BACHELOR

Implementing and testing a new variant of IDRstab

Senders, Mischa J.

Award date:
2019

Eindhoven University of Technology

– Implementing and testing a new variant of IDRstab –

Eindhoven, July 11, 2019

Supervisor:	dr.ir. J. (Jan) van Dijk
Direct Supervisor:	ir. C.E.M. (Chris) Schoutrop
M.J.Senders	0945467

Abstract

Krylov subspace methods are a common way to approximate large linear systems. The aim of this report is to build a new version of the Krylov algorithm IDRstab in MATLAB and test its performance for various test cases. It becomes clear that the algorithm is not fully replicated because the SHERMAN5 matrix, which has been solved using this algorithm before, could not be solved by this specific implementation. The impact of the two free parameters of IDRstab L and S was analysed as well as the importance of the condition number and eigenvalues of the matrix A. The IDRstab implementation could not keep up with the convergence speed of alternative Krylov algorithms such as Bicgstab for any of the test cases discussed in this report. This is probably because of errors remaining in the algorithm and the lack of optimisation which causes the algorithm to do a lot of unnecessary work. The algorithm did however come quite close to a few alternatives for some test , so after optimisation it is plausible that it outperforms other Krylov algorithms in certain situations.

Contents

1 Introduction

Linear equations show up almost everywhere in science and economics. Examples are Leontiefs input-output model, which is a model from macroeconomics describing the relationship between supply and demand [1] or the state of an electron from quantum mechanics [2]. Linear systems are also important in many fields of mathematics such as multi-variable calculus differential equations [3].

Differential equations of dynamical systems from fields such as economics and physics can be approximated numerically by solving the system on a discrete number of points sometimes called a mesh. The more points you use, the more accurate your solution will be. These numerical solutions can be put in an equation of the form $\mathbf{A}\vec{x} = \vec{b}$ where $\vec{x}$ is unknown. Mathematically $\vec{x}$ can be found in several ways. For example by multiplying both sides from the left by $\mathbf{A}^{-1}$. When the size of $\mathbf{A}$ becomes very large, finding $\mathbf{A}^{-1}$ becomes impractical. A better alternative is to invert a matrix using LU decomposition which factors the matrix in a lower and upper triangular matrix [4]. For large matrices this method is still very expensive, however. This is why over the years many algorithms have been developed to approximate $\vec{x}$ iteratively.

A popular family of approximation techniques is called Krylov subspace methods. Krylov methods have become very successful and are perhaps the most important tool nowadays to solve the equation $\mathbf{A}\vec{x} = \vec{b}$ [5]. One of these Krylov methods named IDR (induced dimension reduction) has recently gotten new attention with the introduction of IDRstab [6] [7]. In this report we implement a relatively new variant of IDRstab, algorithm 1 in reference [8], in MATLAB and compare its performance to other popular Krylov algorithms for several test cases. This new variant on IDRstab is very promising since it has already solved many problems much faster than other algorithms. The test cases specified in section 3.1 are divided into dense matrices (section 3.2) and sparse matrices (section 3.3). Furthermore the convergence speed of various Krylov algorithms is compared and we discuss briefly the importance of the condition number and eigenvalues on the convergence.

In section 2 we provide the reader with some theoretical background without fully deriving any of the Krylov algorithms. For full derivations we refer the reader to the sources provided with the particular topic. Instead we attempt to give the reader a general understanding about what is actually going on by describing the working principles of various different Krylov algorithms. Test cases can be found in sections 3.2 and 3.3. A detailed reflection of the test cases and improvements to the algorithm can be found in section 4.

2 Theory

Solving the system $\mathbf{A}\vec{x} = \vec{b}$ for $\vec{x}$ where $\mathbf{A}$ is a matrix and $\vec{x}$ and $\vec{b}$ are vectors can be done in exact arithmetic in many ways. For example by multiplying both sides of the equation by $\mathbf{A}^{-1}$ or by using LU decomposition. These strategies are no longer viable, however, when the matrix $\mathbf{A}$ becomes very large. The computational complexity of matrix inversion is in the best case for dense matrices in the order $O(n^{2.373})$ [9]. This is called the Copperschmith-Winograd barrier. This becomes very time consuming for matrices of order $O(10^6)$ and larger. Such systems are therefore often approximated by iterative methods.

A common and rather successful way of approximating the solution of such systems is by using Krylov subspaces. Reference [10] contains a neat example which was originally published in reference [11] showing how much faster Krylov methods can be compared to direct methods. The example contains a system of equations with approximately five billion unknowns. According to their estimations, it would take a computer over half a million years to find the solution to this system of equations whilst the Krylov method conjugate gradients could do it in 575 seconds. The relative difference is in the order $O(10^{10})$ seconds. This example is obviously rather artificial but the point still holds: Krylov methods can be astronomically faster than direct methods when used in the right situation.

2.1 Machine precision and rounding errors

When we use computers to do arithmetic, we should be careful of rounding errors. When a very small number gets added to a very large number the computer will round the very small number off to 0. We will illustrate this by showing how a computer would tackle a simple Gram-Schmidt orthogonalisation example. Gram-Schmidt orthogonalisation is also an important concept in Krylov analysis. As explained in reference [12], Gram-Schmidt orthogonalisation works by removing the projections of every next vector from the first vector. This leads to an orthogonal set of vectors. Equation 2.1 shows the general procedure to find the Gram-Schmidt vectors $\vec{u_i}$ out of a random set of n vectors $\vec{v_i}$. An orthonormal set can easily be acquired by normalising all the orthogonal Gram-Schmidt vectors afterwards.

$$\vec{u_i} = \vec{v_i} - \sum_{i=2}^{n} \frac{\vec{v_i} \cdot \vec{u_i}}{\vec{u_i} \cdot \vec{u_i}} \vec{u_i} \tag{2.1}$$

It is interesting what happens when rounding errors occur in these vectors. You might expect that for small rounding errors the resulting set of vectors will also slightly deviate from orthonormality. Unfortunately, this is not what we observe. This is illustrated in the following example:

Suppose we have an ϵ such that $1 + \epsilon = 1$ in finite precision arithmetic. Suppose we want to make three orthonormal basis vectors $\vec{u_1}, \vec{u_2}$ and $\vec{u_3}$ from three vectors:

$$\vec{v_1} = \begin{bmatrix} 1 \\ \epsilon \\ 0 \\ 0 \end{bmatrix}, \vec{v_2} = \begin{bmatrix} 1 \\ 0 \\ \epsilon \\ 0 \end{bmatrix}, \vec{v_3} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \epsilon \end{bmatrix}.$$

This means that the first basisvector becomes:

$$\vec{u_1} = \frac{\vec{v_1}}{\|\vec{v_1}\|} = \frac{1}{\sqrt{1+\epsilon^2}} \begin{bmatrix} 1 \\ \epsilon \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ \epsilon \\ 0 \\ 0 \end{bmatrix}. \tag{2.2}$$

This is because the computer cannot distinguish $1 + \epsilon = 1$ and 1. The second basisvector becomes:

$$\vec{u_2} = \vec{v_2} - (\vec{v_2} \cdot \vec{v1})\vec{u_1} = \begin{bmatrix} 0 \\ -\epsilon \\ \epsilon \\ 0 \end{bmatrix}. \tag{2.3}$$

Normalising $\vec{u_2}$ gives:

$$\vec{u_2} = \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}. \tag{2.4}$$

Analogously:

$$\vec{u_3} = \begin{bmatrix} 0 \\ -\frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.5}$$

Since our vectors are supposed to be orthonormal, we expect the dot products of the computed vectors to be 0 or in the order ϵ. When we now compute the dot products we find that the dot products of $\vec{u_1}$ with $\vec{u_2}$ and $\vec{u_3}$ are in the order of ϵ and therefore nearly zero as expected. However, the dot product of $\vec{u_2}$ with $\vec{u_3}$ gives us $\frac{1}{2}$ which means that these vectors are clearly not orthogonal. This shows that even tiny numerical offsets can lead to completely ill-defined systems. When the projection of vector $\vec{v}$ is slightly off due to rounding errors, the direction is completely off because this new vector is very small compared to the first vector. Since these projections are very tiny vectors, minor absolute errors lead to very large relative errors which are scaled up to completely wrong unit vectors in the normalisation step. It is clear that we have to be smarter than simple Gram-Schmidt orthogonalisation. An example of a smarter method is called Modified Gram-Schmidt procedure and is further discussed in section 2.3.

2.2 Condition number

The condition number of a matrix is a measure of how well defined a matrix is [12]. The order of the condition number is roughly the number of significant digits lost in the answer. A large condition number therefore corresponds to an ill-defined matrix leading to large numerical errors. This means that we can expect to be able to solve a matrix with a very low condition number very accurately, since we do not lose many significant digits, whilst solving a matrix with a very high condition number can be very inaccurate. Two important factors in the condition number are the linear dependence and the scaling of the columns. Very linearly independent columns of similar scale will lead to a very low condition number. On the contrary, very linearly dependent columns of different scales will lead to a very high condition number. The condition number of a matrix $\mathbf{A}$ is defined as [12]:

$$k(\mathbf{A}) \stackrel{\text{def}}{=} \|\mathbf{A}\| \left\|\mathbf{A}^{-1}\right\| \tag{2.6}$$

.

Where the norm of the matrix is defined as the sum of the elements of the largest row. The maximum error in $\vec{x}$ can be determined as a function of the condition number and the maximum error in $\vec{b}$ according to equation 2.7. It is clear that when the condition number is relatively high, a small relative error in $\vec{b}$ can cause a large relative error in $\vec{x}$.

$$\frac{\|\delta\vec{x}\|}{\|\vec{x}\|} \leq k(\mathbf{A}) \frac{\left\|\delta\vec{b}\right\|}{\left\|\vec{b}\right\|} \tag{2.7}$$

Krylov methods are usually only competitive for sparse matrices but for dense matrices Krylov can still be competitive if the condition number is smaller than n^2 where n is the size of the matrix [10]. A common way to generate matrices with a good condition number is by adding a multiple of the identity matrix to a random matrix. This will make the columns generally more linearly independent and this will help stabilising the projections that most iterative algorithms use. The identity matrix has a condition number of 1 while for very bad matrices the condition number can go up towards infinity.

Just like most things in numerical analysis, the condition number can only give an indication of the precision and speed of convergence. Relatively "bad" matrices can sometimes converge surprisingly well whilst "good" matrices can converge rather poorly [5].

2.3 Krylov subspaces

The columns of a non-singular square matrix of dimension n span a n-dimensional space. The solution to the initial equation $\mathbf{A}\vec{x} = \vec{b}$ must lie in this vector space. The main idea of the Krylov subspace method is to solve for the projection of $\vec{x}$ in a subspace of dimension $m < n$. This is analogous to approximating the shape of a 3-dimensional object by looking at its shadow on a 2-dimensional plane.

A Krylov subspace is formed by repeatedly multiplying a vector by the same matrix. Every next vector is guaranteed to introduce a new dimension (as long as $m < n$) by a process called the power iteration method or power method [7]. The Krylov subspace of matrix $\mathbf{A}$ and $\vec{v}$ is defined as:

$$\kappa(\mathbf{A}, \vec{v}) \stackrel{\text{def}}{=} \text{span}\{\mathbf{A}^{m-1}\vec{v}, ..., \mathbf{A}\vec{v}, \vec{v}\} \tag{2.8}$$

The issue with this method is that every next vector will become increasingly parallel to the dominant eigenvector of the matrix such that after a few iterations, the computer can no longer distinguish the newly introduced dimensions. Unfortunately, defining all vectors first and orthonormalising afterwards does not work, since you will be orthonormalising an already ill-defined system [13]. A common way of dealing with this issue is by using the Arnoldi orthogonalisation process, which orthonormalises every new vector to the set of all previous vectors after every iteration by using the modified gram-Schmidt method. A variant of the Arnoldi algorithm can be found in appendix C.

2.4 Known algorithms

There are four fundamental Krylov subspace approaches [10]. Three of those are relevant for our purposes. The first is called the Ritz-Galerkin approach. In the Ritz-Galerkin approach every next residual is orthogonal to a growing subspace. This implies that the components of the residual in the current subspace are zero such that the components of $\vec{x}$ which correspond to the current basis are exactly correct. This method gives rise to algorithms such as Bicg and Bicgstab.

The second approach is the minimum norm residual approach. This method is based on the least squares approximation. A basis is chosen such that $\left\|\vec{b} - \mathbf{A}\vec{x}\right\|$ is minimum in the current Krylov subspace. This method gives rise to algorithms such as GMRES.

The third is called the Petrov-Galerkin approach. The Petrov-Galerkin approach forces the residual in a specific subspace of which the dimension decreases. You can also view it as the residual being forced to be orthogonal to a subspace of increasing size. Both views are mathematically equivalent. The Petrov-Galerkin approach gives rise to algorithms such as IDR.

The fourth method is called the minimum norm error approach. This method minimises $\|\vec{x_k} - \vec{x}\|$ but this method is not used in this report. The following sections describe some of the most common Krylov methods and their differences.

2.4.1 Local minimum residual (LMR)

One of the simplest Krylov subspace methods is called the local minimum residual approach. It uses the least squares method to minimise the local residual by tuning the parameter α. This implies that every iteration, alpha is changed such that the residual is minimum. A full MATLAB version of the local minimum residual approach can be found in appendix D. The basic steps of LMR are shown below:

while $\|\vec{r}\| \geq tol$ **do**
  $\alpha = (\mathbf{A}\vec{r} \cdot \vec{r})/(\mathbf{A}\vec{r} \cdot \mathbf{A}\vec{r})$;
  $\vec{x} = \vec{x} + \alpha\vec{r}$;
  $\vec{r} = (\mathbf{I} - \alpha\mathbf{A})\vec{r}$;
end

In the first step the α is found that minimises the residual $\vec{r}$. Then $\vec{x}$ is updated according to this new α. Finally $\vec{r}$ is updated according to this new α. This process is repeated until the required tolerance is reached.

2.4.2 GMRES

GMRES calculates the minimum residual in a growing Krylov subspace. Every iteration a new dimension is introduced by multiplying the previous vector by $\mathbf{A}$. This ensures that the residual can by definition only decrease such that GMRES is guaranteed to reach the exact solution when all dimensions are checked. The drawback of this method is that all previous vectors need to be stored for every iteration. At every iteration the new vector needs to be orthogonalised to a growing number of previous vectors such that every new iteration is more costly and requires more storage than its predecessors. GMRES is both very costly and requires lots of memory. GMRES is known to get stuck on so called plateaus because no attempts are made to look in "clever directions"[14]. This means that the residual can remain constant for many iterations if GMRES looks in very inefficient directions. The costs and amount of memory that GMRES requires can be heavily decreased by doing restarted GMRES which restarts building a set of vectors after a desired number of iterations. This version is much faster than standard GMRES but no longer guarantees convergence [15].

2.4.3 Bicgstab

Bicgstab is a combination of two techniques: CGS and LMR [5]. "CGS" stands for conjugate gradient squared and works for symmetric problems [12]. Conjugate gradient and conjugate gradient squared are improved versions of gradient descent. In gradient descent, at every iteration a step is made in the opposite direction of the gradient such that the gradient decreases until it becomes 0. This next point is then taken as the new trial solution [16]. A common problem with this method is that it tends to bypass the solution at every iteration leading to a kind of zigzag pattern. The conjugate gradient method is somewhat better than gradient descent but still experiences unexpected peaks in the residual against time graph [15]. "stab" stands for stabilising since stabilising polynomials are used at every iteration. The stabilising polynomials step is similar to a LMR step which has already been discussed in section 2.4.1. This attempts to smooth the convergence which is especially effective in case CGS experiences irregular convergence. Because CGS squares the polynomials, rounding errors tend to build up. In that case the residuals tend to become very inaccurate. The LMR step is an effective way to avoid this problem.

2.4.4 IDR(s)

Induced dimension reduction (IDR(s)) is a different type of Krylov algorithm which generates residuals that are forced into a subspaces g_k which decreases over time. Initially, the obvious subspace of choice is the entire Krylov subspace but this space is reduced during the algorithm. Bicgstab and IDR are closely related. IDR(1) for example is mathematically identical to Bi-cgstab [7].

2.5 IDR(S)Stab(L)

IDR(S)Stab(L), to which we will from now on refer to as "IDRstab", is a combination of the IDR(S) algorithm explained in section 2.4.4 and the stabilising polynomials explained in section 2.4.3. This algorithm is particularly interesting because it has been shown to converge requiring less matrix-vector multiplications than the other algorithms in many situations. This is illustrated in figure 2.1 which shows the convergence of IDR(4)Stab(4) for the SHERMAN5 matrix compared to other algorithms. Note that even though GMRES converges a lot faster, it might not be better because it requires a lot more memory. Reference [8] introduced a new version of IDRstab. The difference between the new version of IDRstab and the original one is in the calculation of the residual. Most Krylov algorithms work by estimating the residual instead of calculating the residual explicitly because this is a rather expensive calculation. The original IDRstab tends to acquire a relatively large residual gap meaning that its estimation of the residual does not accurately match the true residual. The new version of IDRstab occasionally decides to explicitly calculate the true residual making it more effective in moving forward. Since reference [8] claims that their version of IDRStab is faster than the original, it is definitely a method that deserves attention.

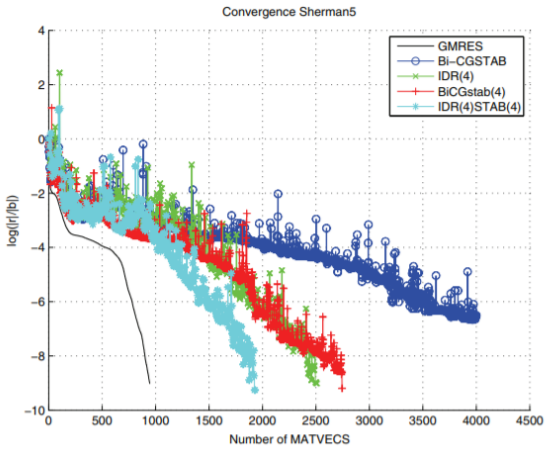

Figure 2.1: The relative residual of IDRstab for the SHERMAN5 matrix compared to other algorithms as a function of the number of matrix-vector multiplications [7].

2.5.1 Algorithm description

The specific algorithm is "algorithm 1" in reference [8]. Even though this algorithm is expected to yield a better performance, the implementation did not turn out to be straightforward. This will be discussed in detail in section 4. A comprehensive version of our interpretation of the algorithm can be found in appendix A where the code is rewritten such that it can easily be translated to any programming language. A MATLAB version of our interpretation can be found in appendix B.

2.5.2 The significance of L and S

The parameters "L" and "S" determine the expensiveness of each iteration. Higher values of L and S make single iterations more costly but in general also more effective. The cost of these algorithms is measured by the number of "AXPYs" ($\alpha * \vec{x} + \vec{y}$) and "MVs" ($A\vec{x}$). L determines how often the IDR step is executed before a LMR step is taken and S determines how many vectors are constructed in the IDR step to choose a new search direction. Tests for several values of L and S can be found in section 3.2.2. The amount of AXPYs and MVs per cycle can be found in table 1. It can be seen that the new algorithm does a couple of extra MVs but saves on the AXPYs making it more efficient overall.

Table 1: The amount of MVs and AXPYs used by the original variant of IDRStab and for the new version [8].

Solver	MVs	AXPYs
original IDRstab	$L(S+1)$	$\frac{1}{2}LS(L+1)(S+1)+L(S^2+3S+2)$
New IDRstab	$L(S+1)+L+1$	$\frac{1}{2}LS(L+1)(S+1)+\frac{3}{2}L+S+\frac{1}{2}$

2.5.3 Example: A specific linear system

An interesting way to get more insight in the algorithm is to work through its steps for a 2×2 matrix. Obviously the algorithm was not designed to solve such small systems. It is still interesting, however, to investigate how the algorithm deals with such a small system and which concrete steps it performs to come to an answer. Let us define:

$\mathbf{A} = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$

$\vec{b} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$ and $\vec{x_0} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ is our initial guess vector.

The exact solution for such a small system is easily computed: $\vec{x} = \begin{pmatrix} \frac{4}{3} \\ -\frac{2}{3} \end{pmatrix}$.

It must be noted that the auxiliary matrix R_0 is chosen to be the identity matrix.

As usual the residual is calculated by: $\vec{r_0} = \vec{b} - \mathbf{A}\vec{x} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$.

In the next step the algorithm makes an orthonormal basis using the Arnoldi algorithm. This gives the orthonormal basis vectors: $\vec{u_1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\vec{u_2} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. These are stored in the columns of U_0. This corresponds to lines 7 until 16 of the comprehensive description of the algorithm in appendix A. In the next step the projections of the columns of $\mathbf{A}$ on this new basis U_0 are stored in the matrix $\boldsymbol{\sigma}$. Since U_0 is the identity matrix these projections return the matrix $\mathbf{A}$. The algorithm then continues to calculate $\boldsymbol{\sigma}$ which corresponds to $\mathbf{A}^{-1}$ which is therefore a trivial solution $\mathbf{A}^{-1}$ is what we needed to determine in the first place. It turns out that for a very simple system, the algorithm simply computes $\mathbf{A}^{-1}$ with a detour.

2.6 Eigenvalues

The eigenvalue configuration of the matrix is a common property that influences the convergence of Krylov algorithms. Algorithms with stabilising polynomials of degree one such as IDR(S) and Bicgstab(1) often have issues solving systems with large non-real eigenvalues close to the imaginary axis [8]. In these situations Bicgstab(L) and IDR(s)Stab(L) are expected to outperform algorithms which do not use higher order polynomials, as long as L is chosen larger than one. Bicgstab is expected to perform well when the eigenvalues are clustered [15] [10].

The convergence behaviour of GMRES can also be expressed in terms of the eigenvalues as discussed in reference [17]. To test the influence of eigenvalues on convergence we will use a Toeplitz matrix. Toeplitz matrices are matrices with certain diagonals filled with same numbers. tri-diagonal matrices are an example of a Toeplitz matrix if they have a fixed value on all three diagonals. The general tri-diagonal Toeplitz matrix is shown in equation 2.18.

Toeplitz matrices are very useful thanks to the distribution of their eigenvalues. They can be calculated using

$\lambda_k = a + 2 * \sqrt{b * c} * \cos \frac{k*\pi}{n+1}$ [18].

By choosing b and c small and a large imaginary we can accomplish the large non-real eigenvalues close to the imaginary axis as discussed before. This has been tested in section 3.3.2.

2.7 Application to physical problems

As explained before Krylov methods are mainly used for sparse matrices. In this section we will introduce an example in which it becomes clear how a sparse matrix would show up in practice. Let us start with the differential equation:

$$\frac{d}{dx}(U\phi - \epsilon\frac{d\phi}{dx}) = S(x) \tag{2.9}$$

in which $S_i = S(x_i)$ and ϵ and U represent advection and diffusion. The exact significance of this equation is not relevant for our purposes. It is interesting, however, how a differential equation such as this one could be approximated numerically and why a linear system of equations would actually show up in the first place.

If U and ϵ are constant, equation 2.9 simplifies to:

$$U\frac{d\phi}{dx} - \epsilon\frac{d^2\phi}{dx^2} = S(x). \tag{2.10}$$

Now two approximations are made which allow us to solve this system numerically. These are:

$$\frac{d\phi}{dx} \approx \frac{\phi_{i+1} - \phi_{i-1}}{i\Delta x} \tag{2.11}$$

and

$$\frac{d^2\phi}{dx^2} = \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{(\Delta x)^2} \tag{2.12}$$

These approximations can be used in equation 2.10 giving:

$$\frac{1}{2}U(\phi_{i+1} - \phi_{i-1}) - \frac{\epsilon}{\Delta x}(\phi_{i+1} - 2\phi_i + \phi_{i-1}) = \Delta x S_i \tag{2.13}$$

Factoring gives:

$$(\frac{1}{2}U - \frac{\epsilon}{\Delta x})\phi_{i+1} + (\frac{2\epsilon}{\Delta x})\phi_i + (-\frac{1}{2}U - \frac{\epsilon}{\Delta x})\phi_{i-1} = \Delta x S_i \tag{2.14}$$

This equation should now be solved n times for n times or positions. This implies that i goes from 1 to n. This can be rewritten as a linear system. The matrix consists of three recurring terms. a, b and c are defined as follows:

$$a \stackrel{\text{def}}{=} \frac{2\epsilon}{\Delta x} \tag{2.15}$$

$$b \stackrel{\text{def}}{=} \frac{1}{2}U - \frac{\epsilon}{\Delta x} \tag{2.16}$$

$$c \stackrel{\text{def}}{=} -\frac{1}{2}U - \frac{\epsilon}{\Delta x} \tag{2.17}$$

In terms of a, b and c the set of linear equations would look like this:

$$\begin{bmatrix} a & 0 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ b & a & c & \ddots & & & & \vdots \\ 0 & b & & & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & & & c & 0 \\ \vdots & & & & \ddots & b & a & c \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & 0 & a \end{bmatrix} * \begin{bmatrix} \phi_1 \\ \vdots \\ \phi_{i-2} \\ \phi_{i-1} \\ \phi_i \\ \phi_{i+1} \\ \phi_{i+2} \\ \vdots \\ \phi_n \end{bmatrix} = \begin{bmatrix} S_1 \\ \vdots \\ S_{i-2} \\ S_{i-1} \\ S_i \\ S_{i+1} \\ S_{i+2} \\ \vdots \\ S_n \end{bmatrix} \tag{2.18}$$

As shown in equation 2.18 this leads to a set of linear equations involving a sparse matrix which can be solved using Krylov methods. It has to be noted that this specific tri-diagonal matrix can also be solved very easily by the Thomas algorithm [19]. Krylov methods are therefore not needed to solve this particular problem. The above is merely an example of how sparse matrices could arise from problems in physics. More complicated matrices with similar structure would arise when more dimensions are introduced or when the approximations such as 2.11 become more sophisticated. An example of discretisation of a problem from fluid dynamics can be found in reference [20]. Finally it has to be noted that a "c" seems to be missing on the first row and a "b" seems to be missing on the last row. This is done to satisfy the boundary conditions. Since equation 2.9 is a second order differential equation, the general solution has two free parameters which can be determined by the boundary conditions of the specific problem.

3 Results

3.1 Test case specification

In this section special test cases are discussed. This ensures the reproducibility of the results. The tests are done using matlab2019 running the code in appendix B. Rng(1) is used in all tests. Random elements are defined in the order that can be found in appendix A. The initial guess vector $\vec{x}$ is the zero vector. The tolerance is 10^{-12}. The parameters are $L = 2$ and $S = 4$ with a maximum of 100 iterations. A single iteration is a full cycle through the while loop. All of the above applies to all test cases unless specifically stated otherwise. The term "residual" always refers to relative residual which is the norm of the absolute residual normalised to the norm of vector $\vec{b}$ as shown in equation 3.1.

$$r \stackrel{\text{def}}{=} \frac{(\left\|\vec{b} - \mathbf{A}\vec{x}\right\|)}{\left\|\vec{b}\right\|} \tag{3.1}$$

Furthermore algorithm 1 from reference [8] is slightly altered. $\boldsymbol{\sigma}$ tends to become badly conditioned in the given algorithm so this was avoided by multiplying it by a preconditioning matrix $\mathbf{P}$ containing the inverse of the absolute values of the columns of $\boldsymbol{\sigma}$ on its diagonal. This rescales the columns of $\boldsymbol{\sigma}$ such that the condition number decreases as explained in section 2.2. $\vec{\alpha}$ was calculated with the altered $\boldsymbol{\sigma}$ and thereafter multiplied by $\mathbf{P}$ again cancelling the overall effect.

3.2 Results dense matrices

Krylov methods are generally applied to equations with sparse matrices. They can however be used to solve dense matrix systems as well. In this section the performance of the algorithm for dense matrices will be discussed. In section 3.3, the performance of the algorithm for sparse matrices will be discussed.

3.2.1 Condition number

In this section the influence of the condition number of the matrix on solutions will be analysed. First we check how the condition number changes as a function of the diagonal dominance and as a function of the size of the matrix. The matrices will be changed using equation 3.2 as a function of ξ.

$$\mathbf{A} = \mathbf{A} + \xi\mathbf{I} \tag{3.2}$$

All test cases consist of a 10×10 matrix $\mathbf{A}$ and 10×1 vectors $\vec{b}$ with random elements (according to rng(1)) and $\vec{x}$ the zero vector. All random elements are in the open interval (0,1). The matrix $\mathbf{A}$ is constructed by the operation $\mathbf{A} = (\mathbf{A} + \mathbf{A}')/2$ where $\mathbf{A}'$ is defined as the transpose of $\mathbf{A}$. This operation makes the matrix $\mathbf{A}$ symmetric. As explained in section 2.4.3, some algorithms such as CGS only work for symmetric matrices. However, IDRstab can also solve non-symmetric matrices [8]. Since the algorithm did not work as expected we decided to use matrices that are generally considered "easy" in our test cases. Figure 3.1 shows how the condition number changes as a function of ξ. It is clear that the condition number behaves rather chaotically until ξ is around one. After that the condition number decreases rapidly as a function of ξ.

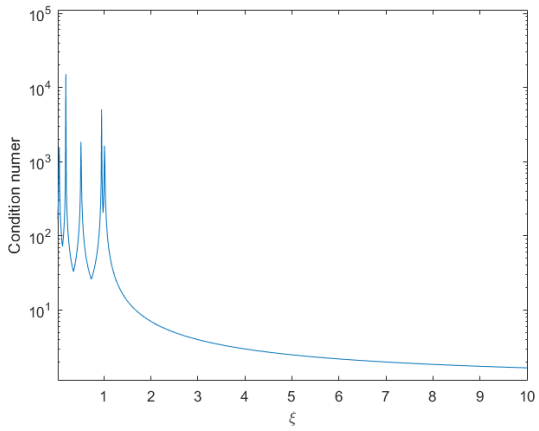

Figure 3.1: Change of the condition number as a function of the ξ

In figure 3.2 it is clear that the algorithm cannot solve the system at all if the diagonal does not receive a slight addition. The residual seems to slowly go towards infinity. The condition number of this matrix is 104.9. This is why a multiple of the identity matrix is added to matrix $\mathbf{A}$ as shown in equation 3.2.

In figure 3.3 we can see that the residual already experiences multiple spikes downwards for the case $\xi = 0.5$. The residual does not quite converge to 10^{-12} within 100 iterations though. The condition number of this matrix is 403.3. In figure 3.4 and in figure 3.5 it is clear that the residual converges to 10^{-12} faster and faster for $\xi = 1.2$ and $\xi = 10$ respectively. Their corresponding condition numbers are 33.02 and 1.661 which are much lower as could already be seen in figure 3.1.

To get more insight into the dependence of the diagonal dominance, the number of iterations until convergence is shown for a matrix of increasing ξ in figure 3.6. It is clear that if $\xi = 1$ the algorithm does not converge within the first 100 iterations. Two rather remarkable observations can be made. The first is that the decrease of necessary iterations happens abruptly. For $\xi = 0.7$ the algorithm does not converge

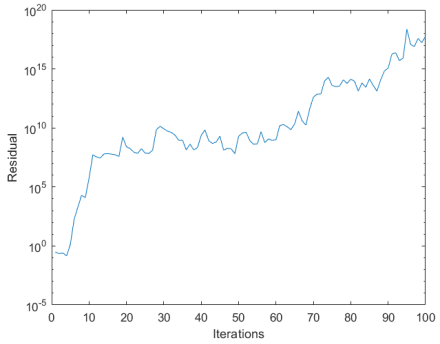

Figure 3.2: Residual during the first 100 iterations for $\xi = 0$.

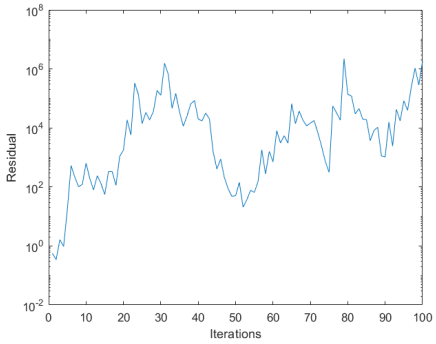

Figure 3.3: Residual during the first 100 iterations for $\xi = 0.5$.

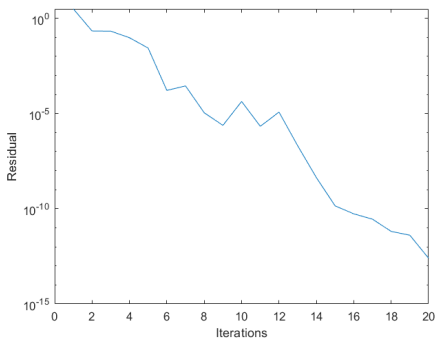

Figure 3.4: Residual during the first 100 iterations for $\xi = 1.2$.

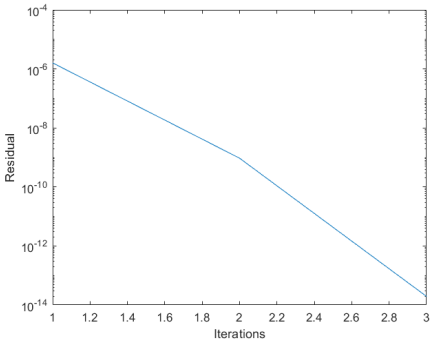

Figure 3.5: Residual during the first 100 iterations for $\xi = 10$.

within 100 iterations while for $\xi = 1.4$ only about 10 iterations are needed. This seems to correlate nicely with the condition number. As was shown in figure 3.1, $\xi = 1$ is the critical value at which both the chaotic behaviour of the condition number seems to stop because the condition number rapidly drops from this point and also the number of iterations until convergence rapidly drops. This implies a very strong dependence between the condition number of the matrix and the convergence speed. Furthermore, it is interesting that the graph is very smooth and does not seem to contain any local minima. Perhaps this is because the step size of 0.1 is too large.

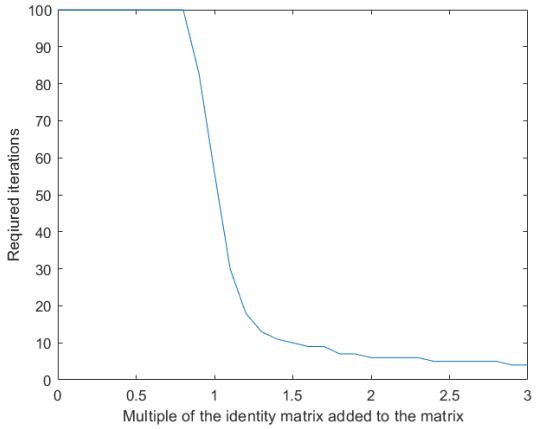

Figure 3.6: Number of iterations needed to achieve the tolerance for an increasing diagonal dominant random matrix as a function of ξ.

To understand how the required diagonal dominance changes for larger matrices, we looked for the required diagonal dominance for matrices of different sizes such that the algorithm converges within 10 iterations. The checking was done in intervals of 0.1 so it is mainly intended to get a sense of the sensitivity rather than to understand the exact process quantitatively. Table 2 shows the required diagonal dominance for different matrix sizes. These points are plotted in figure 3.7. The plot shows that the required ξ increases for larger matrices, but no further research was done on this topic. The table suggests that the algorithm works well when the condition number of the matrix is in the order of 10^2. We would expect larger matrices to tolerate higher condition numbers as explained in section 2.2, because Krylov methods can be competitive with direct methods when the condition number is smaller than n^2. It is possible, however, that 10 iterations is not enough to observe this result. Krylov algorithms can still be competitive with direct methods since direct methods also take a lot longer than the equivalent of 10 iterations for matrices of this size. Perhaps the algorithm simply needs more iterations to challenge direct methods. Unfortunately no further experiments were done on this subject.

Table 2: Required diagonal dominance for a matrix with certain dimensions such that the algorithm converges within 10 iterations.

Matrix dimension	Addition on diagonal	Condition number
10 x 10	1.2	33.02
50 x 50	3.4	48.49
100 x 100	4.8	71.87
200 x 200	7.8	49.73
500 x 500	13	66.11
1000 x 1000	17.2	119.4

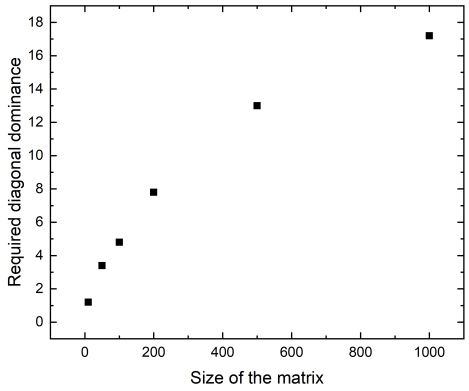

Figure 3.7: Number of iterations needed to achieve the tolerance for an increasing value of ξ.

3.2.2 The significance of L and S

As discussed in section 2.5.2, the choice of L and S strongly influences the convergence of the algorithm. For testing we will use the matrix `rand(1000)+17.2*eye(10)` from section 3.2.1 and calculate the convergence for varying L and S. The amount of iterations for varying L and S are illustrated in table 3.

Table 3: Number of iterations to converge to a tolerance of 10^{-12} for varying L and S.

s L>	1	2	3	4	5	6
1	152	27	25	15	11	9
2	33	19	13	11	9	7
3	575	19	10	8	6	5
4	32	11	8	9	6	6

As expected, higher L and S allow the algorithm to converge in fewer iterations. However, that does not mean that these are preferable choices because the amount of work for the algorithm increases significantly too. To see which parameters are optimal, the same measurement is done to determine the running times. These times as shown in table 4 are the average of 100 measurements. It can be seen that a lot of combinations take around 0.4 seconds per run. The combination $L = 1$ and $S = 3$ proves really bad for this matrix as can be seen from the remarkably long times in both tables. The reason for this is unclear. It is usually the result of matrices becoming nearly singular but we could not find proof for this.

Table 4: Time in seconds required to reach a tolerance of 10^{-12} for varying L and S averaged over 100 measurements.

s L>	1	2	3	4	5	6
1	2.84	0.90	0.42	0.39	0.37	0.37
2	0.48	0.90	0.40	0.42	0.42	0.41
3	4.71	0.51	0.40	0.39	0.40	0.40
4	1.16	0.39	0.40	0.49	0.46	0.50

3.3 Results sparse matrices

As we have seen in the previous section, the algorithm cannot solve dense linear systems equations without adjusting the matrix by adding a multiple of the identity matrix to let the algorithm converge at all. In this section we will investigate the performance in the case of sparse matrices.

3.3.1 SHERMAN matrices

A common test case for Krylov subspace methods is the SHERMAN5 matrix. The matrix can be found in reference [21]. The convergence of this matrix in [8] can be seen in figure 3.8.

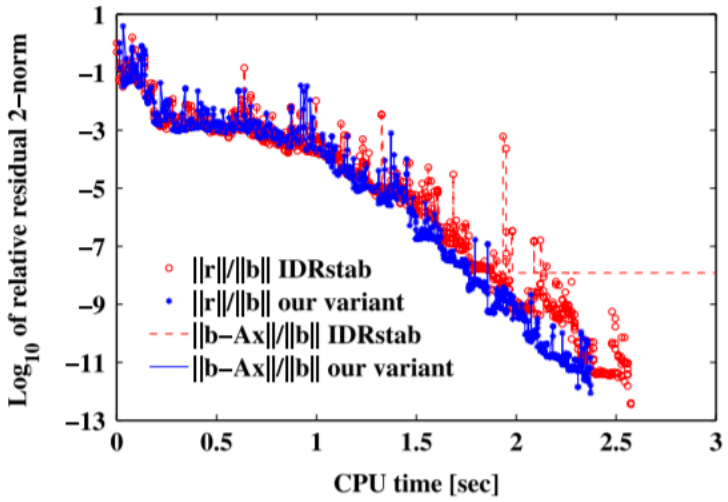

Figure 3.8: Residual over time for the SHERMAN5 matrix [8].

Unfortunately the MATLAB implementation in appendix B cannot solve the SHERMAN5 matrix vector equation. A graph of the residual as a function of iterations can be found for $L = 2$ and $S = 4$ in figure 3.9. It is clear that the residual eventually diverges. The same happens for different choices of L and S. This clearly indicates that the algorithm is still not an exact replica of the one in [8]. Potential reasons for this difference are discussed in the section 4. SHERMAN1, SHERMAN2, SHERMAN3 and SHERMAN4 can also be found in reference [21]. Unfortunately none of these could be solved within 1000 iterations either. All tests were done using the corresponding $\vec{b}$.

3.3.2 Eigenvalues

As explained in section 2.6, Toeplitz matrices are very suitable for analysing the impact of the eigenvalues. In table 5 the times for different matrices with different eigenvalues and their corresponding condition number can be seen for various algorithms. The eigenvalues of these matrices are shown in table 6 and the convergence time in table 7. "Diverge" implies that the algorithm does not converge within 100 iterations. It is not investigated whether the algorithm requires more iterations or actually diverges.

Table 5: Properties of test matrices.

matrix	a	b	c	condition number	size
1	50i	10	-10	2.33	10^6
2	50	10	-10	1.92	10^6
3	1	10	-10	42.0	10^6
4	1	100	-100	401	10^6
5	20	30	-50	2e6	10^6

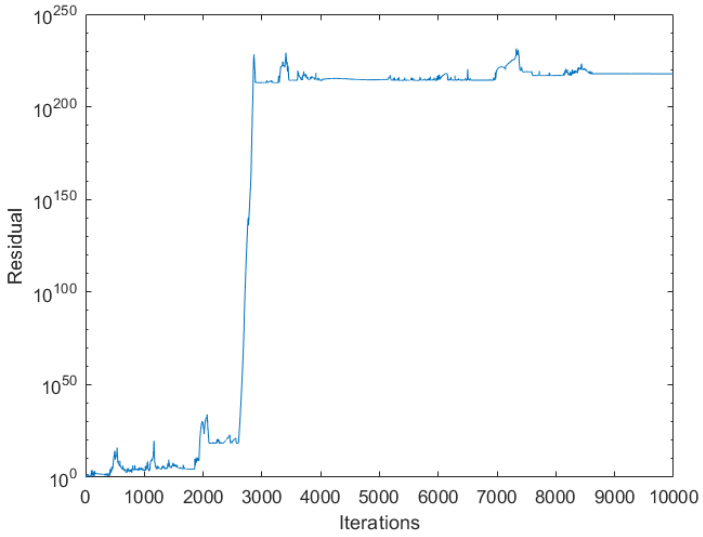

Figure 3.9: The residual as a function of the number of iterations of our implementation of IDRstab for the SHERMAN5 matrix.

Table 6: This table shows the range(s) across which the eigenvalues of the test matrices are spread evenly.

matrix	Range 1	Range 2
1	0+30i to 0+70i	50-20i to 50+20i
2	50-20 to 50+20i	-
3	1-20i to 1+20i	-
4	1-200i to 1+200	-
5	20-80i to 20+80i	-

Table 7: Time to converge to a residual of 10^{-12} in seconds. "NC" means no convergence.

matrix	Our implementation	Bicgstab	Bicgstabl	GMRES
1	22.1	1.45	21.0	8.83
2	10.8	0.87	2.567	3.50
3	NC	NC	NC	NC
4	NC	NC	NC	NC
5	NC	NC	NC	NC

It is clear from table 7 that Bicgstab, Bicgstabl and GMRES perform better or equal to our implementation of IDRstab in all five test cases. Memory requirements are not taken into account. The hypothesis that Bicgstabl and IDRStab do better for linear systems involving unsymmetric matrices with a complex spectrum from reference [7] could not be confirmed by this experiment. It is also not taken into account that the implementation of IDRstab can still be improved significantly. These potential improvements are discussed in section 4.

4 Discussion

4.1 The implementation of the algorithm

As we saw in section 3.3 the implementation of algorithm 1 described in reference [8] cannot solve the SHERMAN5 system. This means that our algorithm still contains at least one minor error. You might therefore wonder why we would bother testing a non functional algorithm at all. It turned out however that there was a lot to learn from testing the algorithm, but first we take look at the reasons why the results from reference [8] could not be reproduced.

The main issue was that the representation of the algorithm contained a lot of ambiguous statements. It is not mentioned where the "if" statements close. The statements from reference [22] from the same writers was used to determine the "end if" locations from these preconditioned IDRstab algorithms. This means that we cannot know with certainty whether all "end if" statements are in the right place. Apart from this incompleteness there were also inconsistencies. On line 20 we have to take columns 1 up to q-1 from $\mathbf{V}$. The first time that this statement is reached is when q=1 such that we have to take the first until the zeroth column. This was interpreted as just skipping this step all together but it is unclear if this is correct.

The main problem arises at line 19. On the page before the algorithm it is stated that the vector $\vec{u}$ contains $\vec{u_0}$ up to $\vec{u_j}$. On line 19, $\vec{u}$ is defined such that it contains the vectors $\vec{u_1}$ up to $\vec{u_j}$. Further in the algorithm we need to take vectors such as $\vec{u_1}$ out of $\vec{u}$. It is not clear whether we should now take the first or the second vector from $\vec{u}$. We can now call the first vector in $\vec{u}$ $\vec{u_0}$ or $\vec{u_1}$ depending on the definition that we follow. The same can be said for $\vec{u_2}$, $\vec{u_3}$ etc.

Apart from actual mistakes in the algorithm it is also far from optimised. For example, the multiplication $\mathbf{AR}_0$ is performed three times per iteration even though it is constant and can just as easily be cached in a variable. This particular problem was fixed but no further attempt was made to do more optimisation. Furthermore every time a part of a matrix or vector had to be used, this was accomplished by the function which can be found at the bottom of the MATLAB algorithm in appendix B. This function actually makes unnecessary copies which slows the algorithm down significantly. This could definitely be accomplished in a more efficient way. It is clear that there is a lot of room for improvement in terms of performance such that the convergence times found in 3.2 can definitely be reduced by a relevant margin.

4.2 Relevance of the results

Due to what was discussed in the previous section, implementing the algorithm in matlab took a lot longer than expected. There was therefore not enough of time to test enough test cases to draw very specific conclusions on the influence of eigenvalues and the condition number, and their influence on the choice of L and S.

First of all most test problems used to test the algorithm were very artificial. Matrices with random elements plus a multiple of the identity matrix do not appear in real physical problems. Toeplitz matrices could appear (see equation 2.18), but the eigenvalues distribution is unlikely to be this smooth. Besides, the tri-diagonal Toeplitz matrices used in this report could be solved by direct methods a lot faster.

Furthermore not all relevant aspects of the efficiency of the different algorithms were taken into account. The number of AXPYs and MVs was not used to compare algorithms. In addition, the amount of memory used was not analysed even though this can impact the performance, especially for very large systems, significantly.

The MATLAB implementation of IDRstab could not compete with other Krylov algorithms as explained in section 3.3. There is, however, a lot of room for improvement so IDRstab might be as good as literature promises [8], but it could not be shown in this report.

5 Conclusion

The results from reference [8] could unfortunately not be verified. This implies that the MATLAB implementation contains one or several flaws. The algorithm was tested for various test cases regardless. The test cases were divided into dense and sparse matrices. We have seen in section 3.2 that the condition number seems to have a strong influence on the convergence speed of the algorithm. It turned out that the condition number was in the order of 10^2 for all test cases for the algorithm to converge within ten iterations, despite the very strong dependence of the condition number on the matrix.

The dependence on the choice of L and S on the converging time was also analysed. As shown in table 4 the choice of L and S do not seem to be of great influence for the particular test case. The choice $L = 1$ and $S = 3$ turned out particularly bad. However, one test case is definitely not enough to draw a convincing conclusion. We can conclude however that many combinations of L and S have the potential to work well.

In section 3.3 Toeplitz matrices were used to compare the convergence of several Krylov methods in terms of the eigenvalues. Bicgstab, Bicgstabl and GMRES converged faster than the matlab implementation. Asymmetric matrices did not converge at all. In these test cases, no dependence on the eigenvalues distribution was found. The one thing we can take from this section is that our implementation of IDRstab needs improvements to become competitive with alternative Krylov algorithms. Potential ways to accomplish these improvements have been discussed in section 4.

To sum up, it is clear that our implementation needs some improvements to become competitive with other algorithms. Luckily we know that these improvements are possible, so perhaps IDRstab can indeed compete with or even outperform other Krylov methods. We think that this implementation can work as a suitable guideline to support further attempts to implement IDRstab.

6 Outlook

As the reader might have noticed, not many actual general conclusions could be drawn from the numerical experiments performed in this report. This section will describe three possible ways forward which may help future research to accomplish more concrete results. Three possible ways forward are listed below

6.1 Improve the algorithm.

It is rather trivial that the first way forward would be to improve the performance of the algorithm. This actually consists of two separate steps. The first one is to fix it in a way such that the results from [8] can be reproduced. The next step is to optimise the efficiency of the algorithm. This includes saving parameters that are needed in the future such as the $\mathbf{AR}_0$ discussed in section 4, but also avoiding making unnecessary copies and being clever with shifting smaller vectors in a larger vector or the columns of a matrix. This new algorithm might just have the properties that we expect but could not quite detect in the research of this report.

6.2 More test cases by using of statistics

Once the algorithm works, there is a major potential for statistical analysis. Using a clever algorithm, one could for example produce many Toeplitz matrices, determine the eigenvalues and condition number and then check the time needed to reach convergence. These could then be compared to other algorithms in a similar fashion as done in this report. Doing this for a large number of matrices might give more insight in the importance of properties such as the condition number or eigenvalues. It seems that the vast amount of relevant properties make this problem very suitable for statistical investigation.

6.3 A connection to physics.

A third interesting option is to take a completely different perspective, for example from physics. Dynamical systems contain an advection term and a diffusion term which strongly influence the eigenvalues of the matrix produced from that equation. It is therefore very interesting to see how the advection and diffusion terms determine the convergence of Krylov algorithms and to specify which algorithms are suitable for which type of physical system in terms of the physics. Furthermore, the linear systems which arise from such problems are systems which often require Krylov methods as a solution tool. This means that it is more natural to use Krylov methods in the first place which gives the analysis a more direct impact.

6.4 Comparison to original IDRstab implementation

Towards the end of this project, a full MATLAB implementation of the original IDRstab was found [23]. In this section we will briefly discuss the performance of this original IDRstab algorithm and compare it to the performance of our algorithm. Furthermore we will briefly compare the MATLAB code to our code and use this perspective to find the potential bugs in our code.

The first thing that we tested were matrices of the form tested in section 3.2. This means dense matrices with random elements. As was shown in section 3.2, our implementation could only solve matrices with a certain diagonal dominance. The implementation from reference [23] can solve random matrices for $\xi = 0$ using equation 3.2. Furthermore, the algorithm can solve all the SHERMAN matrices. From all of the above it should be quite obvious that this implementation is very effective so it is interesting to discuss some of the differences between the codes.

One important differences that the q-loop, which was very troublesome in our implementation, goes from 2 to S instead of 1 to S. This immediately solves the problem with the columns of V discussed in section 4. Doing this requires a slightly different structure of the loop but it would be interesting to check if a similar structure can be integrated in the implementation described in this report.

A useful trick to avoid running into trouble with changing matrix and vector sizes during the algorithm is the use of the statement "end", which can be used to select a number of columns or rows from a matrix or vector up to and including the last one.

Finally, this implementation has not used any preconditioning to the $\boldsymbol{\sigma}$ matrix. This does not seem so be an issue however, because even though MATLAB sometimes complains that this matrix becomes nearly singular, the algorithm still arrives to the right answer.

We think that this implementation can provide as a suitable guideline to critically investigate our implementation and potentially help further research to find the bug. It proves once again that IDRstab deserves a lot of attention.

References

[1] James J Buckley. Solving fuzzy equations in economics and finance. *Fuzzy Sets and Systems*, 48(3): 289–296, 1992.

[2] David J Griffiths and Darrell F Schroeter. *Introduction to quantum mechanics*. Cambridge University Press, 2018.

[3] Robert A. Adams. *Calculus: A Complete Course*. Robert A. Adams, jul 2019. ISBN 0321781074. URL `https://www.xarg.org/ref/a/0321781074/`.

[4] Saeid Abbasbandy, Reza Ezzati, and Ahmad Jafarian. Lu decomposition method for solving fuzzy system of linear equations. *Applied Mathematics and Computation*, 172(1):633–643, 2006.

[5] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[6] Martin H Gutknecht. Idr explained. *Electronic Transactions on Numerical Analysis*, 36(3):126–148, 2010.

[7] Gerard LG Sleijpen and Martin B Van Gijzen. Exploiting bicgstab () strategies to induce dimension reduction. *SIAM journal on scientific computing*, 32(5):2687–2709, 2010.

[8] Kensuke Aihara, Kuniyoshi Abe, and Emiko Ishiwata. A variant of idrstab with reliable update strategies for solving sparse linear systems. *Journal of Computational and Applied Mathematics*, 259: 244–258, 2014.

[9] V Vassilevska Williams. Breaking the coppersmith-winograd barrier. *E-mail address: jml@ math. tamu. edu*, 2011.

[10] Henk A Van der Vorst. *Iterative Krylov methods for large linear systems*, volume 13. Cambridge University Press, 2003.

[11] Horst D Simon. Direct sparse matrix methods. *Modern Numerical Algorithms for Supercomputers*, pages 325–444, 1989.

[12] Steven J Leon, Ion Bica, and Tiina Hohn. *Linear algebra with applications*. Macmillan New York, 1980.

[13] R. V. KOHN M. H. WRIGHT P. G. CIARLET, A. ISERLES. *Iterative Krylov methods for large linear systems*. TU/e, Eindhoven, 2016.

[14] Homer F Walker. Residual smoothing and peak/plateau behavior in krylov subspace methods. *Applied numerical mathematics*, 19(3):279–286, 1995.

[15] Rüdiger Weiss. A theoretical overview of krylov subspace methods. *Applied numerical mathematics*, 19(3):207–233, 1995.

[16] Parul Pandey. Understanding the mathematics behind gradient descent., 2019. URL `https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e`.

[17] Mark Embree. How descriptive are gmres convergence bounds? 1999.

[18] Silvia Noschese, Lionello Pasquini, and Lothar Reichel. Tridiagonal toeplitz matrices: properties and novel applications. *Numerical linear algebra with applications*, 20(2):302–326, 2013.

[19] Biswa Nath Datta. *Numerical linear algebra and applications*, volume 116. Siam, 2010.

[20] C Bailey, GA Taylor, M Cross, and P Chow. Discretisation procedures for multi-physics phenomena. *Journal of Computational and Applied Mathematics*, 103(1):3–17, 1999.

[21] matrixmarket. `https://math.nist.gov/MatrixMarket/`.

[22] Kensuke Aihara, Kuniyoshi Abe, and Emiko Ishiwata. Preconditioned idrstab algorithms for solving nonsymmetric linear systems. *International Journal of Applied Mathematics*, 45(3), 2015.

[23] R Suidgeest. Recycling techniques for induced dimension reduction methods used for solving large sparse non-symmetric systems of linear equations. Master's thesis, 2017.

Appendix

A The algorithm

Input matrix A = (n x n).
Input solution vector b = (n x 1).
Define initial guess vector x = (n x 1).
Define matrix R0 = (n x s).
r0=b-A*x
r=[r0] for q=1:s

if q==1
u0=r0
else
u0=A*u0
mu=U0(:,1:q-1)'*u0
u0=u0-U0(:,1:q-1)*mu
end if
u0=u0/norm(u0)
U0(:,q)=u0
end for

while norm[r0] > tol
for j=1:L
σ = (A'*R0)'*U_{j-1}
if j==1
α=$\sigma^{-1} * (R'_0 * r_0)$
else
α=$\sigma^{-1} * ((A' * R_0) * r_{j-2})$
end if
$x = x + U_0 * \alpha$
$r_0 = r_0 - A * (U_0 * \alpha)$
for i=1:j-2
$r_i = r_i - U_{i+1} * \alpha$
end for
if $j > 1$
$r = [r; A * r_{j-2}]$
end if
for q=1:s
if q==1
u=r
else
for k=1:j
$u((k-1) * n + 1 : k * n, 1) = u_k$
end for
if length(u)¿j*n
u(j*n+1:length(u))=[];
end if
end if
$\beta = \sigma^{-1} * ((A' * R_0)' * u_{j-1})$
$u = u - U * \beta$
$u = [u; A * u]$
if q > 1
$\mu = V'_{j(:,1:q-1)} * u_j$

$u = u - V_{j(:,1:q-1)} * \mu$
end if
$u = u/norm[u_j]$
if q==1
$V = zeros(1 : length(u), q)$
end if
$V_{1:length(u),q} = u;$
end for
U=V
end for
$r = [r; A * r_{l-1}]$

for i=1:L
$rmatrix0(:, i) = r_{i-1}$
end for
for i=1:L
$rmatrix1(:, i) = r_i$
end for
$\gamma = rmatrix1^{-1} * r_0$
$x = x + rmatrix0 * \gamma$
$r_0 = r_0 - A * rmatrix0 * \gamma$
Umatrix=zeros(n,s)
for k=1:L
$Umatrix = Umatrix + U_k * \gamma(k, 1)$
end for
$U = U_0 - Umatrix$
end while

B IDRstab in MATLAB

```
clear
rng(1);
n=100;%The dimension of the matrix is n x n.
s=4;
L=2;%s=4 and l=2 are the standard parameters for IDR(s)Stab(l).
tol=1e-12;%This is the tolerance of the method.
B=rand(n);
A=(B+B')/2+0*eye(n);%This makes A a symmetric matrix of size n x n.
b=rand(n,1);%The vector b is a random solution to Ax=b.
%x_exact=A\b;%This can be used to find the difference between the exact
%solution and the solution of the algorithm for matrices that are small
%enough to be solved in exact fashion.
%{
path='C:\Users\s157181\Documents\jaar 4\bep\testmatrices\Sherman\';
A=mmread([path,'Sherman2.mtx']);

disp('Done loading A')
b=mmread([path,'sherman2_rhs1.mtx']);
disp('Done loading b')
%}
tic
n=size(A,1);
x=zeros(n,1);%The first trial solution
R0=rand(n,s);
R0=orth(R0);
U0=rand(n,s);%R0 and U0 are a random auxillary matrices of n x s.
r0=b-A*x; %This is the first residual.
r=r0;%This is the vector containing all vectors r0...rj in a long vector
V=zeros(n,s);
```

```
maxiter=100;
AR0=A'*R0;%This operation has been discussed in the discussion.

%Arnoldi step
for q=1:s
    if q==1
       u0=r0; %In the first iteration u0 is set to be the initial residual.
    else
        u0=A*u0;
        %In any iteration that isnt the first one, the new u0 is multiplied
        %by A to introduce a new dimension of the krylov space.
        mu=U0(:,1:q-1)'*u0;
        u0=u0-U0(:,1:q-1)*mu;
    end
    u0=u0/norm(u0);%This normalises u0.
    U0(:,q)=u0;%This puts u0 in the qth column of U0.
end
U=U0; %The columns of U0 and U are ortho-normalised.

while norm(b-A*x)/norm(b)>tol && maxiter>0
    r=r(1:n,1);
    for j=1:L
        %IDR step
       sigma=(AR0)'*Outputmatrix(n,j-1,U);%This takes out the j-1the matrix of U. so Outputmatrix(n,1,U) gives
       P=diag(1./sum(abs(sigma),1));
       sigma=sigma*P;
       if j==1
           alfa=sigma\(R0'*Outputmatrix(n,0,r));
       else
           alfa=sigma\((AR0)'*Outputmatrix(n,j-2,r));%The first iteration this else is reached j=2 so we need
       end
           alfa=P*alfa;
           x=x+Outputmatrix(n,0,U)*alfa;
           r(1:n)=r(1:n)-(A*(Outputmatrix(n,0,U)*alfa));

       for i=1:j-2
           r((i*n)+1:(i*n)+n)=r((i*n)+1:(i*n)+n)-Outputmatrix(n,i+1,U)*alfa;
       end

       if j>1
           r=[r; A*Outputmatrix(n,j-2,r)];
       end

       for q=1:s
          if q==1
             u=r;
          else
              for k=1:j
                  u((k-1)*n+1:k*n,1)=Outputmatrix(n,k,u);
              end
              if length(u)>j*n%if u is longer than j*n as needed this if statement removes the rows j*n+1 to t
                  u(j*n+1:length(u))=[];
              end
          end
          beta=sigma\((AR0)'*Outputmatrix(n,j-1,u));
          u=u-U*beta;
          u=[u; A*Outputmatrix(n,j-1,u)];
          if q>1
            Vj=Outputmatrix(n,j,V);
            mu=Vj(:,1:q-1)'*Outputmatrix(n,j,u);
            u=u-V(:,1:q-1)*mu;
          end
          u=u/norm(Outputmatrix(n,j,u));
          if q==1
              V=zeros(length(u),s);
          end
          V(1:length(u),q)=u;
       end
    U=V;
    end
```

```
    r=[r; A*Outputmatrix(n,L-1,r)];
    %Polynomial step
    for i=1:L
        rmatrix1(:,i)=Outputmatrix(n,i,r);
    end
    for i=1:L
        rmatrix0(:,i)=Outputmatrix(n,i-1,r);
    end
    gamma=rmatrix1\Outputmatrix(n,0,r);
    x=x+rmatrix0*gamma;
    r(1:n,1)=r(1:n,1)-A*(rmatrix0*gamma);
    Umatrix=zeros(n,s);
    for k=1:L
        Umatrix=Umatrix+Outputmatrix(n,k,U)*gamma(k,1);
    end
    U=Outputmatrix(n,0,U)-Umatrix;
    r1=Outputmatrix(n,1,r);
    if norm(b-A*x)/norm(b)<tol
       residue=norm(b-A*x)/norm(b);
       Z=['the residual is:',num2str(residue),'.'];
       disp(Z)
       toc
       return
    end
    if r1==0
       residue=norm(b-A*x)/norm(b);
       Z=['the residual is:',num2str(residue),'.'];
       disp(Z)
       toc
        return
    end
    maxiter=maxiter-1;
end
residue=norm(b-A*x)/norm(b);
Z=['the residual is:',num2str(residue),'.'];
disp(Z)
toc

%This function takes out the n*j+1'th up to the n*j+n'th row(s) from a
%matrix or vector.
function Output=Outputmatrix(n,j,Input)
g=n*j+1;
h=n*j+n;
Output=Input(g:h,:);
end
```

C Arnoldi algorithm

```
clear
m=10;
A=rand(m);%This gives a m x m with random entries on the domain (0,1).
v1=rand(m,1);
v1=v1/norm(v1);%v1 is a random normalised vector.
w_vectors=zeros(m,m);%This creates a matrix of which the columns will be filled with the vectors w_j.
v_vectors=zeros(m,m);%This creates a matrix of which the columns will be filled with the vectors v_j which are
v_vectors(:,1)=v1;

Hmatrix=zeros(m);%This makes the matrix with elements h_ij.
for j=1:m
    for i=1:j
        Hmatrix(i,j)=dot(A*v_vectors(:,j),v_vectors(:,i));
        sumseries=zeros(m,1);
        for i=1:j
            sumseries(:,1)=sumseries(:,1)+Hmatrix(i,j)*v_vectors(:,i);
        end
        w_vectors(:,j)=A*v_vectors(:,j)-sumseries;
    end
```

```
    Hmatrix(j+1,j)=norm(w_vectors(:,i));
    if Hmatrix(j+1,j)==0
        break
    else
        v_vectors(:,j+1)=w_vectors(:,j)/(Hmatrix(j+1,j));
    end
end

testmatrix=zeros(m,m);%Testmatrix gives the dot products of all the v_vectors. The entries on the diagonal of
for i=1:m
    for j=1:m
        testmatrix(i,j)=dot(v_vectors(:,i),v_vectors(:,j));
    end
end
testmatrix
```

D Local minimum residual approach

```
clear
%This algorithm is called local minimum residual.
alfa=1;
tol=1e-12;
n=10;
B=rand(n);
A=(B+B')/2+100*eye(n);
n=size(A,1);
b=ones(n,1);
x=zeros(n,1);
x_exact=A\b;
r=b-A*x;
i=0;
count=0;
while norm(r)>tol
    alfa=dot(A*r,r)/dot(A*r,A*r);
    x=x+alfa*r;
    r=(eye(n)-alfa*A)*r;
    count=count+1;
    if count==100
        norm(x-x_exact)/norm(x_exact)
        return
    end
end
norm(x-x_exact)/norm(x_exact)
```