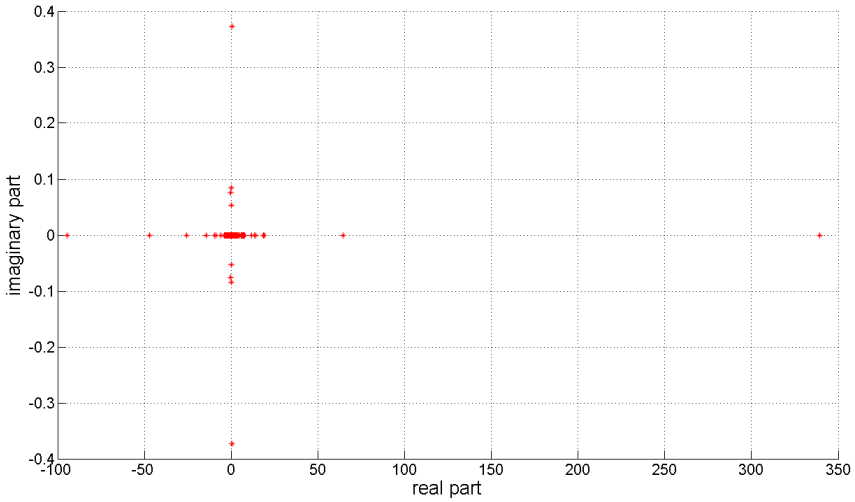

Figure 7.17: Eigenvalues (black) and Ritz values (red) of the Kahan(500) matrix with $s = 2$

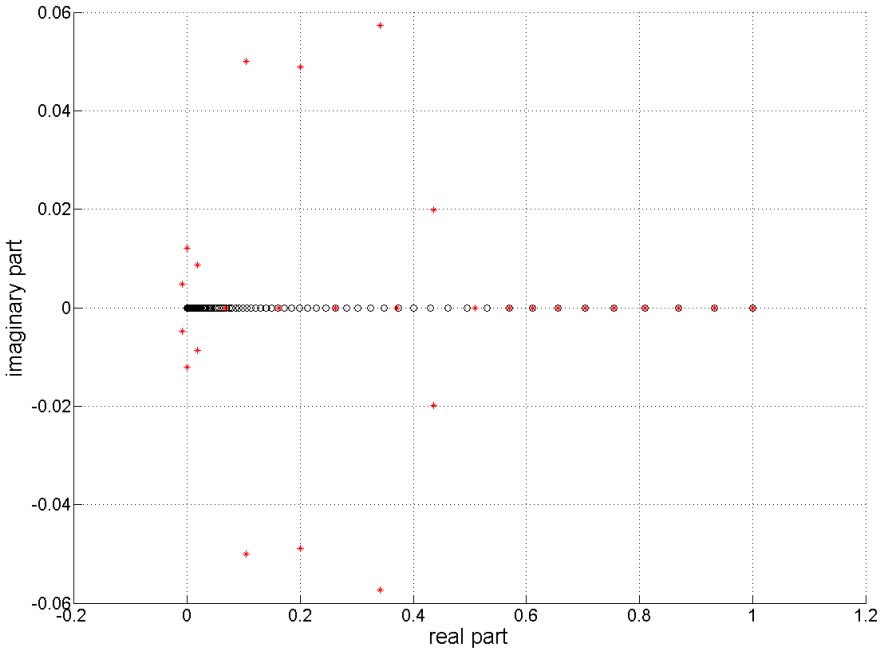

Figure 7.18: Eigenvalues (black) and Ritz values (red) of the Kahan(500) matrix with $s = 4$

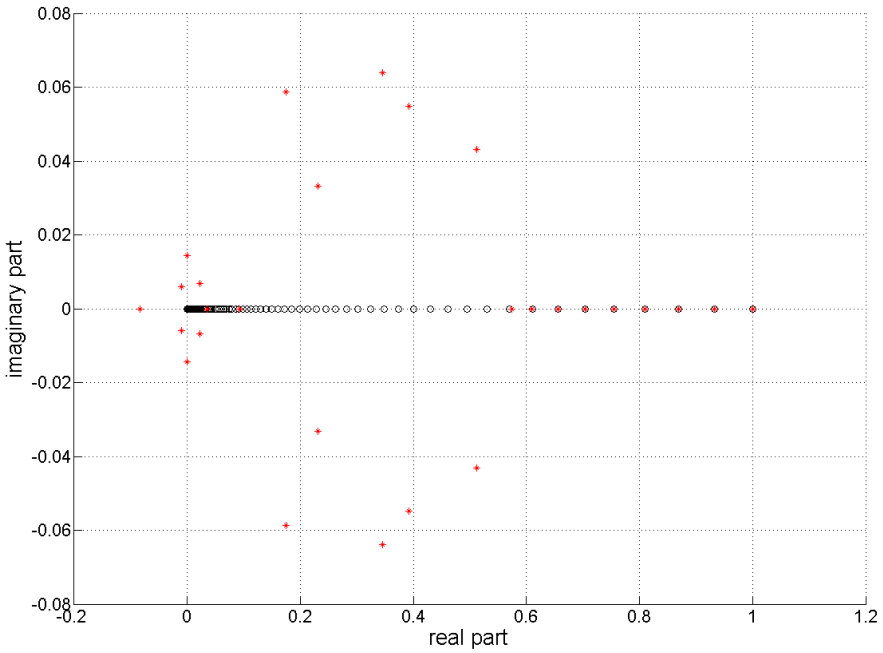

Figure 7.19: Eigenvalues (black) and Ritz values (red) of the Kahan(1000) matrix with $s = 4$

Chapter 8

Conclusions

In this thesis we considered the IDR(s) method, which is short for the Induced Dimension Reduction(s) method. IDR(s) is one of the many Krylov subspace methods that are used to solve systems of linear equations. Krylov subspace methods can be seen in the framework of projection methods, which are iterative methods that try to find an approximate solution in a subspace $\mathcal{K}$ of $\mathbb{C}^n$ such that the corresponding residual is orthogonal to a subspace $\mathcal{L}$ of $\mathbb{C}^n$. It was unclear how we could implement IDR(s) as a projection method and this is unfortunate, since we know much about these methods. Hence, our goal was to present an implementation of IDR(s) as a projection method.

In chapter 3 we explained the theory behind projection methods. It turned out that all projection methods, both for eigenvalue problems and solving systems of linear equations, can be described using a general algorithm. In this chapter we also considered Krylov subspaces methods, which are a special class of projection methods. In chapter 4 we implemented and described several Krylov subspace methods to get acquainted with the subject.

In chapter 5 we discussed the IDR(s) method, which forms the basis of this project. We derived the algorithm using the IDR theorem and the definition of the residuals of a general Krylov-type solver. To illustrate its functionality, we included four examples, in which we showed that IDR(s) can be a good alternative to solve systems of linear equations. In chapter 6 we have formulated IDR(s) as a projection method (which we call PIDR(s)) and we proposed two algorithms for the PIDR(s) method: one for solving system of linear equations and one for solving eigenvalue problems. We first derived the correct right subspace and showed that, together with the left subspace, PIDR(s) is indeed a projection method. The three numerical examples in section 7.1 show that the PIDR(s) method gives results that are consistent with the results of the standard IDR(s) method. In section 7.2 we showed that PIDR(s) gives correct approximations of the eigenvalues.

8.1 Summary of the results

In section 6.4 we presented an algorithm for the PIDR(s) method from a theoretical point of view. This means that our algorithm is not optimal, in the sense that it does not efficiently compute a solution. This does not mean that the algorithm is useless, because PIDR(s) shows that IDR(s) can indeed be seen in the framework of projection methods. We based the

PIDR(s) algorithm on the general algorithm for projection methods described in section 3.1, where we have used the following definitions for the right and left subspace:

$$\widetilde{\mathcal{K}}_m(A, \widetilde{r}_0) \;=\; \Omega_j(A)\, \mathcal{K}_{m-j}(A, r_0) \;=\; \mathcal{K}_{m-j}(A,\ \Omega_j(A) r_0)$$

$$\widetilde{\mathcal{L}}_j(A^T, \widetilde{P}) \;=\; (\Omega_j(A))^{-T}\, \mathcal{K}_j(A^T, P) \;=\; \mathcal{K}_j(A^T,\ (\Omega_j(A))^{-T}\, P)$$

Algorithm 6.1, which uses the bases $\widetilde{V}_m$ for the right subspace $\widetilde{\mathcal{K}}_m$ and the basis $\widetilde{W}_j$ for the left subspace $\widetilde{\mathcal{L}}_j$, shows a general algorithm of PIDR(s) . We present three implementations of this algorithm in appendix C.2, C.3 and C.4. The first two are implementations for PIDR(s) that solves systems of linear equations, the last is an implementation for PIDR(s) that solves eigenvalue problems. First, note that we explicitly compute the left subspace $\widetilde{W}_j$ and right subspace $\widetilde{V}_m$ in the implementations, which differs from the original version IDR(s). Secondly, we see that in the implementation for eigenvalue problems, we do not need to compute $\widetilde{W}_j$, which suggests that PIDR(s) for systems of linear equations uses a different right subspace than PIDR(s) for eigenvalue problems. Lastly, note that we can only compute a unique solution every $(s+1)$st iteration, since the inverse of the matrix $\widetilde{W}_j^{\,T} A \widetilde{V}_m$ is not defined in the intermediate iterations.

8.2 Recommendations for future research

With the implementation of PIDR(s) for systems of linear equations, we have shown that IDR(s) can be seen in the framework of projection methods. The PIDR(s) algorithm for solving systems of linear equations is not optimal, but this does not matter, since we already have an efficient algorithm for IDR(s). Hence, further research with the goal of improving the PIDR(s) algorithm is not useful.

Instead we can focus on the PIDR(s) method for solving eigenvalue problems. We already have a substantial knowledge about IDR(s) as a method for solving linear systems, but far less is known about IDR(s) as a method to approximate eigenvalues. In this light, the numerical experiments for PIDR(s) in section 7.2 show promising results and these results can be used to create a thorough theoretical framework for IDR(s) as an eigenvalue method.

To conclude, Simoncini and Szyld [12] present a modification of IDR(s), called Ritz-IDR, in which they use the reciprocals of a subset of the Ritz values as the values for ω_j. The Ritz values are computed using twenty iterations of the Arnoldi method. If we derive an efficient algorithm for IDR(s) for eigenvalue problems, we can compute the Ritz values with IDR(s) itself.

Bibliography

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

[2] R. Astudillo and M. B. van Gijzen. An Induced Dimension Reduction Algorithm to Approximate Eigenpairs of Large Nonsymmetric Matrices. In *11th International Conference of Numerical Analysis and Applied Mathematics 2013*, volume 1558, pages 2277–2280. AIP Publishing, 2013.

[3] V. Faber and T. Manteuffel. Necessary and Sufficient Conditions for the Existence of a Conjugate Gradient Method. *SIAM Journal on Numerical Analasys*, 21(2):352–362, January 1984.

[4] R. Fletcher. Conjugate Gradient Methods for Indefinite Systems. In *Numerical Analysis*, volume 506 of *Lecture Notes in Mathematics*, pages 73–89. Springer Berlin Heidelberg, 1976.

[5] G. H. Golub and C. F. van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore, MD, USA, 3rd edition, 1996.

[6] M. H. Gutknecht and J.-P. M. Zemke. Eigenvalue Computations Based on IDR. *SIAM Journal on Matrix Analysis and Applications*, 34(2):283–311, 2013.

[7] S. Guttel. Rational Krylov Approximation of Matrix Functions: Numerical Methods and Optimal Pole Selection. *GAMM-Mitteilungen*, 36(1):8–31, 2013.

[8] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952.

[9] C. Lanczos. An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators. *Journal of Research of the National Bureau of Standards*, 43(4):255–282, October 1950.

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

[11] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, July 1986.

[12] V. Simoncini and D. B. Szyld. Interpreting IDR as a Petrov-Galerkin Method. *SIAM Journal on Scientific Computing*, 32(4):1898–1912, June 2010.

[13] G. L. G. Sleijpen, P. Sonneveld, and M. B. van Gijzen. Bi-CGSTAB as an Induced Dimension Reduction method. *Applied Numerical Mathematics*, 60(11):1100–1114, November 2010.

[14] T. Sogabe, M. Sugihara, and S. L. Zhang. An extension of the Conjugate Residual Method to Nonsymmetric Linear Systems. *Journal of Computational and Applied Mathematics*, 226(1):101–113, April 2009.

[15] P. Sonneveld. CGS, a Fast Lanczos-type Solver for Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 10(1):36–52, 1989.

[16] P. Sonneveld and M. B. van Gijzen. 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, November 2008.

[17] P. Sonneveld and M. B. van Gijzen. An Elegant IDR(s) Variant that Efficiently Exploits Bi-orthogonality Properties. *ACM Transactions on Mathematical Software*, 38(1), November 2011.

[18] H. A. van der Vorst. BI-CGSTAB: A Fast and Smoothly Converging Variant of BI-CG for the Solution of Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, March 1992.

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

[20] P. Wesseling and P. Sonneveld. 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, 1980.

Appendix A

Implentations for solving eigenvalue problems

A.1 Arnoldi.m

```
function [Vec,Val,resvec,iter]=Arnoldi(A,v1,m)

n = length(A);

% Calculations & declarations
V(:,1) = v1 / norm(v1);

for j=1:m

    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
    [Vec,Val] = sorteig(H(1:j,1:j));
    s         = Vec(:,j);

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

end

iter = j;

% Approximation of the eigenvalues and eigenvectors of A
Val = sort(diag(Val));
Vec = V(1:end,1:j) * Vec;
```

A.2 Lanczos.m

```
function [Vec,Val,resvec,iter]=Lanczos(A,v1,m)

n = length(A);

% Calculations & declarations
V(:,1) = zeros(n,1);
V(:,2) = v1 / norm(v1);

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

for j=1:m

    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) <= 1e-15
        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
    [Vec,Val] = sorteig(T);
    y         = Vec(:,j);

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

end

iter = j

% Approximation of the eigenvectors and eigenvalues of A
Val = sort(diag(Val));
Vec = V(:,2:end-1) * Vec;
```

A.3 Bi_Lanczos.m

```
function [Vec,Val,resvec,iter]=Bi_Lanczos(A,v1,m)

n = length(A);

% Calculations & declarations
V(:,1)   = zeros(n,1);
W(:,1)   = V(:,1);
V(:,2)   = v1 / norm(v1);
W(:,2)   = V(:,2);

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

for j = 1:m

    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);
    delta(j+1) = sqrt(abs(Vbar' * Wbar));

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

    beta(j+1) = (Vbar' * Wbar) / 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
    [Vec,Val] = sorteig(T);
    y         = Vec(:,j);

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

end

iter = j;

% Approximation of the eigenvectors of A
Val = sort(diag(Vec));
Vec = V(:,2:end-1) * Val;
```

Appendix B

Implementations for solving systems of linear equations

B.1 FOM.m

```
function [V,x,resvec,iter]=FOM(A,b,x0,m)

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

for j=1:m

    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);

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

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

end

iter = j;                       % Number of iterations
x    = x0 + V(:,1:j)*y;         % Solution
```

B.2 GMRES.m

```
function [V,x,resvec,iter]=GMRES(A,b,x0,m)

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

for j=1:m

    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);

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

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

end

iter = j;                       % Number of iterations
x    = x0 + V(:,1:j)*y;         % Solution
```

B.3 CG.m

```
function [x, resvec, iter] = CG(A,b,x,maxit)

% Calculations & declarations
r          = b - A*x;
p          = r;
resvec(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
    resvec(j+1) = norm(r);
    if  resvec(j+1) / normb < 10^-8
            break
    end
end

iter = j;              % Number of iterations
```

B.4 CR.m

```
function [x, resvec, iter] = CR(A,b,x,m)

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

for j=1:m

    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
    resvec(j+1) = norm(r);
    if  resvec(j+1) / normb < 10^-8
            break
    end
end

iter = j;           % Number of iterations
```

B.5 Bi_CG.m

```
function [x, resvec, iter] = Bi_CG(A,b,x,maxit)

% Calculations & declarations
r         = b - A*x;
p         = r;
rster     = r;
pster     = p;
normb     = norm(b);
resvec(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
    resvec(j+1) = norm(r);
    if  resvec(j+1) < normb * 10^-8
            break
    end

end

iter = j;              % Number of iterations
```

B.6 Bi_CR.m

```
function [x, resvec, iter] = Bi_CR(A,b,x,m)

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

for j=1:m

    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
    resvec(j+1) = norm(r);
    if  resvec(j+1) < normb * 10^-8
            break
    end

end

iter = j;               % Number of iterations
```

Appendix C

PIDR(s) files

C.1 main.m

```
clear all;
close all;
clc;

% Defaults
s       = 200;                  % Block size
n       = 10;                   % Dimension of the problem
m       = 10;                   % #internal points in unit cube
maxit   = 1200;                 % Maximum number of iterations
tol     = 10^-8;                % tolerance of the residual
beta    = 200;                  % parameter for convection term

% System defaults
A     = mmread('.\Matrices\sherman1.mtx'); n = size(A,1)

x0    = zeros(n,1);             % Initial guess for linear system
b     = rand(n,1);              % RHS of linear system
v1    = rand(n,1);              % starting vector for eigenvalue problem
omega = rand(n,1);              % random list of omegas
P     = orth(rand(n,s));        % Random matrix for IDR(s)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%   Convergence plots   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[~] =  idrs_example(A,b,x0,s,maxit,tol,omega,P);
[~] = pidrs_example(A,b,x0,s,maxit,tol,omega,P);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  IDR as a projection method for eigenvalue problems and linear systems  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 % disp('pidrs iteration');
 [Vhat,x,resvec,iter] = pidrs(A,b,x0,s,maxit,tol,omega,P);
 Residual_vector      = resvec'
 Iterations           = iter
 semilogy(Residual_vector)

 % disp('pidrs_eachiter iteration');
 [Vhat,x,resvec,iter] = pidrs_eachiter(A,b,x0,s,maxit,tol,omega,P);
 Residual_vector      = resvec'
 Iterations           = iter
 semilogy(Residual_vector)

 % disp('pidrs_eigenvalue iteration');
 [Vec,Val,resvec,iter] = pidrs_eigenvalue(A,v1,s,maxit,tol,omega,P);
 Residual_vector       = resvec';
 Iterations            = iter;
 semilogy(Residual_vector);
```

C.2 pidrs.m

```
function [Vhat,x,resvec,iter] = pidrs(A,b,x0,s,maxit,tol,omega,P)

n = length(A);

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

% Generate defaults of the IDR(s) algorithm
I = eye(n);                       % Identity matrix
B = I;                            % Initial polynomial

% Starting vector for the Krylov subspace K(A,r0)
V(:,1) = r0 / norm(r0);

for j=1:maxit

    % Updating the polynomial
    B = (I - omega(j)*A) * B;

    % Building a basis W for Km(A',Omega*P) and
    % a basis V for Km(A,Omega*r0)
    if j==1
        V    = Arnoldi_Basis(A,V,s-1);
        W    = P;
        Vhat = B * V;
        What = B'\W;
    else
        V    = Arnoldi_Basis(A,V,s);
        W    = Arnoldi_Basis_Block(A',W,1,s);
        Vhat = B * V;
        What = B'\W;
    end

    % Calculating the solution
    y = (What'*A*Vhat) \ (What'*r0);
    x = x0 + Vhat*y;

    % Calculating the norm and checking stopping criterion
    resvec(j+1) = norm(b-A*x);
    if resvec(j+1) < normb * tol
        break
    end
end

iter = j;
```

C.3 pidrs_eachiter.m

```
function [Vhat,x,resvec,iter] = pidrs_eachiter(A,b,x0,s,maxit,tol,omega,P)

n = length(A);

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

% Generate defaults of the IDR(s) algorithm
I = eye(n);                     % Identity matrix
B = I;                          % Initial polynomial

% Starting vector for the Krylov subspace K(A,r0)
V(:,1) = r0 / norm(r0);

%Do s steps of FOM
V = Arnoldi_Basis(A,V,s-1);

for j=s+1:maxit

    if mod(j,s+1)~=0
        V = Arnoldi_Basis(A,V,1);
    else

        % Updating the polynomial
        B = (I - omega(j/(s+1))*A)*B;

        % Building the basis What for Km(A,P)
        if j == s+1
            W    = P;
            Vhat = B * V;
            What = inv(B') * W;
        else
            W    = Arnoldi_Basis_Block(A',W,1,s);
            Vhat = B * V;
            What = inv(B') * W;
        end

        % Calculating the solution
        y = (What'*A*Vhat) \ (What'*r0);
        x = x0 + Vhat*y;

        % Calculating the norm and checking stopping criterion
        resvec(j/(s+1)+1) = norm(b-A*x);
        if resvec(j/(s+1)+1) < normb * tol
            break
        end
    end
end

iter              = j / (s+1);
```

C.4 pidrs_eigenvalue.m

```
function [Vec,Val,resvec,iter]=pidrs_eigenvalue(A,v1,s,maxit,tol,omega,P)

n = length(A);

% Starting vector for K(A,r0)
V(:,1) = v1 / norm(v1);

% Defaults
I = eye(n);
B = I;

% Start of the algorithm
for j = 1:n/s

    % Updating the polynomial
    B = (I - omega(j)*A) * B;

    % Building a basis W for Km(A',Omega*P) and
    % a basis V for Km(A,Omega*r0)
    if j==1
        V    = Arnoldi_Basis(A,V,s-1);
        W    = P;
        Vhat = B * V;
        What = B'\W;
    else
        V    = Arnoldi_Basis(A,V,s);
        W    = Arnoldi_Basis_Block(A',W,1,s);
        Vhat = B * V;
        What = B'\W;
    end

    % Calculation of the eigenvalues and eigenvectors
    Q = What'*A*Vhat;
    C = What'*Vhat;
    [Vec, Val] = sorteig(Q,C);

    %Residual vector and stopping criterion
    resvec(j)=norm(A*Vhat*Vec(:,end)-Val(end,end)*Vhat*Vec(:,end));
    if resvec(j) < tol
        break
    end
end

iter = j;

% Approximation of the eigenvalues and eigenvectors of A
Val = sort(diag(Val));
Vec = V * Vec;

% Make plot of eigenvalues and ritzvalues
plot(real(EigReal), imag(EigReal), 'ok')
hold on
plot(real(EigApprox), imag(EigApprox), 'r*')
```

C.5 pidrs_example.m

```
function [A] = pidrs_example(A,b,x0,s,maxit,tol,omega,P)

options.omega = omega;

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

choice = 1;
scrsz = get(0,'ScreenSize');
fig = figure('Position',[scrsz(1) + scrsz(3)/2 scrsz(4)/2
                         -80 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,size(A,1),[],[],x0);
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,[],[],x0);
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(' ');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
54
55 t = cputime;
56 disp('IDR(4) iteration...');
57 [x,flag,relres,iter,resvec] = idrs(A,b,4,tol,maxit,[],[],x0,options.omega );
58 time = cputime - t;
59 resvec = log10(resvec/resvec(1));
60 figure(fig);
61 it = [0:1:length(resvec)-1];
62 plot(it,resvec,'r-o');
63 drawnow;
64 disp(['Final accuracy: ', num2str(norm(b-A*x)/norm(b))]);
65 disp(['Iterations: ',num2str(iter)]);
66 disp(['CPU time: ',num2str(time),'s.']);
67 disp(' ');
68
69 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
70
71 t = cputime;
72 disp('IDR(s) iteration...');
73 [x,flag,relres,iter,resvec] = idrs(A,b,s,tol,maxit,[],[],x0,options.omega );
74 time = cputime - t;
75 resvec = log10(resvec/resvec(1));
76 figure(fig);
77 it = [0:1:length(resvec)-1];
78 plot(it,resvec,'r-+');
79 drawnow;
80 disp(['Final accuracy: ', num2str(norm(b-A*x)/norm(b))]);
81 disp(['Iterations: ',num2str(iter)]);
82 disp(['CPU time: ',num2str(time),'s.']);
83 disp(' ');
84
85 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
86
87 t = cputime;
88 disp('PIDR(s) iteration...');
89 [Vhat,x,resvec,iter]=pidrs(A,b,x0,s,maxit,tol,omega,P);
90 sizeVhat = size(Vhat,2);
91 iter = iter * (s+1);
92 time = cputime - t;
93 figure(fig);
94 resvec = log10(resvec/resvec(1));
95 it = [0:(s+1):(s+1)*(length(resvec)-1)];
96 plot(it,resvec,'g-x');
97 drawnow;
98 disp(['Final accuracy: ', num2str(norm(b-A*x)/norm(b))])
99 disp(['Iterations: ',num2str(iter)])
100 disp(['CPU time: ',num2str(time),'s.'])
101 %fprintf('CPU time = %.8g \n',time )
102 disp(' ')
103
104 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
105
106 legend('GMRES', 'BI-CG','IDR(4)','IDR(s)','PIDR(s)');
107 hold off;
```

Appendix D

Other Matlab files

D.1 Aanroep_methodes.m

```
clear all
close all
clc

% defaults
s     = 4
n     = 200;
maxit = 1000;
tol   = 10^-8;

% System defaults
A = gallery('tridiag',n,-1,2,-1);

b  = A * ones(n,1);      % RHS of the linear system
x0 = zeros(n,1);         % initial guess for the linear system
v1 = rand(n,1);          % starting vector for eigenvaleu problem

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%           Execution of symmetric methods (Lanczos type)             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[Vec,Val,resvec,iter] = Lanczos(A,v1,maxit);
[x,resvec,iter]       = Lanczos_system(A,b,x0,maxit);
[x,resvec,iter]       = CG(A,b,x0,maxit);
[x,resvec,iter]       = CR(A,b,x0,maxit);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%            Execution of general methods (Arnoldi type)              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[Vec,Val,resvec,iter] = Arnoldi(A,v1,maxit);
[V,x,resvec,iter]     = FOM(A,b,x0,maxit);
[V,x,resvec,iter]     = GMRES(A,b,x0,maxit);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%           Execution of general methods (Bi-Lanczos type)            %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[Vec,Val,resvec,iter] = Bi_Lanczos(A,v1,maxit)
[x,resvec,iter]       = Bi_Lanczos_system(A,b,x,maxit)
[x,resvec,iter]       = Bi_CG(A,b,x,maxit)
[x,resvec,iter]       = Bi_CR(A,b,x,maxit)
```

D.2 Arnoldi_Basis.m

```
function V = Arnoldi_Basis(A,V,m)

[n,s] = size(V);

for j=1:m

    w = A * V(:,end);

    % orthogonalising against previous vectors
    for i = 1:s
        H = w' * V(:,i);
        w = w - H*V(:,i);
    end

    %normalisation
    V(:,s+1) = w / norm(w);
    s=s+1;
end
```

D.3 Arnoldi_Basis_Block.m

```
function V = Arnoldi_Basis_Block(A,V,m,s)

for j=1:m

    Z = A * V(:,end-s+1:end);

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

    % Compute QR decomposition
    [Q, ~] = qr(Z,0);
    V = [V,Q];

end
```

D.4 CDE.m

```
function A = CDE(m,b)

eps = 1;
beta(1) = b;
beta(2) = b;
beta(3) = b;
r = 0;

% Generate matrix
h = 1/(m+1);
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;
A = full(A);
```

D.5 sorteig.m

```
function [EV2,EW2] = sorteig(H,B)
% 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

if nargin == 1
[EV EW] = eig(H);

EW2 = diag(sort(diag(EW),'ascend'));
[c, ind]=sort(diag(EW),'ascend');
EV2=EV(:,ind);
end

if nargin == 2
[EV EW] = eig(H,B);

EW2 = diag(sort(diag(EW),'ascend'));
[c, ind]=sort(diag(EW),'ascend');
EV2=EV(:,ind);
end
```