Acceleration of Iterative Methods for Reconstruction in Tomography

Lars Holtse Bonde

Kongens Lyngby 2011
IMM-B.Sc.-2011-24

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-B.Sc.: ISSN 0909-3192

Summary

In this bachelor thesis some iterative methods for solving *ill-posed discrete inverse problems* have been modified. These iterative methods are the *Landweber* method and the *Cimmino* methods. These methods produce *regularized* solutions that are linear combinations of some basis vectors. It has been shown that in some cases these basis vectors are a bad basis for the solution.

The work in the thesis has been to introduce a *preconditioner* that will change these basis vectors. It has been shown that a preconditioner in some cases result in a better solution.

When investigating preconditioners it was found that many natural choices were rank deficient and therefore invalid. Therefore an α-value was inserted. The effect on the solution of the placement and value of the α parameter was shown. Also a catalogue of proposed preconditioners and a MATLAB function to produce these preconditioners has been made.

The preconditioning of the two *SIRT* methods has been implemented in MATLAB for constant relaxation parameter and a number of iterations as stopping criteria.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfilment of the requirements for acquiring the B.Sc. degree in engineering. The thesis represents a workload of 20 ETCS points during the spring semester 2011. Professor Per Christian Hansen from IMM DTU has proposed and supervised the project.

During the work I of course have learned a lot academically. However I have also learned that things may take an unexpected turn resulting in having to change one's original plan several times. Also I am excited to see the difference between theoretical discussions and practical implementations. In this case how, in theory, the filter factors and SVD components play a crucial role in the understanding of the problem, but in practical implementation a completely different approach is used.

Kongens Lyngby, June 2011

Lars Holtse Bonde

Acknowledgements

First of all I would like to thank Per Christian Hansen for the support throughout the project period. He has been a great help in understanding the, for me new, theory of solving inverse problems using SVD. Besides help with the academic challenges, we have had some good discussions of how to structure the report. No matter how the end result is, I have learned a lot about structuring reports.

I would also like to thank Ph.D. student at IMM DTU Jakob Heide Joergensen for answering questions regarding his master thesis and his offer of letting me come by for a discussion.

Finally I would like to thank my family, Hanne, Jens and Niels, for support during my study at DTU so far, and their patience while listening to me talk about preconditioners and inverse problems.

Contents

List of Figures

Chapter 1

Introduction

As the science of physics developed, solving inverse problems became more and more relevant. According to wikipedia[1] the theory of inverse problems evolved with the soviet-armenian physicist Viktor Ambartsumian in the beginning of the 20th century. Since then it has become a large field of research and is used in almost any branch of science. Some inverse problems are straight forward to solve and some are very complex - at this time impossible to solve. This means that there are many mathematical approaches to solving these kind of problems.

In this bachelor thesis we will work with discrete inverse problems. They will be of a nature that make them impossible to solve in an ordinary fashion. They will be solved using some numerical methods known as *SIRT*. These methods produce good results in some cases but in other they are far from a good solution. This thesis will discuss how to use *preconditioning* to hopefully obtain better solutions in some of these cases.

During the work it was discovered that preconditioning these problems led to a thorough investigation of what effect it had. Therefore the focus of the thesis became less on numerical implementation and more on theoretical and experimental study of the effects.

The result is a catalogue of proposed matrices used for preconditioning dis-

[1] http://en.wikipedia.org/wiki/Inverse_problem

crete inverse problems, and a simple implementation on the two SIRT methods *Landweber* and *Cimmino* in MATLAB . Furthermore a MATLAB function to obtain the proposed matrices has been written.

1.1 Structure of the Thesis

The thesis will guide the reader into the world of inverse problems from tomography applications. Therefore a basic description of tomography will be taken out and coupled with the theory of inverse problems. Some of the issues regarding solving these inverse problems will be addressed to motivate the preconditioning. Then a lot of work with preconditioning will be taken out and the catalogue of preconditioning matrices is discussed. Finally the methods will be effectively implemented in MATLAB and we will see a difference of approach between theory and practice.

Chapter 2 We will briefly explain what tomography is and how this produces inverse problems. This will be used to get an understanding of why and how tomography problems are ill-conditioned.

Chapter 3 Here we will start with an introduction to inverse problems. Then some of the theory regarding solving discrete inverse problems will be explained. In this chapter we will understand some of the characteristics of the solutions from the *SIRT* methods through an example.

Chapter 4 This is one of the main chapter of the thesis. Here the preconditioning of the system will be derived theoretically and the effects of different preconditioners will be discussed.

Chapter 5 After having discussed the preconditioner's effect it is time to implement it effectively in MATLAB , which is done here.

Chapter 6 Finally we will sum up the work done in this thesis and tasks for future work will be proposed.

Appendix A This important appendix holds a list of proposed matrices used for preconditioning. Also it will be shown which of the proposed matrices can be used and which cannot.

Appendix B This appendix is a listing of the MATLAB code that can be used to obtain matrices for preconditioning.

Appendix C This appendix is a listing of preconditioning implemented to the *Landweber* method, with constant λ.

Appendix D This appendix is a listing of preconditioning implemented to the *Cimmino* method, with constant λ.

CHAPTER 2

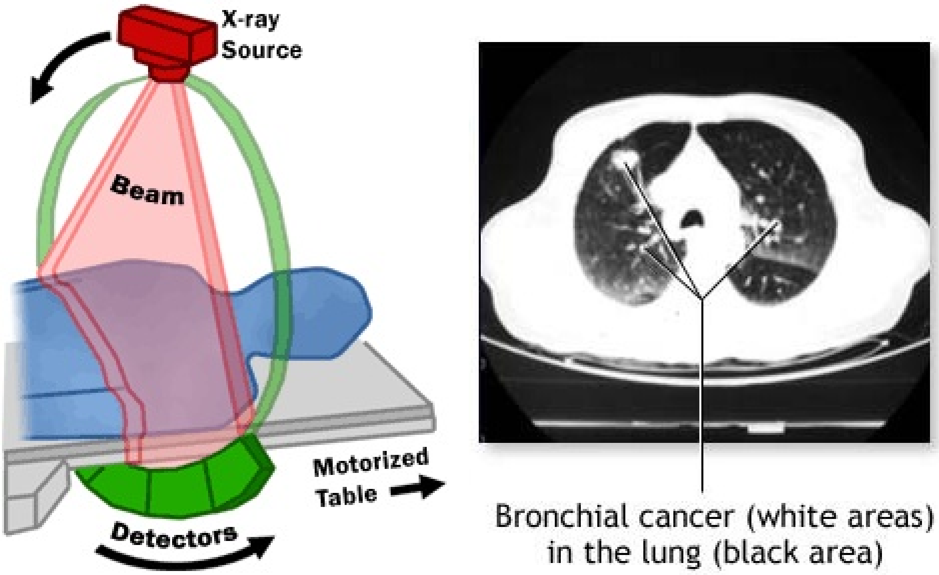
Tomography

Tomography is a method for ”looking inside objects”. The word derives from the Greek words *tomos* which means *part* or *section* and *graphein* which means *to write*. This describes the result of doing tomography. We get a slice of the object investigated, commonly as an image. Tomography is used in many applications from looking at objects in nano-scale with electron microscopy to searching for oil reserves deep under ground. A commonly known application of tomography is in CT-scanners, or Computerized Tomography scanners. Here a set of X-rays are sent through the patient's body, to see inside without having to put him/her into surgery. Tomography is a *non invasive* method and therefore very useful in many situations.

The CT-scanner works by sending a set of parallel X-ray beams, with known intensity, through the patient and measuring the outgoing intensity. This is done for several angles, typically covering 180°. This can be seen as a projection of the object onto a line for every angle. The tomography problem is to reconstruct the object from these projections. Figure 2.1(a)[1] illustrates how the CT-scanner works. The X-ray emitter rotates and thereby a set of beams are sent through the same cross section of the patient at different angles. Figure 2.1(b)[2] shows the result of the tomography. High absorption areas are bright and low absorption areas (in this case the lungs) are dark.

[1] http://www.broadwayimagingcenter.com/wp-content/uploads/ucm115328.gif

[2] http://health.allrefer.com/health/thoracic-ct-bronchial-cancer-ct-scan.html



(a) CT-scanning sends X-rays through a slice of the body.

(b) Result of a CT-scanning is an image of a cross section of the body.

Figure 2.1: Illustration of how a CT-scanner works.

We will now see the mathematical model of the described tomography problem.

2.1 Setting up the Tomography Problem

In setting up the problem we will look at the mentioned application, using X-ray beams. The X-rays decay exponentially when passing through material. Then let us first assume that the object under investigation is homogeneous, e.g. consists of the same material all the way through, and has a linear attenuation coefficient u, which specifies the material's intensity absorption. Then the X-ray intensity sent into the object, I_0, and the intensity coming out, I, can be calculated by:

$$I = I_0 e^{-ux}$$

where x is the length of the path that the beam went through the object.

Assume that the beam travels through several materials having different attenuation coefficients, at different lengths. The output intensity can now be calculated as:

$$I = I_0 e^{\sum -u_i x_i}$$

Now we say that the beam travels through infinitely many materials of different absorption. We then get the attenuation function depending on how far on the beam line we are, and we get an integral equation:

$$I = I_0 e^{\int_L -u(x)dx}$$

The L denotes the path of the X-ray beam. For the two-dimensional and three-dimensional cases we would have $u(x,y)$ and $u(x,y,z)$ respectively, or in general $u(\mathbf{x})$ and then one could substitute e.g. $dxdydz$ with ds. So in general we would have the equation:

$$I = I_0 e^{-\int_L u(\mathbf{x})ds}$$

Which can be rewritten as:

$$\int_L u(\mathbf{x})ds = -\log \frac{I}{I_0} \tag{2.1}$$

Now we have a mathematical description of what is measured for every beam in Computerized Tomography. The goal of the tomography problem is to obtain the function $u(\mathbf{x})$, from measurements of different lines from different angles through the object. In practice, though, the problem would often be discretized, so the object would be split into a finite amout of fields known as voxels (3D) or pixels (2D). We will see a way to do that in section 2.2.

2.2 Discrete Tomography Problems

Let us consider the two-dimensional case, as described earlier and seen in figure 2.1. Assume that the rays are parallel. We then have the situation seen in figure 2.2[3]. We see a set of parallel beams sent through an object at different angles, and the attenuation of each beam. If we discretize the object, we split it into pixels, which gives us an $N \times N$ matrix X. This is seen in Figure 2.3[4] with $N = 5$. Each pixel X_{ij} describes the attenuation of the X-ray. Then if we stack the columns of the matrix X we get an N^2 vector x. Consider the k'th beam. If it travels the length a_{ki} through pixel x_i it will be attenuated by $a_{ki}x_i$. For the entire beam we get that the total attenuation b_k is given by:

$$b_k = \sum_{i \in I_k} a_{ki} x_i$$

[3] From the document 'Computerized Tomography' used in course 02637. Made by Ph.D. student at IMM DTU, Jakob Heide Joergensen.

[4] From the document 'Computerized Tomography' used in course 02637. Made by Ph.D. student at IMM DTU, Jakob Heide Joergensen.

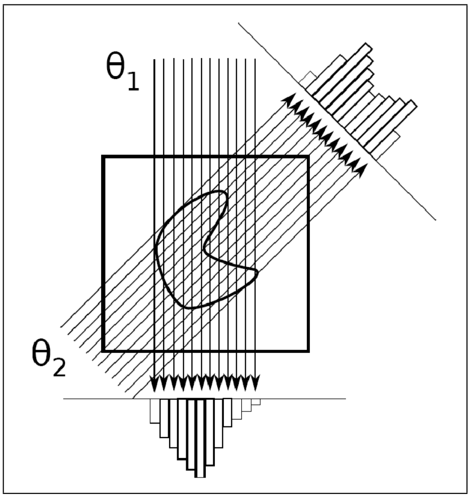
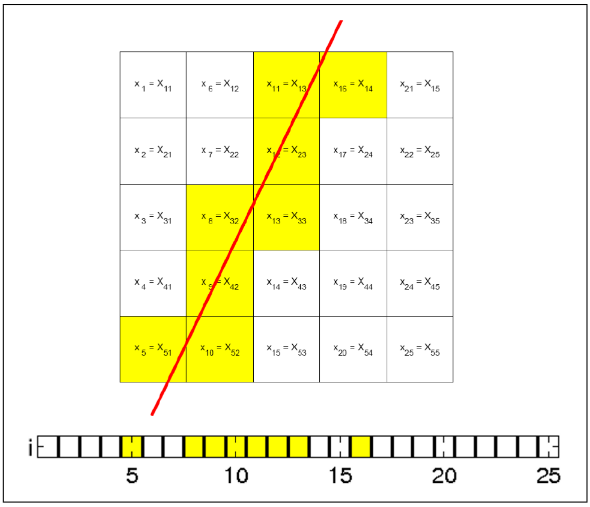


Figure 2.2: Parallel X-ray beams from two different angles, θ_1 and θ_2.



$x_1 = X_{11}$	$x_6 = X_{12}$	$x_{11} = X_{13}$	$x_{16} = X_{14}$	$x_{21} = X_{15}$
$x_2 = X_{21}$	$x_7 = X_{22}$	$x_{12} = X_{23}$	$x_{17} = X_{24}$	$x_{22} = X_{25}$
$x_3 = X_{31}$	$x_8 = X_{32}$	$x_{13} = X_{33}$	$x_{18} = X_{34}$	$x_{23} = X_{35}$
$x_4 = X_{41}$	$x_9 = X_{42}$	$x_{14} = X_{43}$	$x_{19} = X_{44}$	$x_{24} = X_{45}$
$x_5 = X_{51}$	$x_{10} = X_{52}$	$x_{15} = X_{53}$	$x_{20} = X_{54}$	$x_{25} = X_{55}$

Figure 2.3: Trace of a single X-ray beam travelling through the yellow pixels of the object. Both seen as matrix X and vector x.

where I_k is the set of pixels in x which the beam, k, has travelled through (the yellow ones in figure 2.3). If a pixel x_i has not been hit by a beam, we say that the beam has travelled the length 0 through it. Thereby giving:

$$b_k = \sum_{i=1}^{N^2} a_{ki} x_i \tag{2.2}$$

We see that the right-hand side of (2.2) reminds us of the left-hand side of (2.1).

If we send a total of M beams through the object we will get a linear system of equations. The vector b holds the calculated attenuations from (2.2) for every beam. x is the discretized 2-dimensional object stacked as a vector, as in figure 2.3. We then introduce the $M \times N^2$ matrix A which holds the information of how long every beam travels through every pixel. So that the k'th row in A is equal to the vector a_k for the k'th beam. Then we have the similar system of linear equations:

$$Ax = b \tag{2.3}$$

We remember from section 2.1 that we want to find these attenuations for every pixel, that is determine x by solving the system of equations. In practice the matrix A will be very big but also very sparse. Also there will be noise on the data so that $b = b^{exact} + e$. This is important to keep in mind when solving the equations. This thesis will deal with these discrete tomography problems and look into methods for solving (2.3).

Chapter 3

Inverse Problems

Inverse problems appear in almost any scientific field. The basic idea of inverse problems is that we want to determine some parameters from observed data. For instance we could have a model, M, which depends on some parameters, p, and gives some data, d. The relationship would then be $M(p) = d$. Say we know the model and can observe the data, but do not know the parameters of the model. We would have an inverse problem. If we knew the parameters and the model, but could not observe the data, we would have a forward problem, since we could simply plug in the parameters to the model and calculate the data. In the case of M being a discrete linear operator we would have:

$$Mp = d \tag{3.1}$$

This is equivalent to (2.3), where the situation is that we know our model, observe some data and from that need to determine the parameters - in that case attenuations.

Inverse problems can also have a continuous formulation. For instance the integral equation (2.1) is a continuous inverse problem, where we have some measurement on the right-hand side and an unknown function u, depending on some parameters $\mathbf{x}$, on the left-hand side.

In the following sections we will show why a classical method of solving a linear system of equations will not be satisfactory in the tomography case (2.3). First however we take a brief look at the *Hadamard conditions*.

3.1 Hadamard Conditions and Ill-Conditioned Matrices

Jacques Hadamard was a French mathematician who introduced the therm *well-posedness* to describe some properties of an inverse problem. The term and understanding of it is straight forward. An inverse problem is well-posed if and only if the following three properties are satisfied:

1 *Existence:* There exists at least one solution $\overline{p}$ satisfying (3.1).

2 *Uniqueness:* There is at most one solution.

3 *Stability:* The solution $\overline{p}$ is depending on a stable manner of the data. This means that small changes in d results in small changes in $\overline{p}$.

While *Hadamard* believed that all natural occurring phenomena would be well-posed it has later been shown that this is not the case. Problems that are not well-posed are said to be ill-posed.

Another concept we will be using in this chapter is the *condition number* of a matrix. As stated in [2] the condition number of a *square non-singular* matrix is defined as:

$$\kappa(A) = \| A \| \| A^{-1} \| \tag{3.2}$$

A matrix is said to be well-conditioned if it has a small condition number, and ill-condition if high. The identity matrix has the smallest possible condition number with $\kappa(I) = 1$.

3.2 Singular Value Decomposition

The Singular Value Decomposition (SVD) is a way to decompose a matrix. The reader may know of LU-decomposition and QR-factorisation which are discussed in [2, 1]. The SVD is as follows:

Theorem 3.1 *Any matrix $A \in \mathbb{R}^{m\times n}$ with $m \geq n$ can be factorized to:*

$$A = U \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^T$$

where $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{n\times n}$ *are orthogonal and* $\Sigma \in \mathbb{R}^{n\times n}$ *is diagonal:*

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \sigma_n \end{bmatrix}, \quad \sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$$

It is clear that if $m < n$ the factorization can be done on A^T. In tomography, however, m will be considerably larger than n. The proof of the theorem can be seen in [1]. The columns in U and V are called *singular vectors* and the diagonal elements in Σ are called *singular values.* Later we will see that these singular vectors are in some way the main focus of this project.

If we consider $U = [U_1 U_2]$ where $U_1 \in \mathbb{R}^{m\times n}$, we get the so called *thin SVD*:

$$A = U_1 \Sigma V^T$$

Then we can write this as the *outer product form*:

$$A = \sum_{i=1}^{n} \sigma_i u_i v_i^T \tag{3.3}$$

which can be derived from:

$$A = U_1 \Sigma V^T = [u_1 u_2 \cdots u_n] \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_n^T \end{bmatrix} =$$

$$[u_1 u_2 \cdots u_n] \begin{bmatrix} \sigma_1 v_1^T \\ \sigma_2 v_2^T \\ \vdots \\ \sigma_n v_n^T \end{bmatrix} = \sum_{i=1}^{n} \sigma_i u_i v_i^T$$

This outer product form we will use in section 3.3, to use SVD to solve a linear system of equations such as (2.3).

3.2.1 SVD and Vector Spaces

First we will refresh our knowledge of the *column space* or *range*, $\mathcal{R}(A)$, of a matrix, A. Consider the matrix $A \in \mathbb{R}^{m\times n}$, where $A = [v_1 v_2 ... v_n]$. Then the range of A is defined as:

$$\mathcal{R}(A) = span\{v_1, v_2, ..., v_n\}$$

which is equivalent to:

$$\mathcal{R}(A) = \{y \mid y = Ax \quad \forall x \in \mathbb{R}^n\}$$

In similar fashion the *null space*, $\mathcal{N}(A)$, of a matrix, A, is a subspace of the range and is defined as:

$$\mathcal{N}(A) = \{x \mid Ax = 0\}$$

Also we remember that orthogonal matrices, such as U and V, have orthonormal rows and columns.

Assuming $A \in \mathbb{R}^{m \times n}$ has rank $r \leq n$, then $n - r$ of the singular values will be equal to zero. That is:

$$\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r \geq \sigma_{r+1} = ... = \sigma_n = 0 \tag{3.4}$$

Now using the outer product form, (3.3), we get:

$$y = Ax = \sum_{i=1}^{n} \sigma_i u_i v_i^T x = \sum_{i=1}^{r} \sigma_i u_i v_i^T x = \sum_{i=1}^{r} \left(\sigma_i x v_i^T\right) u_i = \sum_{i=1}^{r} \alpha_i u_i \ , \quad \alpha_i \in \mathbb{R}$$

Because of (3.4) it must hold that:

$$z = \sum_{i=r+1}^{n} \sigma_i u_i v_i^T x = \sum_{i=r+1}^{n} \sigma_i u_i x^T v_i = \sum_{i=r+1}^{n} \left(\sigma_i u_i x^T\right) v_i = \sum_{i=r+1}^{n} \beta_i v_i = 0 \ , \quad \beta_i \in \mathbb{R}$$

We see that z is within the null space since $Az = 0$.

We have now shown two important features of the singular vectors regarding vector spaces:

Theorem 3.2 *Given* $A = U \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} V^T$*, with rank* r*, then:*

1 The singular vectors $u_1, u_2, ..., u_r$ *are an orthonormal basis in* $\mathcal{R}(A)$ *and*

$$rank(A) = dim(\mathcal{R}(A)) = r$$

2 The singular vectors $v_{r+1}, v_{r+2}, ..., v_n$ *are an orthonormal basis in* $\mathcal{N}(A)$ *and*

$$dim(\mathcal{N}(A)) = n - r$$

Also we can now define the condition number of a matrix in terms of the singular values. In (3.2) we saw how to determine the condition number of a square non-singular matrix.

Having the SVD of a, not necessarily square, matrix A with rank r, the condition number of A can be calculated by terms of the singular values:

$$\kappa\left(A\right) = \frac{\sigma_1}{\sigma_r} \tag{3.5}$$

This of course is very convenient if the SVD is already computed. We will now see how to use the SVD to find the least squares solution of a linear system of equations.

3.3 Least Squares Solutions with SVD

We will now see how the SVD can be used to find the *least squares solution* to a linear system of equations. In tomography problems we will typically have an overdetermined system of equations with *perturbations*, $Ax \sim b$, where we assume A has full rank. The SVD will then be:

$$A = [U_1 U_2] \left[\begin{array}{c} \Sigma \\ 0 \end{array}\right] V^T$$

where $A, U_1 \in \mathbb{R}^{m\times n}$. The residual is then defined as:

$$\|r\|^2 = \|b - Ax\|^2 = \|b - [U_1 U_2] \left[\begin{array}{c} \Sigma \\ 0 \end{array}\right] V^T x\|^2 \tag{3.6}$$

The x^* that minimizes (3.6) is the least squares solution. From the above an expression for x^* can be derived. This is omitted here but can be seen in done in [1, 3]. Note that the approaches in the two are somewhat different. The result however is the same:

Theorem 3.3 *Let the matrix $A \in \mathbb{R}^{m\times n}$, $m \geq n$ have full column rank and the thin SVD $A = U_1 \Sigma V^T$. Then the least squares problem $\min\limits_x \|Ax - b\|^2$ has the unique solution:*

$$x^* = V\Sigma^{-1}U_1^T b = \sum_{i=1}^{n} \frac{u_i^T b}{\sigma_i} v_i$$

It is noticed that the solution is a linear combination of the singular vectors, v_i. To get a grasp of the issues with solving tomography problems, an example will be taken out in section 3.5.

3.3.1 Issues with Solving Tomography Problems with SVD

We have now seen how a tomography problem is set up and can be discretized. We have also seen how to find the least squares solution to such a problem. One might think it is a done deal and now straight forward to solve tomography problems. However there are some complications that we will look into.

First of all, in practical applications the system matrix, A, will be very large and therefore, if not impossible, then very time consuming to compute the SVD. This will be disregarded for now, and taken care of in chapter 5. The other issues will be illustrated by an example.

Using the "shaw" test problem from the *Regularization Tools* MATLAB toolbox[1], a tomography problem is set up (section 3.5). From (3.5) the condition number of the system matrix, A, is calculated to:

$$\kappa(A) = \frac{\sigma_1}{\sigma_{40}} = \mathcal{O}\left(10^{18}\right)$$

It is clear from the condition number that it is a very ill-posed problem. From introductory numerical computation [2], it is known that working on a computer with machine precession $\mathcal{O}\left(10^{-16}\right)$, as when working in MATLAB , the solution is not to be trusted, since every digit will be influenced by errors.

The *Picard plot* in figure 3.1, shows the behaviour of the described test problem. First of all it is seen that the singular values decay as expected. Even though it is a bit difficult we see that the quantity $|u_i^T b|$ decay faster than the singular values, σ_i until some point where it levels off. Also we see that at this point the values of $\frac{|u_i^T b|}{\sigma_i}$ stop descending and start ascending. This is due to the fact that the *Discrete Picard Condition* is no longer satisfied. More of the discrete Picard condition can be seen in [3]. However we will now see how this influences the solution.

$$\frac{|u_i^T b|}{\sigma_i} = \frac{|u_i^T\left(b^{exact}+e\right)|}{\sigma_i} = \frac{|u_i^T b^{exact}|}{\sigma_i} + \frac{|u_i^T e|}{\sigma_i}$$

In theory it is known that $\frac{|u_i^T b^{exact}|}{\sigma_i}$ decays as i increases [3]. This meaning without noise and rounding errors. Knowing that e is gaussian white noise it is clear that $e \sim \textit{constant}$ regardless of which SVD components (i.e. values of i). This means that as i increases and thereby the values of σ_i decrease, the values of the quantity $\frac{|u_i^T e|}{\sigma_i}$ increase.

[1] http://www2.imm.dtu.dk/ pch/Regutools

Since $\frac{|u_i^T b^{exact}|}{\sigma_i}$ decrease and $\frac{|u_i^T e|}{\sigma_i}$ increase, at some point the noised term will dominate the sum. Meaning that when the values of term $\frac{|u_i^T (b^{exact}+e)|}{\sigma_i} = \frac{|u_i^T b|}{\sigma_i}$ starts increasing, the noise dominates. This means that at some point the SVD components are dominated by noise and do not contribute in a good manner to the solution - quite the contrary. A straight forward way to deal with this is to simply disregard the latter SVD components. This is called *Truncated Singular Value Decomposition*, *TSVD*, and is a *regularized* solution.

So in practice the least squares SVD solution will be very affected by noise. Therefore we would like to regularize the problem.

3.3.2 Regularization

Before introducing the SIRT methods we will first look at way of constructing a regularized solution. As mentioned a simple way of doing this is to simply disregard the SVD components at some point, l, and obtain the TSVD solution:

$$x_{TSVD}^{[l]} = \sum_{i=1}^{l} \frac{u_i^T b}{\sigma_i} v_i \tag{3.7}$$

However another way of regularizing the solution is to construct some variables, $[0;1]$, which can be multiplied to the SVD components and in that way phase out the components as they get noisy, instead of the brutal cut-off approach in (3.7). Say we call these variables *filter factors* and denote them $\phi_i^{[p]}$. (3.7) can then be written as:

$$x_{filtered}^{[p]} = \sum_{i=1}^{n} \phi_i^{[p]} \frac{u_i^T b}{\sigma_i} v_i \tag{3.8}$$

where p is some parameter, and we obtain a *filtered* solution.

Theorem 3.4 *The truncated SVD solution (3.7) is equivalent to a filtered SVD solution (3.8) with the following filter factors:*

$$\phi_i^{[p]} = \begin{cases} 1 & i \leq l \\ 0 & i > l \end{cases}$$

An illustration of the filter factors is seen in figure 3.4. This shows that compared to a truncated solution the filtered solution ensures a somewhat smoother transition between which SVD components are included in the solution.

3.4 Simultaneous Iterative Re-constructive Technique

The *SIRT* is a class of iterative methods to solve inverse problems. They start with an initial vector $x^{[0]}$ (starting guess), often $x^{[0]} = \underline{0}$, and they take the general form of:

$$x^{[k+1]} = x^{[k]} + \lambda T A^T M \left(b - Ax^{[k]}\right), \quad k = 0, 1, 2, ... \tag{3.9}$$

As seen in [3, 4] the *Classical Landweber Method* corresponds to setting $T = M = I$ in (3.9). The k'th iterate can be expressed as a filtered SVD soltion (3.8) where the value λ, known as the *relaxation parameter*, determines the filter factors:

$$\phi_i^{[k]} = 1 - \left(1 - \lambda\sigma_i^2\right)^k \tag{3.10}$$

It is seen that for small singular values $\phi_i^{[k]} \sim 0$. We have now come to an important conclusion.

Theorem 3.5 *The k'th iterate in the Classical Landweber Method can be expressed as a filtered SVD solution defined as:*

$$x^{[k]} = V\Phi^{[k]}\Sigma^{-1}U^T b = \sum_{i=1}^{n} \phi_i^{[k]} \frac{u_i^T b}{\sigma_i} v_i$$

where $\Phi^{[k]} = diag\left(\phi_1^{[k]}, ..., \phi_n^{[k]}\right)$ *and* $\phi_i^{[k]} = 1 - \left(1 - \lambda\sigma_i^2\right)^k$.

3.5 Tomography Test Problem

Using the *shaw* test problem from the *Regularization Tools* MATLAB toolbox[2], a tomography problem is set up. In this case we have $A \in \mathbb{R}^{40\times 40}$ and a Gaussian noise of 1.00%. The condition number of A, $\kappa(A) = 2.1 \cdot 10^8$.

The Picard plot in figure 3.1 shows that only the approximately 5 or 6 first singular vectors should be used for the solution. The filter factors (see figure 3.4) used for obtaining the Landweber solution show that the 4 first singular vectors are weighted a lot, compared to the rest. This corresponds to the observations in the Picard plot. It is also noted, in figure 3.4, that even though the number of iterations, k, increase the filter factors do not change significantly.

[2] http://www2.imm.dtu.dk/ pch/Regutools

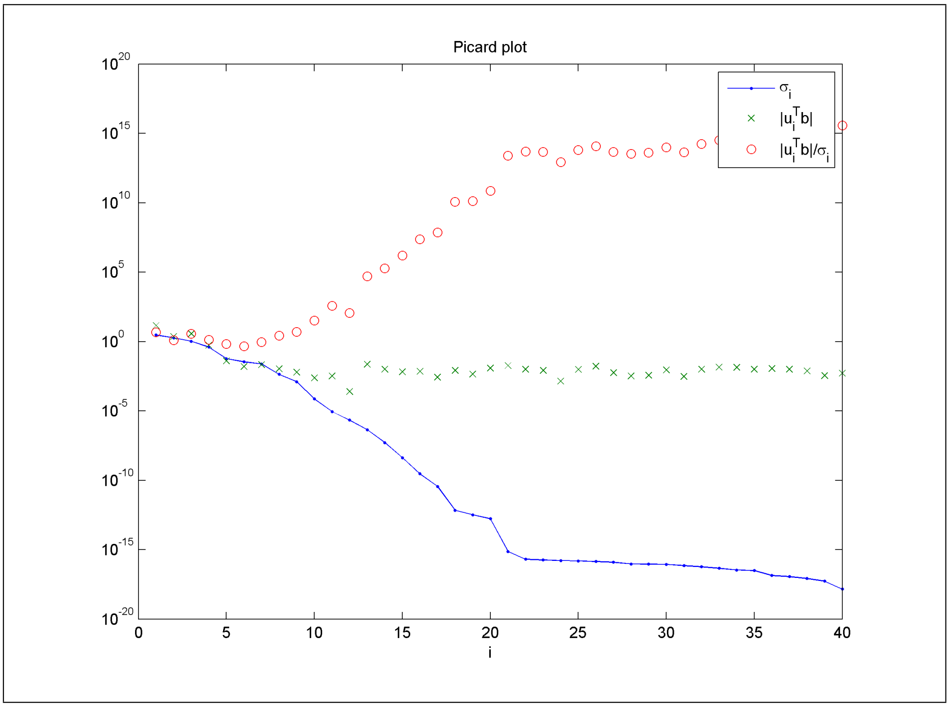


Figure 3.1: Picard plot of the *shaw* test problem from Regularization Tools, with noise level 1.00%. Showing that at some point the noise will dominate the SVD components.

Figure 3.2 shows the naive solution to the test problem. In line with the knowledge from introductory numerical computation, the large condition number of A makes the classical solution to a linear system of equations unreliable. The solution looks like Gaussian noise and is far from the true solution, which is easier to see in figure 3.5, where it is plotted together with the regularized solution obtained with the Landweber method. It should be noted that a constant $\lambda = \frac{1}{\sigma_1}$ was used. A discussion of how λ can be chosen and changed throughout the iterations can be seen in [4].

Looking at figure 3.5 we see that the Landweber solution approximates the problem and has the same tendencies. However it seems to be shifted a little to the left. This solution was obtained with 20 iterations and it did not improve significantly with more iterations. The solution obtained with the Landweber method is subject to two important factors:

1 The singular vectors seen in figure 3.3. No matter how many iterations

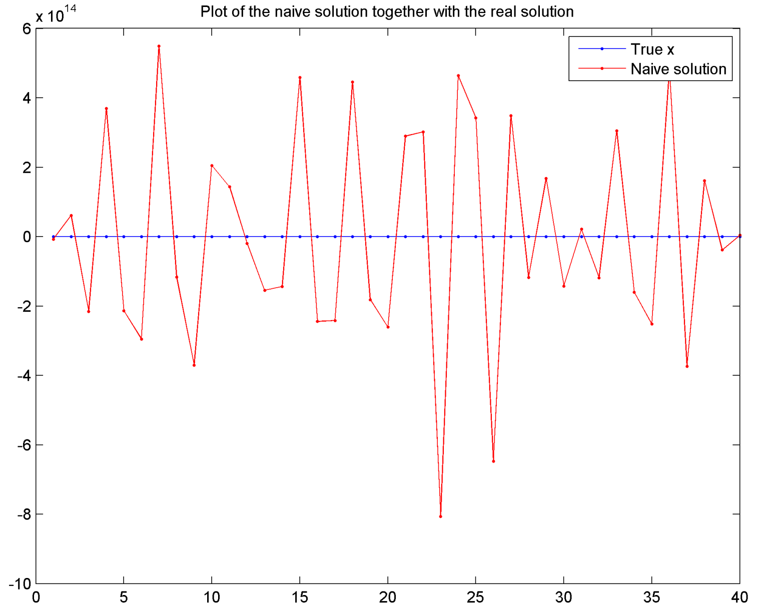


Figure 3.2: Illustration of why the naive solution is not suitable for ill-posed problems.

we use the solution is still a linear combination of the singular vectors.

2 The filter factors seen in figure 3.4. We see that as the number of iterations, k, increase the change in the filter factors, $\phi_i^{[k]}$, becomes smaller. This is mathematically described in theorem 3.5.

In this case it looks from the regularized solution that the basis vectors, v_i, was an acceptable basis for the real solution. However this is not always the case, as we will see in section 4.3. This can mean that a good solution is impossible to obtain. The goal of the next chapter is to change the basis of the solution, i.e. the singular vectors, and thereby obtain better solutions quicker, in some cases.

Another thing to note about the regularized solution is that it is somewhat smoother than the real solution. This is a property of the *SIRT* methods which Landweber is one of.

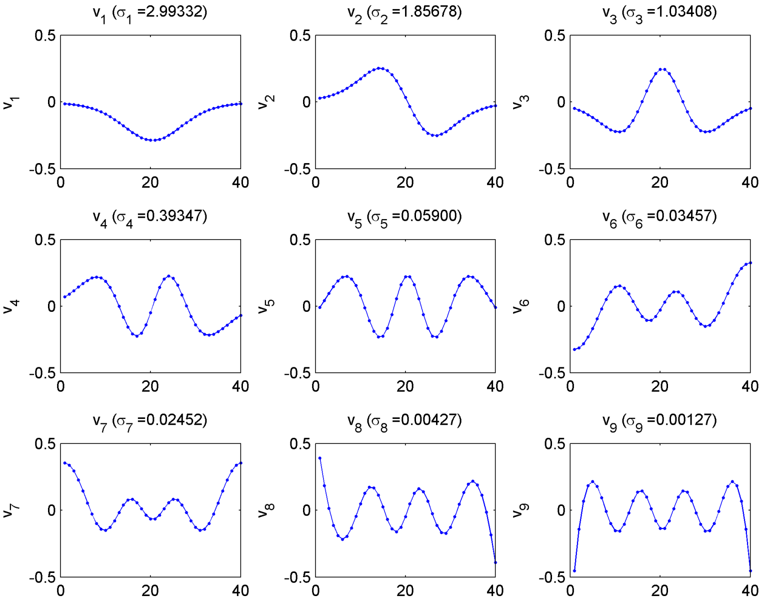


Figure 3.3: Showing the first 9 singular vectors of the *shaw* test problem.

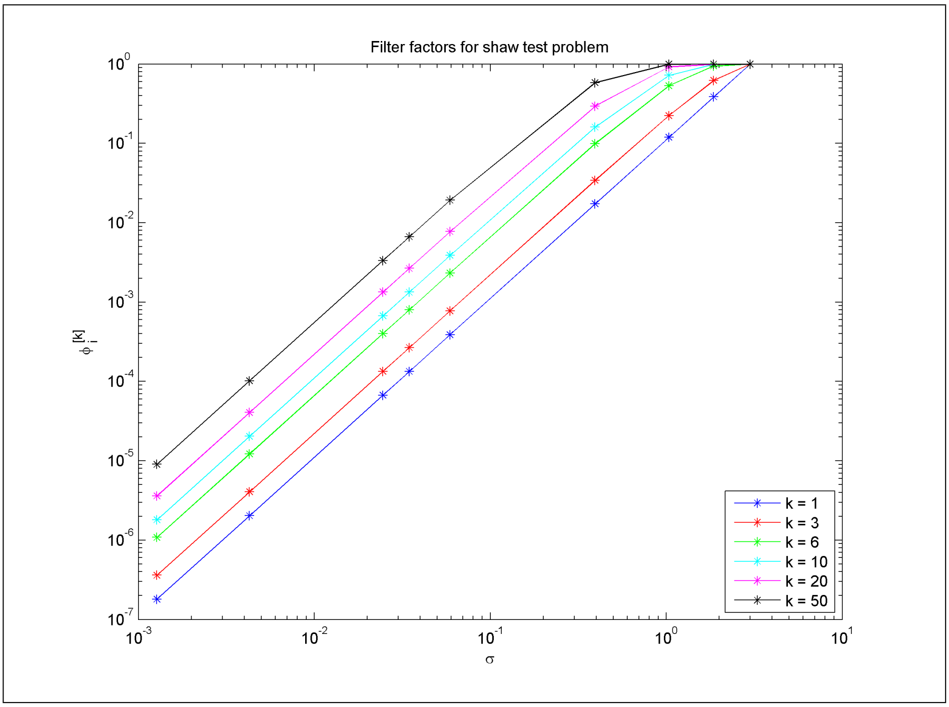


Figure 3.4: Plot of the 9 first filter factors, $\phi_i^{[k]}$, for a filtered SVD solution to the *shaw* test problem. It is seen that the solution is primarily a linear combination of the first 4 singular vectors.

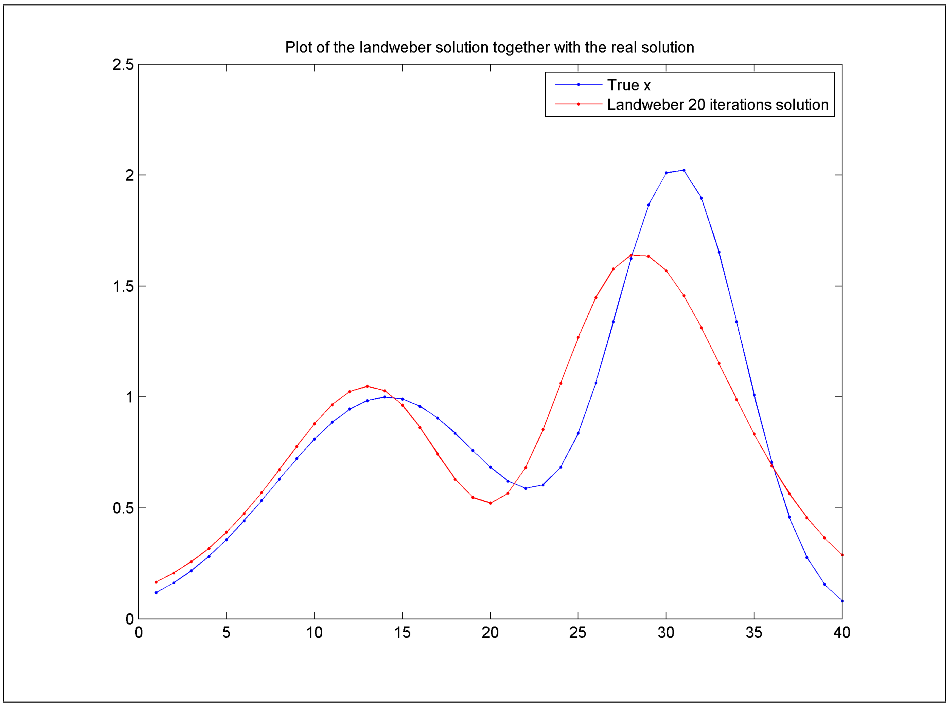


Figure 3.5: Showing the Landweber solution after 20 iterations, together with the real solution.

Chapter 4

Acceleration of SIRT using Preconditioning

This chapter will look into the *Simultaneous Iterative Reconstructive Technique* (SIRT) methods, to solve tomography problems. Especially we will look into the *Landweber* algorithm and use *preconditioning* in the hope of getting better results in some cases. As seen in section 4.3, the regularized solution does not always lie within an appropriate subspace in regards of the real solution. That is the basis for our solution, $v_i, \ i = 1, 2..., n$, may be close to parallel in some directions/dimensions.

4.1 Preconditioning in the Landweber Iterations

Because of the fact mentioned in section 3.5, where it was shown that the basis from the SVD solution to the system was not necessarily good. This might change by *preconditioning* the system. Preconditioning will transform the system, so that the regularized solution will be described by another set of basis vectors. It will however still be a regularized solution for the same problem, just in another subspace.

To motivate the preconditioner, we will now transform our original problem,

$Ax = b$ by defining a matrix, L, and vector ξ such that:

$$\left.\begin{array}{l} x = L^{-1}\xi \\ \xi = Lx \end{array}\right\} \Rightarrow \begin{array}{l} \left(AL^{-1}\right) Lx = b \\ \left(AL^{-1}\right) \xi = b \end{array} \tag{4.1}$$

Now the *Generalized SVD* (GSVD), which is described in [3], is used to determine a solution to the transformed system. We will not go into detail with the GSVD here, but instead jump to the definition.

Consider the system $Ax = b$ which is transformed into $\left(AL^{-1}\right)\xi = b$, $A, L \in \mathbb{R}^{m\times n}, m \geq n$. The GSVD is then defined as:

$$\begin{array}{l} A = U_1'\Sigma'\left(X'\right)^{-1} \\ L = V'M'\left(X'\right)^{-1} \end{array} \tag{4.2}$$

where U_1', V' are orthogonal and Σ', M' are diagonal.

Let us define a new system:

$$A'\xi = b \quad , \quad A' = AL^{-1} \tag{4.3}$$

Before it was shown that A' from (4.3) has the SVD defined as:

$$A' = U_1'\Gamma'\left(V'\right)^T \tag{4.4}$$

If we apply the Classical Landweber to the system (4.3) we will obtain the iterations:

$$\xi^{[p]} = V'\Phi'^{[p]}\left(\Gamma'\right)^{-1}\left(U_1'\right)^T b = \sum_{i=1}^{n} \phi_i'^{[p]} \frac{u_i'^T b}{\gamma_i'} v_i' \tag{4.5}$$

Theorem 3.5 gives the following definition of Φ'.

$$\Phi' = \begin{bmatrix} \phi_1'^{[k]} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \phi_n'^{[k]} \end{bmatrix} \tag{4.6}$$

where $\phi_i'^{[k]} = 1 - \left(1 - \lambda\left(\gamma_i'\right)^2\right)^k$.

To define Γ' we will now look at the GSVD defined in theorem 4.2, making the following clear:

$$\begin{array}{rcl} L^{-1} & = & X'\left(M'\right)^{-1}\left(V'\right)^T \\ AL^{-1} & = & U_1'\Sigma'\left(X'\right)^{-1}X'\left(M'\right)^{-1}\left(V'\right)^T \\ & = & U_1'\left(\Sigma'\left(M'\right)^{-1}\right)\left(V'\right)^T \end{array} \tag{4.7}$$

By letting $A' = AL^{-1}$, Γ' can now be expressed using (4.4) and (4.7):

$$\begin{aligned}
\Gamma' &= \Sigma' \left(M'\right)^{-1} \\
&= \begin{bmatrix} \sigma'_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma'_n \end{bmatrix} \begin{bmatrix} \mu'_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mu'_n \end{bmatrix}^{-1} \\
&= \begin{bmatrix} \sigma'_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma'_n \end{bmatrix} \begin{bmatrix} \frac{1}{\mu'_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\mu'_n} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\sigma'_1}{\mu'_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{\sigma'_n}{\mu'_n} \end{bmatrix} && (4.8) \\
&\Rightarrow \quad \gamma'_i = \frac{\sigma'_i}{\mu'_i} \quad , \quad i = 1, 2, ..., n && (4.9)
\end{aligned}$$

Using the GSVD we have now expressed γ'_i as a fraction between σ'_i and μ'_i. Thereby we have a good understanding of the preconditioned system. However we want to transform the system back to our original vector space, during so simply by multiplying with L^{-1}. Remembering that $\xi^{[k]}$ lies within our transformed vector space and $x^{[k]}$ within the original (see (4.1)). Using (4.5) and theorem 4.2, getting back to the original vector space is done by:

$$\begin{aligned}
\bar{x}^{[k]} &= L^{-1}\xi'^{[k]} \\
&= X' \left(M'\right)^{-1} \left(V'\right)^{T} V' \Phi'^{[k]} \left(\Gamma'\right)^{-1} \left(U'_1\right)^{T} b \\
&= X' \left(M'\right)^{-1} \Phi'^{[k]} \left(\Gamma'\right)^{-1} \left(U'_1\right)^{T} b && (4.10)
\end{aligned}$$

Note the bar in the latter. This is to indicate that even though $\bar{x}$ is in the original vector space, the solution is not expressed by the same basis vectors as x. This will be clear later.

We will now look into one of the factors of the result in (4.10) and use the expression for Γ' from (4.8).

$$
\begin{aligned}
(M')^{-1}\,\Phi'^{[k]}\,(\Gamma')^{-1} &= \begin{bmatrix} \mu'_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mu'_n \end{bmatrix}^{-1} \begin{bmatrix} \phi_1'^{[k]} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \phi_n'^{[k]} \end{bmatrix} \begin{bmatrix} \frac{\sigma'_1}{\mu'_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{\sigma'_n}{\mu'_n} \end{bmatrix}^{-1} \\
&= \begin{bmatrix} \frac{1}{\mu'_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\mu'_n} \end{bmatrix} \begin{bmatrix} \phi_1'^{[k]} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \phi_n'^{[k]} \end{bmatrix} \begin{bmatrix} \frac{\mu'_1}{\sigma'_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{\mu'_n}{\sigma'_n} \end{bmatrix} \\
&= \begin{bmatrix} \frac{1}{\mu'_1}\phi_1'^{[k]}\frac{\mu'_1}{\sigma'_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\mu'_n}\phi_n'^{[k]}\frac{\mu'_n}{\sigma'_n} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\phi_1'^{[k]}}{\sigma'_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{\phi_n'^{[k]}}{\sigma'_n} \end{bmatrix} \\
&= \begin{bmatrix} \phi_1'^{[k]} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \phi_n'^{[k]} \end{bmatrix} \begin{bmatrix} \sigma'_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma'_n \end{bmatrix}^{-1} \\
&= \Phi'^{[k]}\,(\Sigma')^{-1} \qquad (4.11)
\end{aligned}
$$

Substituting (4.11) in (4.10) we get:

$$
\begin{aligned}
\bar{x}^{[k]} &= X'\Phi'^{[k]}\,(\Sigma')^{-1}\,(U'_1)^T\,b \\
&= \begin{bmatrix} | & | & \\ x'_1 & x'_2 & \cdots \\ | & | & \end{bmatrix} \begin{bmatrix} \frac{\phi_1'^{[k]}}{\sigma'_1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{\phi_n'^{[k]}}{\sigma'_n} \end{bmatrix} \begin{bmatrix} (u'_1)^T\,b \\ (u'_2)^T\,b \\ \vdots \end{bmatrix} \\
&= \begin{bmatrix} | & | & \\ x'_1 & x'_2 & \cdots \\ | & | & \end{bmatrix} \begin{bmatrix} \phi_1'^{[k]}\frac{(u'_1)^T b}{\sigma'_1} \\ \phi_2'^{[k]}\frac{(u'_2)^T b}{\sigma'_2} \\ \vdots \end{bmatrix} \\
&= x'_1\phi_1'^{[k]}\frac{(u'_1)^T\,b}{\sigma'_1} + x'_2\phi_2'^{[k]}\frac{(u'_2)^T\,b}{\sigma'_2} + \ldots + x'_n\phi_n'^{[k]}\frac{(u'_n)^T\,b}{\sigma'_n} \\
&= \sum_{i=1}^{n}\phi_i'^{[k]}\frac{(u'_i)^T\,b}{\sigma'_i}x'_i \qquad (4.12)
\end{aligned}
$$

where $\phi_i'^{[k]} = 1 - \left(1 - \lambda \left(\frac{\sigma_i'}{\mu_i'}\right)^2\right)^k$.

Note the similarities and differences between the expressions from (3.8) and (4.12). One of the significant differences is that the original solution is a linear combination of the v_i vectors and the preconditioned solution is a linear combination of the x_i vectors. In the further analysis of preconditioning matrices, L, we will look into how theses affect the filter factors.

The above theory holds for classical landweber. For the other SIRT methods, a similar analysis can be carried out by replacing A with $M^{\frac{1}{2}}A$ (where M is from the particular SIRT method). We will not do so in this work.

The conclusion is again that the iterates $\bar{x}^{[k]}$ are different from $x^{[k]}$. This leaves us with a question however: *Which L-matrices should be used for preconditioning the system?* In the following we will discuss the choices of L-matrices and see how they affect the solution.

4.2 Choice of L-matrices for Preconditioning

In principle virtually any invertible matrix could be used for preconditioning the system. However it would be difficult to predict what influence it would have on the solution. In this project the focus is on matrices that approximate either the first or second derivative.

Keeping (4.1) and (4.10) in mind, we see that an L-matrix approximating a derivative means that $\bar{x}^{[k]}$ will be an integration of $\xi^{[k]}$. As we know, integration has a smoothing effect. Thereby using matrices that approximates differentiation for preconditioning, we obtain a smoothed solution. The SIRT methods have a partly smoothing property [3] and therefore they will normally be used for problems where the solution is expected to be smooth. This is why the focus has been on L-matrices that resemble discrete derivatives. A list of proposed matrices for preconditioning is found in appendix A.

If we do not assume boundary conditions for the solution the L-matrix is rectangular (see appendix A). Since L has to be square (because of the fact that it will be inverted) some boundary conditions have to be applied. Unfortunately both reflective and periodic boundary conditions lead to rank deficient matrices (see appendix A). Therefore either a zero boundary condition or insertion of some scalar α into the matrix has to be done. In the following the properties of these workarounds will be discussed.

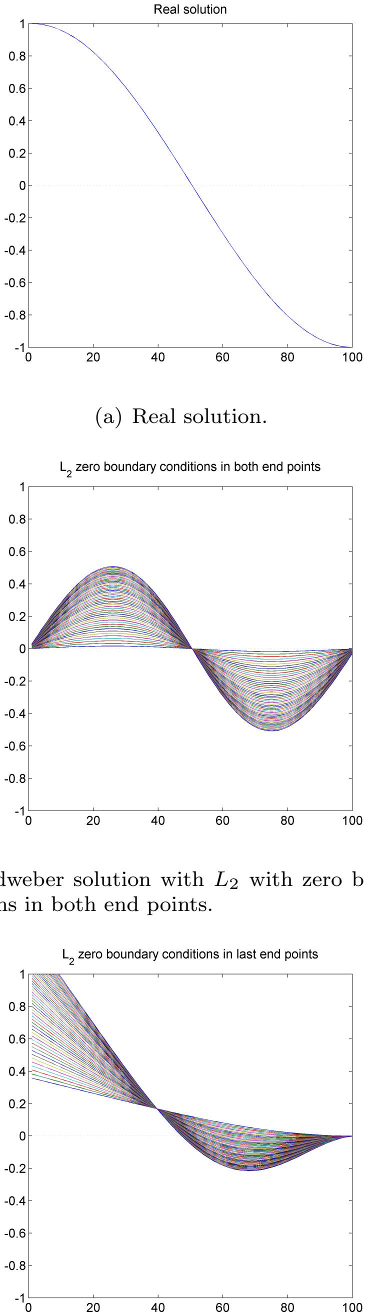
4.2.1 The Influence of Zero Boundary Conditions in L-Matrices

A zero boundary condition, seen in e.g. L_2^{zul} (A.32) or L_2^{zl} (A.30), seem to force the solution towards the starting guess on the boundary where it has been applied. Experiments suggest that these conditions in the preconditioning overrule trends in the solution. In this section we primarily look at starting guesses $x_0 = \underline{0}$. Figure 4.1(a) show the real solution to a test problem, figure 4.1(b) the Landweber solutions with zero boundary conditions to L_2 in first and last end points, and figure 4.1(c) shows the Landweber solutions with zero boundary conditions on the last end points. This perfectly illustrates how the zeros boundary conditions force the solution towards zero with the $\underline{0}$ as start guess.

If we have an a priori knowledge of the solution, e.g. that it should be zero at the last end point, we can use this to choose where to use the boundary conditions in the L-matrices. As seen before we obtain bad solutions if we apply zero conditions on a boundary where the solution is not zero. Figure 4.2 shows another problem where we cannot apply zero boundary conditions in both the first and last end points. Instead they are applied on the two last end points.

We have now seen that the zero boundary conditions seem to force the solution towards zero. However until now we have used the zero vector as a starting guess. One might think that it would behave differently with another starting guess. Figure 4.3 shows that a good starting guess at the end points with zero boundary conditions makes for good approximations to the solution in those points. Unfortunately they do not influence the other elements of the solution and therefore do not seem to be useful.

For now we will conclude that the zero boundary conditions can have a positive effect on the solution (see 4.2(c)), however we need to have an a priori knowledge of the values of the solution in at least one end. We may be able to use an alternative strategy to the zero boundary conditions. This is discussed in section 4.2.3. In the following section we will see what effect the degree of differentiation has on the solution.

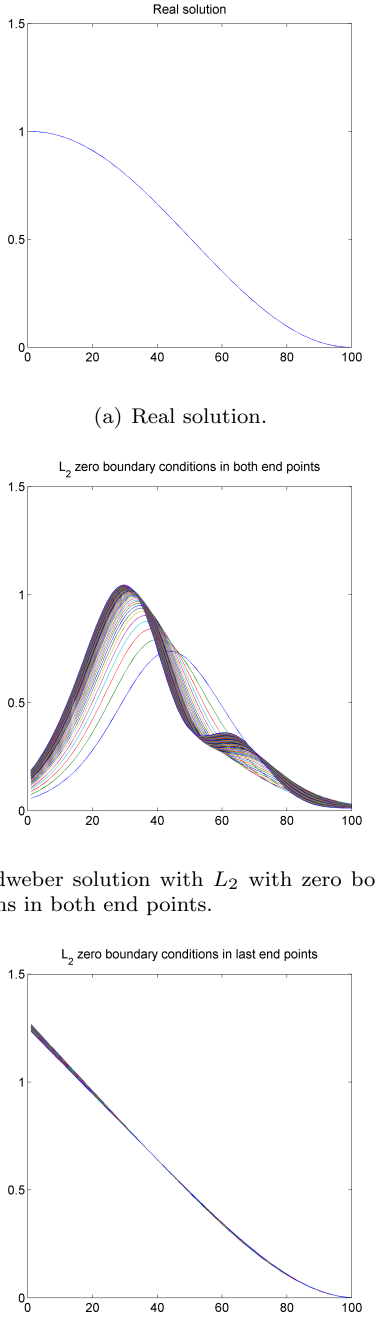


(a) Real solution.

(b) Landweber solution with L_2 with zero boundary conditions in both end points.

(c) Landweber solution with L_2 with zero boundary conditions in bottom end points.

Figure 4.1: Illustration of how the zero boundary conditions force the solution towards the starting guess and overrule the properties in the real solution.

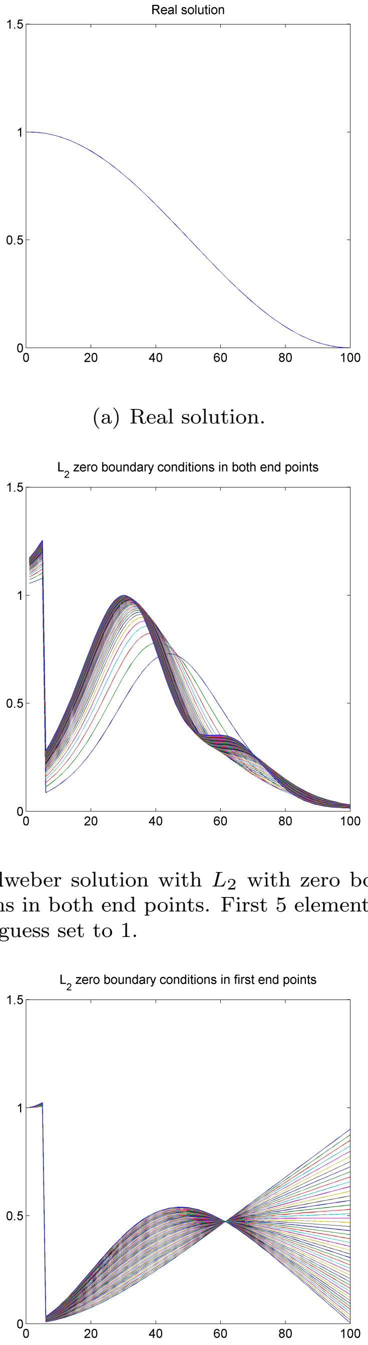


(a) Real solution.

(b) Landweber solution with L_2 with zero boundary conditions in both end points.

(c) Landweber solution with L_2 with zero boundary conditions in bottom end points.

Figure 4.2: Illustration of how the zero boundary conditions can be used to obtain a better solution.

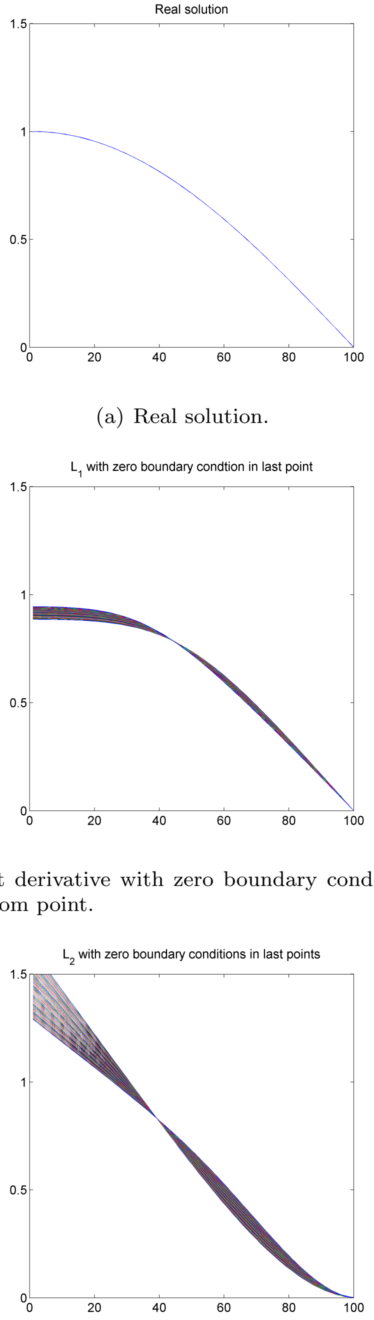


(a) Real solution.

(b) Landweber solution with L_2 with zero boundary conditions in both end points. First 5 elements of the starting guess set to 1.

(c) Landweber solution with L_2 with zero boundary conditions in top end points. First 5 elements of the starting guess set to 1.

Figure 4.3: Illustration of how the zero boundary conditions force the solution towards the starting guess and overrule the properties in the real solution. Unfortunately it does not seem to affect the other elements and therefore not very useful.

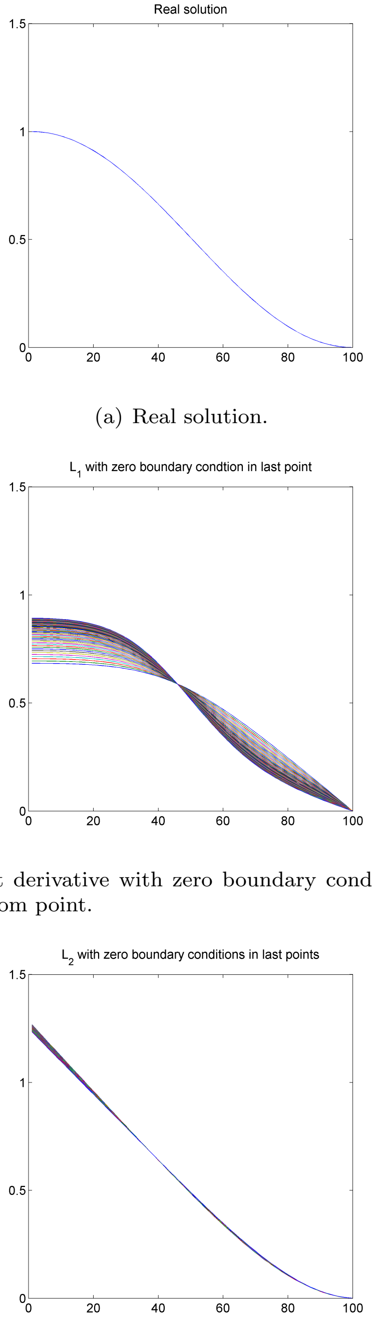


(a) Real solution.

(b) First derivative with zero boundary condition in the bottom point.

(c) Second derivative with zero boundary conditions in the bottom points.

Figure 4.4: Illustration of the fact that it does matter whether or not L_1 or L_2 are used for preconditioning the system. Here L_1 in figure 4.4(b) is preferable to L_2 in figure 4.4(c).



(a) Real solution.

(b) First derivative with zero boundary condition in the bottom point.

(c) Second derivative with zero boundary conditions in the bottom points.

Figure 4.5: Illustration of the fact that it does matter whether or not L_1 or L_2 are used for preconditioning the system. Here L_2 in figure 4.5(c) may be preferable to L_1 in figure 4.5(b).

4.2.2 The Influence of Choosing First or Second Derivative L-Matrices

We have not yet seen which effect it has whether or not we use a first or second derivative approximating L-matrix. An illustration of where the first derivative is preferable is seen in figure 4.4. In figure 4.5 one may discuss which of the L-matrices give the better solution. Undoubtedly the L_2 converges to the final solution a lot quicker than L_1. Which is better may depend on which properties of the solution is wanted and how important the number of iterations is for the user.

During experiments it has not been possible to determine some general features separating the results using either first or second derivatives. Intuitively second derivative matrices may be good for extra smooth solutions since we will then do a double integration of $\xi^{[k]}$. Unfortunately it has not been able to come up with test problems that show tendencies to general differences between the two, even though figure 4.7 supports this idea.

4.2.3 The Influence of α-Values in L-Matrices on Filter Factors

We have now seen examples of how the zero boundary conditions affect the solution by forcing it towards 0, with appropriate starting guess. In that respect it was concluded that these boundary conditions are not necessarily desired, as in figure 4.1. We will now see what effect the choice of α parameters has.

Using the same problem as in figure 4.1 a L_1-matrix with $\alpha = 10^{-3}$ in the lower right we get the filter factors as seen in figure 4.6. It is seen that the first filter factor is dominating the others, meaning that the first singular vector, x'_1 will dominate the solution (is seen from (4.12)). Of course this is an undesired property since it limits the vector-space of which the solution can be expressed. (4.6) shows that the filter factors among other depend on λ. In this project we will not investigate the choice of λ but instead use $\lambda = \frac{1}{\gamma_1'^2}$ from [4] . This gives the following expression for the second filter factor:

$$\phi_2'^{[k]} = 1 - \left(1 - \frac{1}{\gamma_1'^2}\gamma_2'^2\right)^k \tag{4.13}$$

As seen on the 1. axis of the plot in figure 4.6, the first singular value is very big compared to the second. (4.13) shows that when this is the case the term

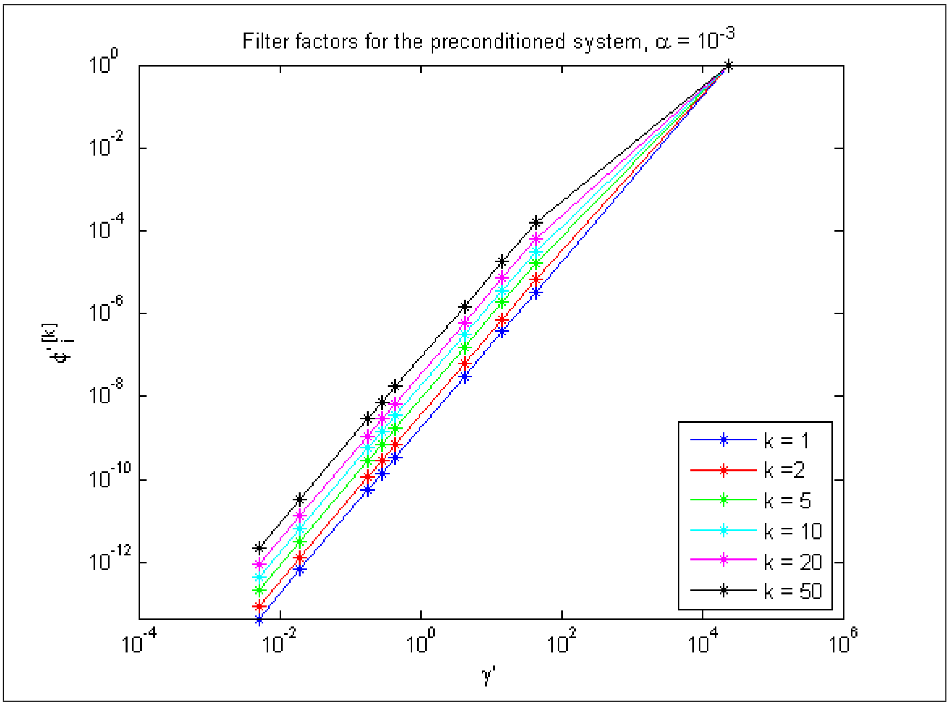


Figure 4.6: First 9 filter factors for L_1 with small α-value.

will dominate and the filter factors will stay small even with a high number of iterations, k.

To understand why this is the case we keep (4.3) and (4.4) in mind. We then take out an example with $L_1^{\alpha l}$. We then have the situation that:

$$L = \begin{bmatrix} L_{11} & l \\ 0 & \alpha \end{bmatrix}, \quad L_{11} \in \mathbb{R}^{n-1 \times n-1}, \quad l \in \mathbb{R}^{n-1 \times 1} \tag{4.14}$$

where

$$L_{11} = \begin{bmatrix} -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \\ & & & -1 \end{bmatrix}, \quad L_{11} \in \mathbb{R}^{n-1 \times n-1}$$

and

$$l = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}, \quad l \in \mathbb{R}^{n-1 \times 1}$$

Then we have that

$$L^{-1} = \begin{bmatrix} L_{11}^{-1} & -\frac{1}{\alpha}L_{11}^{-1}l \\ 0 & \alpha^{-1} \end{bmatrix} \tag{4.15}$$

From now we will define a vector $z = -\frac{1}{\alpha}L_{11}^{-1}l$. Also we will define A such that:

$$A = \begin{bmatrix} A_1 & a_2 \end{bmatrix}, \quad A_1 \in \mathbb{R}^{m\times n-1}, \quad a_2 \in \mathbb{R}^{m\times 1} \tag{4.16}$$

Now an expression for AL^{-1} can be written:

$$\begin{aligned} A' &= AL^{-1} \\ &= \begin{bmatrix} A_1 & a_2 \end{bmatrix} \begin{bmatrix} L_{11}^{-1} & z \\ 0 & \alpha^{-1} \end{bmatrix} \\ &= \begin{bmatrix} A_1 L_{11}^{-1} & A_1 z + a_2\alpha^{-1} \end{bmatrix} \end{aligned} \tag{4.17}$$

Taking a closer look at the last column we see that:

$$\begin{aligned} A_1 z + a_2\alpha^{-1} &= \alpha^{-1}a_2 - \alpha^{-1}A_l L_{11}^{-1} l \\ &= \alpha^{-1}\left(a_2 - A_1 L_{11}^{-1} l\right) \end{aligned} \tag{4.18}$$

We know that the values in A_1, a_2, L_{11} and l are of the order $\mathcal{O}\left(10^0\right)$ or lower. If α is chosen as a small number, i.e. $\mathcal{O}\left(10^{-3}\right)$ the inverse will become large. As seen from (4.17) and (4.18) this will influence the last column in AL^{-1} which will have high elements compared to the other columns.

From Gershgorin's circle theorem applied to $\left(AL^{-1}\right)^T AL^{-1}$,[1] this means that the matrix AL^{-1} will have one large singular value compared to the rest. The size of this singular value is approximately equal to the norm of the large column. This is observed in figure 4.6 where the first singular value is $\mathcal{O}\left(10^3\right)$ larger than the second.

It can be shown that AL^{-1} will have one large singular value even though the α-value is inserted in any other column. This can be shown by permutation as in section 4.2.4.

We have now seen that small α-values should not be used. It can also be concluded that large α-values will result in a column with small values which will not influence the largest singular values. We will now look into how the α influence the solution.

[1] Suggested by Per Christian Hansen, IMM DTU

4.2.4 The Influence of α-Values in L-Matrices on the Solutions

We have seen that using a small α-value influences the filter factors in a way that is not good for the solution. We will now see how a large α-value affects the solution. This will be done for the Landweber method. The similar can be done for the Cimmino method since the only difference is the M-matrix, but that will not be done here.

Applied to the Landweber iterations, (4.18) gives that:

$$\xi^{[k+1]} = \xi^{[k]} + \lambda_k \left(AL^{-1}\right)^T \left(b - AL^{-1}\xi^{[k]}\right) = \begin{array}{|c|}\hline \\ \mathcal{O}\left(1\right) \\ \\ \hline \alpha^{-1} \\ \hline \end{array} \tag{4.19}$$

$$\Rightarrow \bar{x}^{[k]} = L^{-1}\xi^{[k]} = \begin{array}{|c|c|}\hline & \\ \mathcal{O}\left(1\right) & \mathcal{O}\left(1\right) \\ & \\ \hline 0 & \alpha^{-1} \\ \hline \end{array} \; \begin{array}{|c|}\hline \\ \mathcal{O}\left(1\right) \\ \\ \hline \alpha^{-1} \\ \hline \end{array} = \begin{array}{|c|}\hline \\ \mathcal{O}\left(1\right) \\ \\ \hline \alpha^{-2} \\ \hline \end{array} \tag{4.20}$$

This shows that for large α-values in the last column, the last element in the regularized solution will be very small. We will now show what effect it will have to have the α-value in another column. Say we have the matrix:

$$L = \begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ & \cdots & \alpha & \cdots & \end{bmatrix}$$

Using a permutation matrix, P, to switch two columns:

$$P = \begin{bmatrix} 1 & 0 & & & & \\ 0 & 1 & 0 & & & \\ & & \ddots & \ddots & & \\ & & & 0 & & 1 \\ & & & & \ddots & \\ & & & 1 & & 0 \end{bmatrix}$$

We can permute the L matrix. Note that this is okay as long as we remember to switch back after the iterations are done. This gives the following:

$$\hat{L} = LP = \begin{array}{|c|c|} \hline & \\ \hat{L}_{11}^{-1} & \hat{l} \\ & \\ \hline 0 & \alpha \\ \hline \end{array} \tag{4.21}$$

From (4.20) it is now clear that last element in the permuted iterates, $\hat{\xi}^{[k]}$, will tend have very small values. However we need to do the backwards permutation to return to our original coordinate system:

$$
\begin{aligned}
\bar{x}^{[k]} &= L^{-1}\xi^{[k]} = P\hat{L}^{-1}\hat{\xi}^{[k]} \\
&= P \left[\begin{array}{c|c} \hat{L}_{11}^{-1} & \hat{z} \\ \hline 0 & \alpha^{-1} \end{array}\right] \left[\begin{array}{c} \mathcal{O}\left(1\right) \\ \hline \alpha^{-1} \end{array}\right] \\
&= P \left[\begin{array}{c} \mathcal{O}\left(1\right) \\ \hline \alpha^{-2} \end{array}\right] = \left[\begin{array}{c} \\ \hline \alpha^{-2} \\ \hline \\ \end{array}\right]
\end{aligned}
\tag{4.22}
$$

Using permutation we have now proved how the placement of α has an influence on the solution. This means that if we have some a priori knowledge about where the solution is close to zero, we can use this to place a large α-value at that corresponding column in L. An example of this can be seen in appendix A.3.

4.3 Accelerating Effects and Better Solutions

As mentioned we use L-matrices that approximate differentiation since that will have a smoothing effect on $\bar{x}$ and thereby expectedly obtain better solutions in fewer iterations on smooth problems. We have just seen how we can force the solution to be near zero at a selected place. Even though it was discovered because of practicalities with rank deficiencies, it is seen to have a very positive influence on the solution.

Figure 4.7 shows the test problem from 3.5 with two preconditioned solutions. In this case (the shaw test problem) preconditioning give worse results. In the plot a first and second derivative matrix with zero boundary conditions were used. The other of the discussed boundary conditions do not provide better results. It is worth noticing that the solution obtained with the second derivative L-matrix results in a smoother solution.

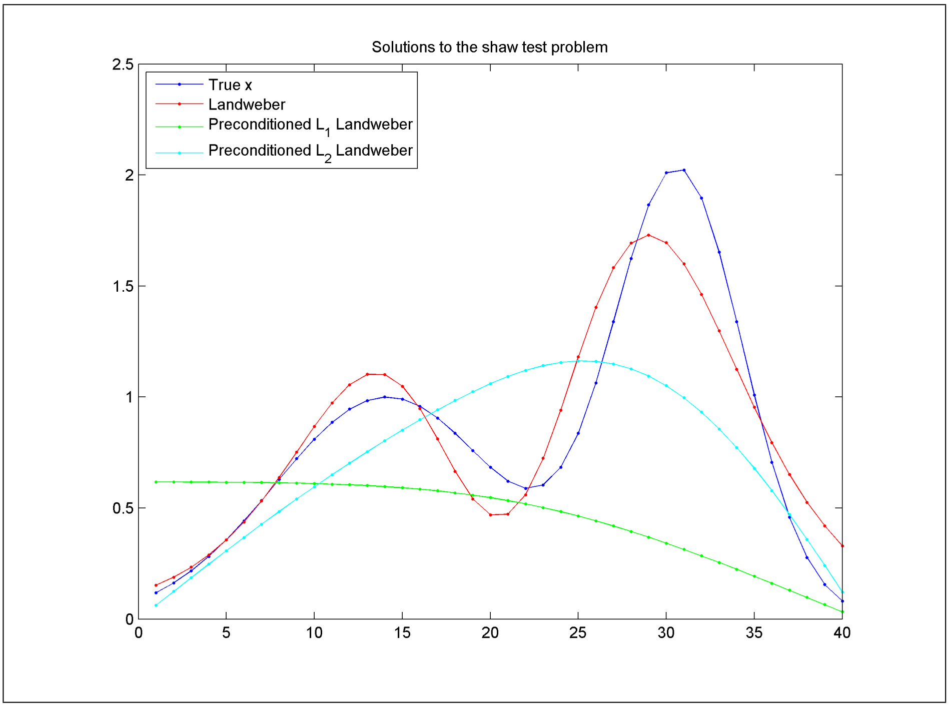
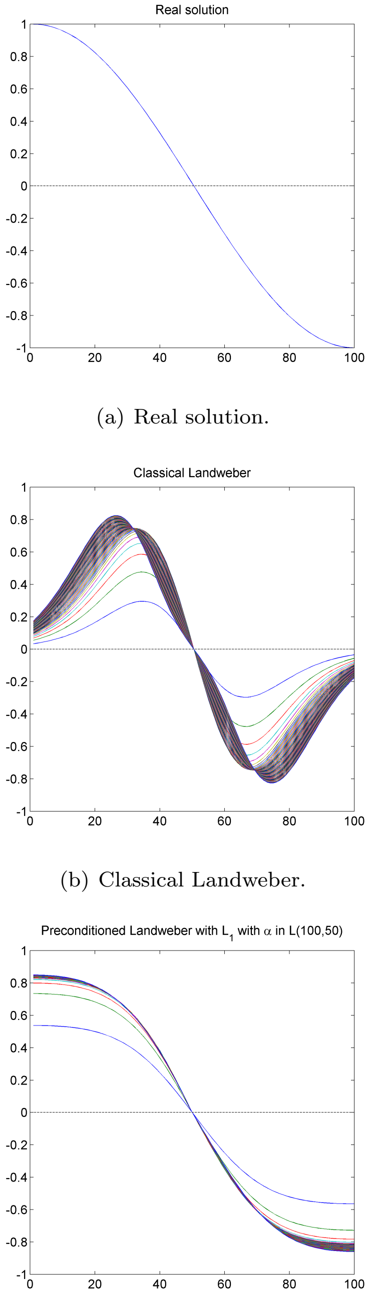


Figure 4.7: Illustration of preconditioning not suitable for the shaw problem. Zero boundary conditions applied to both L-matrices. However the result was not significantly different with other boundary conditions

In some cases the classical Landweber produce bad solutions. The tendency is to be close to zero in the end points. Sometimes this can be avoided with an a priori knowledge and usage of α when preconditioning the system. Figures 4.8 and 4.13 show when preconditioning can become very effective. Note that not only do they give much better end results, but they converge towards them in few iterations.

Figures 4.9, 4.10, 4.11 and 4.12 show the singular vectors and filter factors to the solutions in figure 4.8. It may be difficult to conclude, from the singular vectors, which set give the best results. The important thing to note is that it is clear to see that they are different and thereby may be able to span different subspaces.

From the filter factors in figures 4.10 and 4.12 it is seen that both methods primarily consist of linear combinations of their first 4 singular vectors. The 5 first singular vectors from the classical problem are all close to zero in the end



(a) Real solution.

(b) Classical Landweber.

(c) First derivative with large α in middle column.

Figure 4.8: A case where preconditioning is a better solution than the classical Landweber. The relative norm between the real and regularized solution is 0.1038.

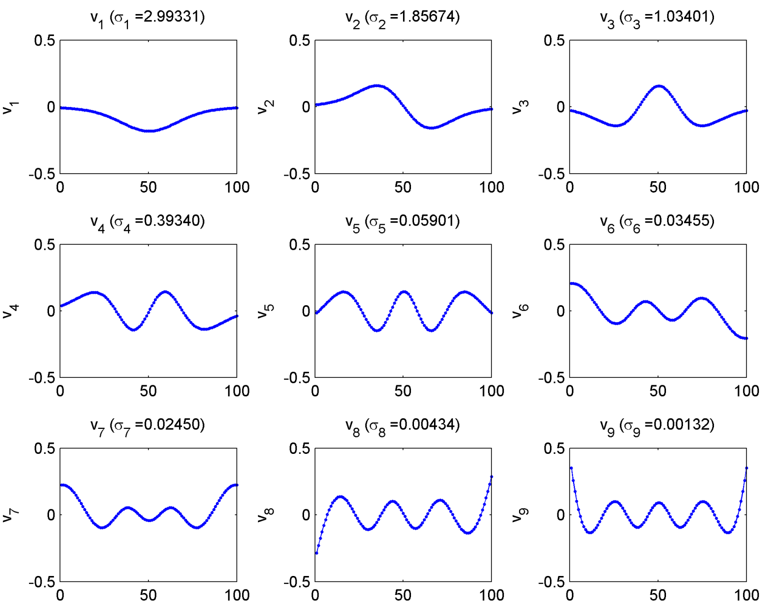


Figure 4.9: First 9 singular vectors for the classical problem with the solution in figure 4.8(b). Note that the first singular vectors tend to be zero in the end points.

points and therefore not suited for this particular solution, whereas the singular vectors from the preconditioned solution are different from zero from the 4th. However the first 3 are not suited for this particular solution.

This suggests that it may be possible to use another L-matrix to obtain more suitable singular vectors. For now we will be satisfied with showing how preconditioning changes the basis for the solution and sometimes this is a better basis for the solution.

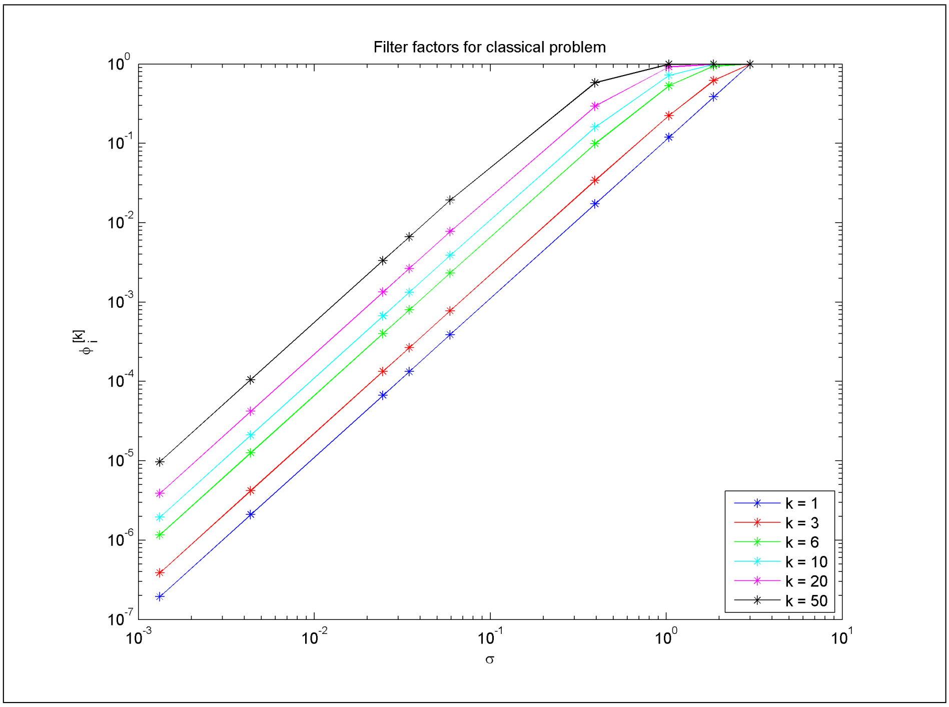


Figure 4.10: First 9 filter factors for the classical Landweber solution, with the solution in figure 4.8(b).

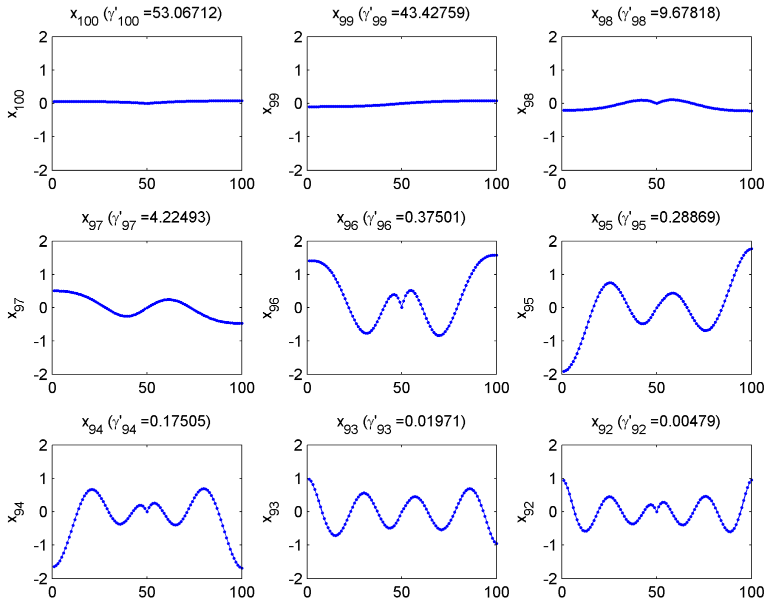


Figure 4.11: First 9 singular vectors for the regularized problem with the solution in figure 4.8(c).

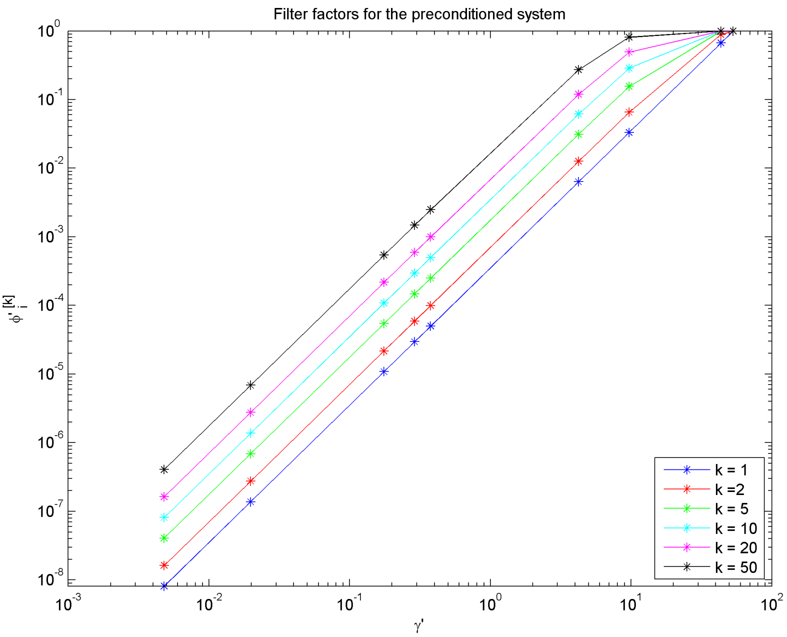
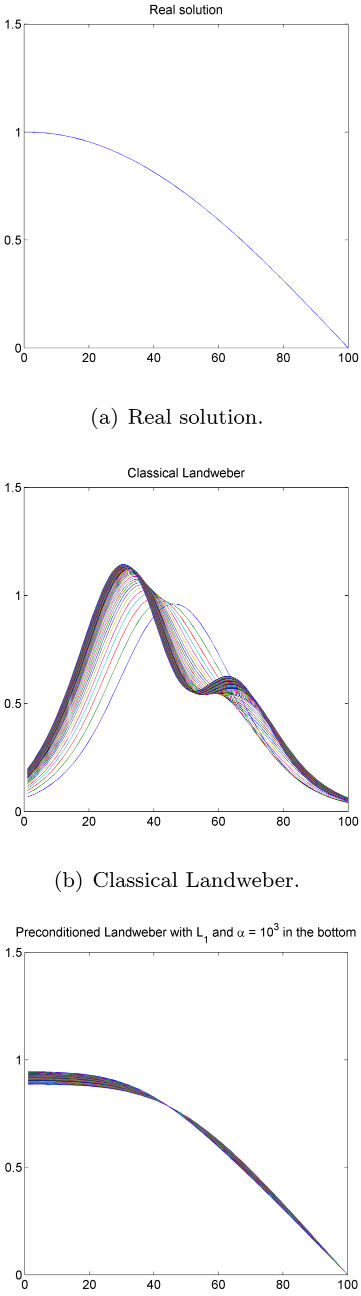


Figure 4.12: First 9 filter factors for the regularized Landweber solution, with the solution in figure 4.8(b).



(a) Real solution.

(b) Classical Landweber.

(c) First derivative with large α in last column.

Figure 4.13: A case where preconditioning is a good solution. The relative norm between the real and regularized solution is 0.0305.

Chapter 5

Effective Implementation in Matlab

We have now seen how the two SIRT methods Landweber and Cimmino can be preconditioned and how this affect their solutions. In this chapter a discussion of how to implement these methods effectively in Matlab will be taken out. A code listing of the two implementations are found in appendices C and D. Also a function for providing L-matrices has been written and is listed in appendix B.

5.1 Landweber and Cimmino

For now the implementations are only made for constant λ. This choice has been made since the theory of relaxation parameters has not been thoroughly studied in the work with this thesis. If a λ is not provided by the user the default value is set to $\lambda = \frac{1}{\gamma_1^2}$, which should be a good choice to ensure convergence [4]. For the same reasons the only stop criteria for now is a number of iterations.

Both functions have a function header explaining how to use the function. This acts as a help in Matlab . Following the header is a section of input checks and setting of default values. The comments in the code should be self explanatory.

It has been chosen that the fifth input can be either a starting guess or a relaxation parameter. Since these are different data structures it is easy to determine which is the case. After these checks and all necessary default values have been set, the iterations begin.

When implementing these functions one has to keep in mind that the problems typically will be large and sparse, as discussed in chapter 3. This means that an alternative method may be preferred, when creating the M-matrix in Cimmino. As it is now a fast method is used but this uses more memory and therefore another, slower method, is out-commented and may be used if this becomes an issue.

It also means that when calculating the largest singular value it is not a good idea to use the built-in MATLAB functions. For the functions in [4] the function SVDS is used. However a similar function does not yet exist for calculating the largest GSVD singular values. Therefore these are approximated by solving the eigenvalue problem $\left(A^T A\right) V = \left(L^T L\right) V D$. [1]

Instead of doing a lot of matrix multiplying and calculation of GSVD the MATLAB '\' operator is used on each iterate. The '\' operator solves the problem $Ax = B$.

Remembering (4.3), the preconditioned iterates (λ is left out for now) are given by:

$$\begin{aligned} \xi &= \left(A'\right)^T \left(b - A'\xi\right) \\ &= \left(AL^{-1}\right)^T \left(b - \left(AL^{-1}\right)\xi\right) \\ &= L^{-T}A^T \left(b - AL^{-1}\xi\right) \end{aligned} \tag{5.1}$$

Since $x = L^{-1}\xi$, (5.1) gives:

$$\begin{aligned} x &= L^{-1}\xi \\ &= L^{-1}L^{-T}A^T \left(b - Ax\right) \end{aligned} \tag{5.2}$$

This means that the MATLAB operator, '\', can be used on every iterate instead of multiplying and transposing L-matrices, which uses a lot of memory. We exploit the fact that the MATLAB operator is fastest for triangle matrices, since it will then just be either a forward or backward substitution. Therefore a triangle flag, TF, is set if L is a triangle matrix. If it is not, a QR-factorization is done and thereby ensuring a quick solution.

[1] Suggested by Per Christian Hansen, IMM DTU

This is a fast MATLAB implementation and we never calculate the GSVD nor filter factors. These were, however, very useful in the analysis.

5.2 get_full_l

The GET_FULL_L function uses the existing GET_L to produce the rectangular matrix. This is the sparse data structure since L by nature is very sparse. Therefore full matrices (non-sparse data structures) will use a lot more memory for large data sets.

At first glance it may seem irrelevant discussing effective implementation of GET_FULL_L since it mostly adds up to 2 rows and 4 elements into an array. However it is relevant for large data sets. That is why rows are added to the original L-matrix, instead of preallocating a completely new matrix. The following MATLAB code shows two ways to create the full sized matrix and the computation times as comments.

```
clear all
n = 10^5;

% Method 1
tic;
LL = get_l(n, 1);
L = spalloc(n, n, 3*(n-2)+4);
L(2:n, :) = LL;
toc
% 32 seconds

clear L LL

% Method 2
tic;
L = get_l(n, 1);
L = [spalloc(1,n,4); L];
toc
% 0.036 seconds
```

The latter method has been used in GET_FULL_L. Without going into details, the last method is faster for both small and large values of n, which is seen in figure 5.1.

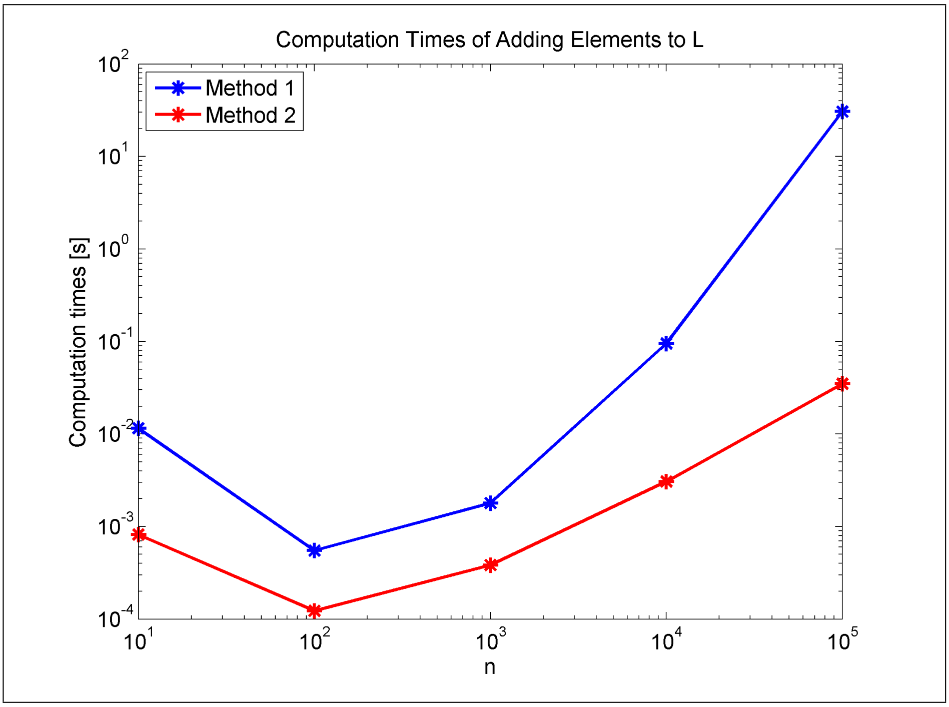


Figure 5.1: Illustration of differences in computation times between two ways of adding rows to a matrix.

The code in GET_FULL_L itself is not very complex. There is a section of input checks and setting of default values. The idea is primarily to set needed default values and do a fair amount of input checks. However there may be some situations which have not been taken into account. Further checks can easily be inserted if needed.

Because of the combinations possible for the second derivative matrices, the code may seem complex and be difficult to read. However the commenting and structure of the code is written in a way so that it should be easy to understand.

Chapter 6

Conclusion

The main goal for this thesis was to investigate how ill-posed discrete inverse problems could be preconditioned and later implemented in some SIRT methods in an existing Matlab toolbox. To do this a general study of ill-posed discrete inverse problems had to be taken out. It was then showed theoretically how the Landweber method could be preconditioned. Then an investigation of possible preconditioners were made. It turned out that the choice of preconditioners was not as straight forward as expected. However some good footwork has been done and the preconditioning of the Landweber and Cimmino methods have been implemented in Matlab .

First of all it was theoretically shown that these ill-conditioned inverse problems cannot be solved in an ordinary way, and that a regularized solution has to be calculated. This was shown using a Picard plot and plot of the filter factors. We then showed how the SIRT methods produce a regularized solution. We could now start with the actual work - preconditioning the problem.

It has clearly been shown that preconditioning can have positive effects on the SIRT methods. Preconditioning the problem leads to another set of basis vectors which tend to be significantly different. It was also shown that it does matter which preconditioner is used. When using an α-parameter in the preconditioner, we showed that small values of α is a very poor choice, since the solution will primarily be a linear combination of one singular vector because one filter factor will dominate the others. We also showed that choosing a large α-value in column i will force the ith element in the solution towards zero.

In the work with the preconditioners we found that many of the obvious preconditioners were rank deficient and therefore invalid, leading to the usage of α parameters as mentioned above. A catalogue of valid and invalid preconditioners has been made and can be used as a base for future work with preconditioning.

Lastly three Matlab functions have been made. One that can produce the preconditioners in the catalogue. Then the preconditioned Landweber and Cimmino have been implemented effectively in Matlab . During that it has been shown how a mathematical problem, sometimes is approached very differently in theory and practice. In implementing the two methods only a number of iterations is used as stopping criteria.

6.1 Future Work

The future work may include an expansion of the preconditioner catalogue. Since the differential approximations produce smooth solutions and the SIRT methods have a smoothing effect, it may useful to use integration preconditioners in some cases.

During the work with this thesis we have only looked at 1D problems, since the focus has been to investigate the preconditioning effects. Clearly the next step could be to define preconditionres for 2D and 3D. This means that the Matlab function should be expanded.

It could also be investigated what effect preconditioning has on the stopping criteria in the existing SIRT methods. Also the methods could be able to use variable relaxation parameters even though a preconditioner is used. This work has only implemented preconditioners to two of the SIRT methods. This could also be done in *CAV* and *DROP*.

Appendix A

L-Matrices For Preconditioning

This appendix holds a selection of possible L-matrices for preconditioning inverse problems. For each matrix a short description of why this is proposed. If possible an example of it's influence on the solution will be shown. Some matrices are rank deficient and therefore cannot be inverted which is needed.

In the following we have looked at the most reasonable L-matrices for preconditioning tomography systems. We found out that unfortunately some of the mathematically nice matrices do not have full rank and can therefore not be used for preconditioning, since an the matrices are inverted. Here will follow a list of which matrices have full rank:

First derivative zero boundary condition (A.7), (A.8)

First derivative with α (A.9), (A.10)

Second derivative zero boundary condition (A.30), (A.31), (A.32)

Second derivative zero boundary condition with α (A.33), (A.34), (A.35), (A.36)

Second derivative reflective boundary condition with α (A.19), (A.20), (A.21), (A.22)

Second derivative periodic boundary condition with α (A.26), (A.27), (A.28), (A.29)

Before looking into which matrices can be used and which boundary conditions could be applied, we will give a list of figures illustrating how these what these boundary conditions mean. As show (A.11) and (A.11) approximating a derivative we will run out of points. This issue can be solved by substituting with other points. Figures A.1 A.2, A.3 and A.4 show which points the different methods used in this appendix use.

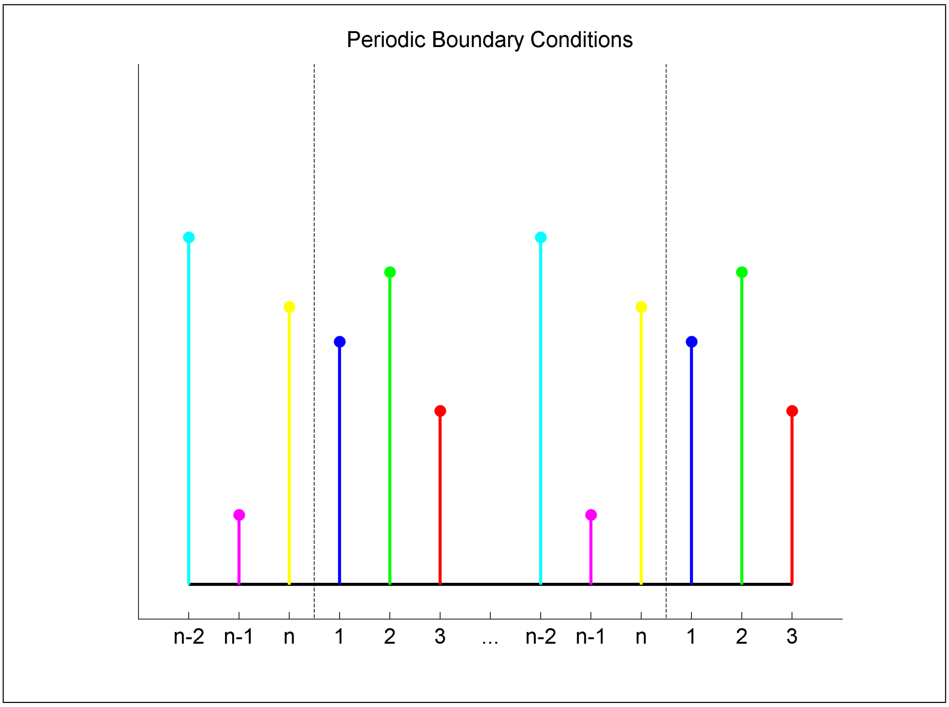


Figure A.1: Illustration of periodic boundary conditions.

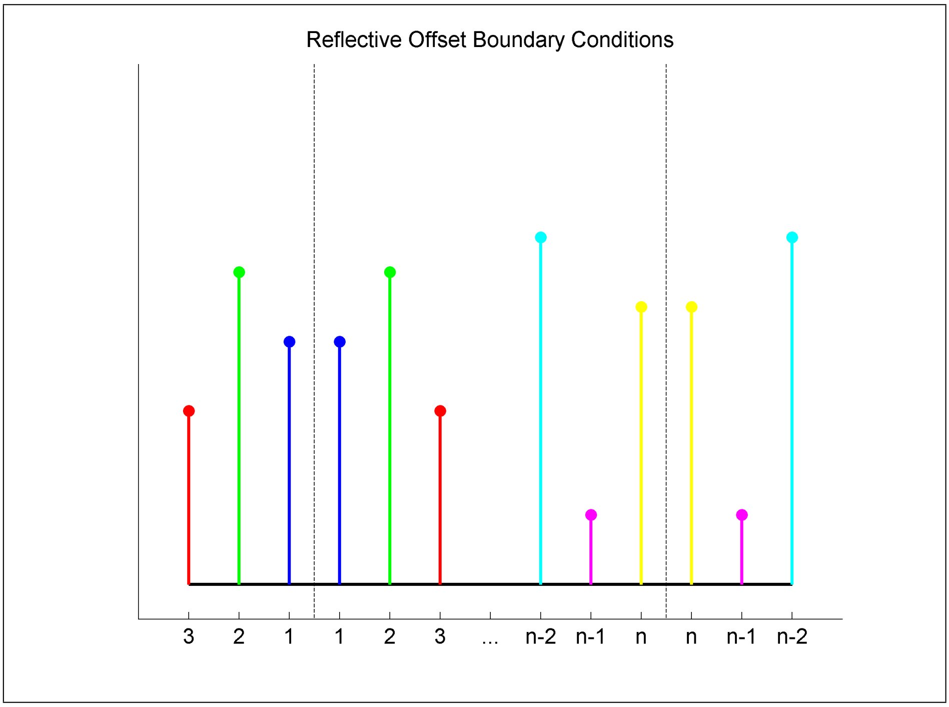


Figure A.2: Illustration of reflective offset boundary conditions.

A.1 First Derivative Matrices, L_1

The first derivative discrete approximation is calculated as:

$$f'(x) = \frac{f(x+h) - f(x)}{h} \tag{A.1}$$

For this application the $\frac{1}{h}$ can be left out since we multiply with the inverse cancelling out that term. Defining a L-matrix representing the first derivative we get a non-square matrix since we cannot approximate it from the last point.

$$L_1 = \begin{bmatrix} -1 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 1 & & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & -1 & 1 \end{bmatrix}, \quad L_1 \in \mathbb{R}^{m \times m-1} \tag{A.2}$$

A way to approximate the derivative for the last point is to set a boundary condition. One is to choose a *reflective* boundary condition which give the

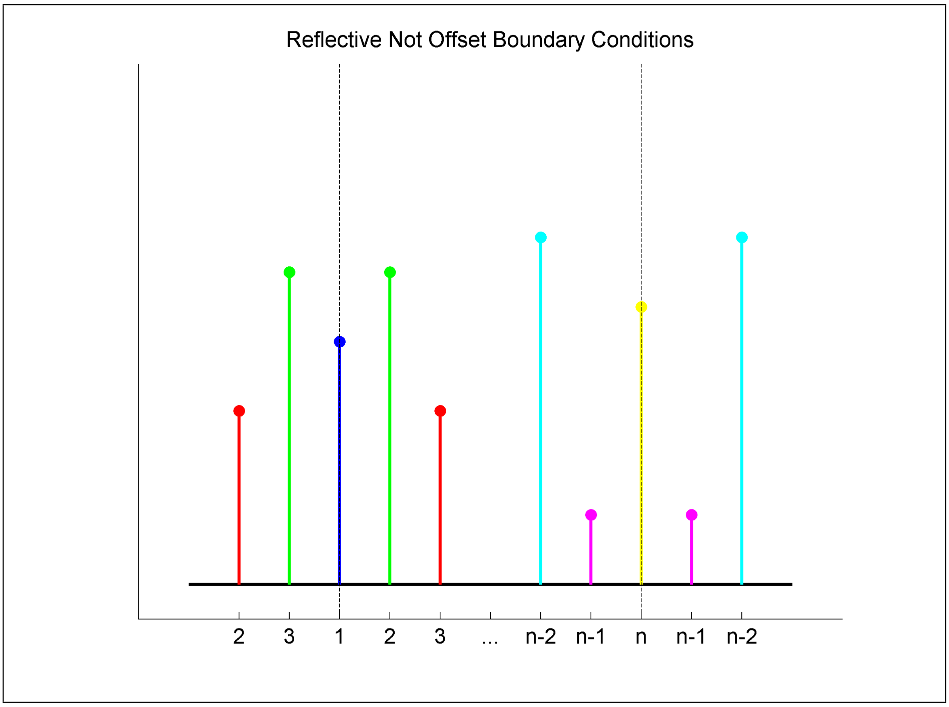


Figure A.3: Illustration of reflective not offset boundary conditions.

following matrix:

$$L_1^{rl} = \begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ & & & 1 & -1 \end{bmatrix}, \quad L_1^{rl} \in \mathbb{R}^{m\times m} \tag{A.3}$$

However L_1^r has rank $m-1$ since $r_{m-1} = -r_m$. The equivalent holds even though we insert the boundary condition in the first row:

$$L_1^{ru} = \begin{bmatrix} 1 & -1 & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{bmatrix}, \quad L_1^{ru} \in \mathbb{R}^{m\times m} \tag{A.4}$$

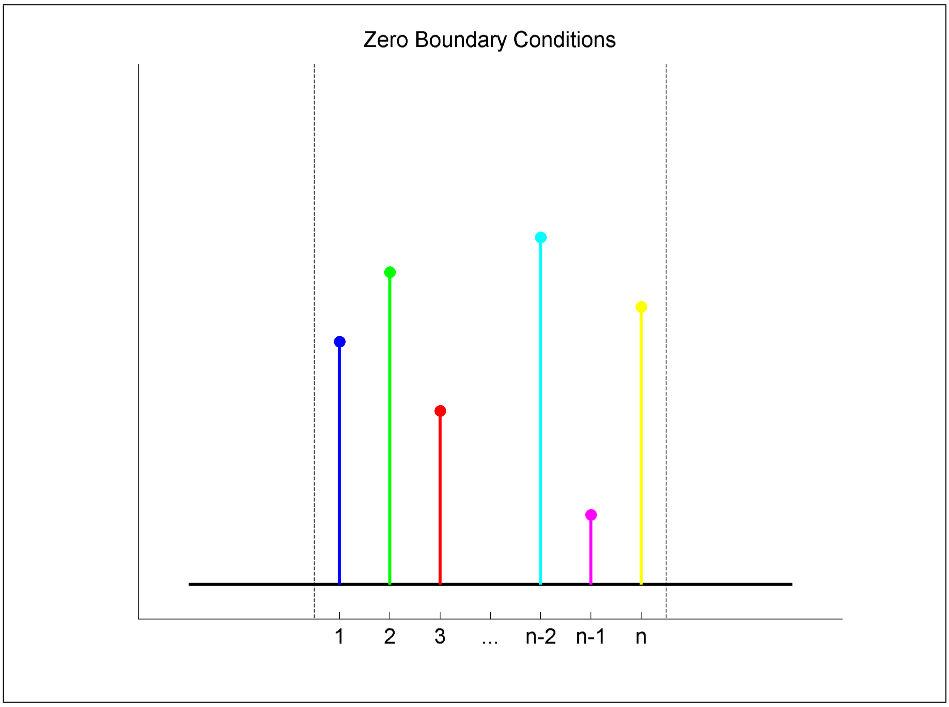


Figure A.4: Illustration of zero boundary conditions.

The L_1 with *periodic* boundary condition looks like:

$$L_1^{pl} = \begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ 1 & & & & -1 \end{bmatrix}, \quad L_1^{pl} \in \mathbb{R}^{m\times m} \tag{A.5}$$

Also L_1^{pl} has rank $m-1$ since $r_m = -r_1 - r_2 - ... - r_{m-1} = \underline{0}^T$. Again the same is the case for the periodic upper implementation:

$$L_1^{pu} = \begin{bmatrix} 1 & & & & -1 \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{bmatrix}, \quad L_1^{pu} \in \mathbb{R}^{m\times m} \tag{A.6}$$

An alternative to using either reflective or periodic boundary conditions is to assume that the boundary is zero. This means that the terms regarding the boundary can be neglected, since they will become 0 anyway:

$$L_1^{zl} = \begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ & & & & -1 \end{bmatrix}, \quad L_1^{zl} \in \mathbb{R}^{m \times m} \tag{A.7}$$

$$L_1^{zu} = \begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{bmatrix}, \quad L_1^{zu} \in \mathbb{R}^{m \times m} \tag{A.8}$$

Instead of having to use the zero boundary condition it is possible to add an arbitrary scalar α, to ensure full rank:

$$L_1^{\alpha l} = \begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \\ & & & & \alpha \end{bmatrix}, \quad L_1^{\alpha l} \in \mathbb{R}^{m \times m} \tag{A.9}$$

$$L_1^{\alpha u} = \begin{bmatrix} \alpha & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{bmatrix}, \quad L_1^{\alpha u} \in \mathbb{R}^{m \times m} \tag{A.10}$$

The choice of α has an influence on the solution as well as on the condition number of the matrix.

A.2 Second Derivative Matrices, L_2

The second derivative discrete approximation is calculated as:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} \tag{A.11}$$

Similar to the first derivatives, the term $\frac{1}{h^2}$ can be left out for this application since we multiply by the inverse and it evens out. Defining a L-matrix representing the second derivative we get a non-square matrix since we cannot approximate it from the first and last point.

$$L_2 = \begin{bmatrix} 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 1 & -2 & 1 \end{bmatrix}, \quad L_2 \in \mathbb{R}^{m \times m-2} \tag{A.12}$$

Similar to the first derivative matrices one can here use either reflective or periodic boundary conditions to get a square matrix. For the second derivative however we can either apply the boundary conditions on the both top and bottom row, two top rows or two bottom rows, i.e. applied on the first, last or both end points. Let us take a look at the different L-matrices.

For the reflective boundary conditions the mirror line can be set either right at the endpoint or $\frac{h}{2}$ after the endpoint as illustrated on figure REFERENCE!

The reflective not offset boundary conditions applied on the two bottom rows:

$$L_2^{rl} = \begin{bmatrix} 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 2 & -2 \\ & & & 1 & -2 & 1 \end{bmatrix}, \quad L_2^{rl} \in \mathbb{R}^{m \times m} \tag{A.13}$$

L_2^{rl} has rank $m-1$. It is clearly seen that $r_{m-2} = r_m$.

The reflective not offset boundary conditions applied on the two top rows:

$$L_2^{ru} = \begin{bmatrix} 1 & -2 & 1 & & & & \\ -2 & 2 & & & & & \\ 1 & -2 & 1 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 & 1 \end{bmatrix}, \quad L_2^{ru} \in \mathbb{R}^{m \times m} \tag{A.14}$$

L_2^{ru} has rank $m-1$. It is clearly seen that $r_1 = -r_3$.

The reflective not offset boundary conditions applied on the top and bottom rows:

$$L_2^{rul} = \begin{bmatrix} -2 & 2 & & & & & \\ 1 & -2 & 1 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & & 1 & -2 & 1 \\ & & & & & 2 & -2 \end{bmatrix}, \quad L_2^{rul} \in \mathbb{R}^{m \times m} \tag{A.15}$$

L_2^{rul} has rank $m-1$, since $\frac{r_1}{2} + r_2 + r_3 + ... + r_{m-1} + \frac{r_m}{2} = \underline{0}^T$.

So the reflective not offset boundary conditions produce rank deficient L-matrices. Now the reflective offset boundary conditions will be considered.

The reflective offset boundary conditions applied on the two bottom rows:

$$L_2^{rol} = \begin{bmatrix} 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2 & 1 & \\ & & & & & 1 & -1 \\ & & & & & 1 & -1 \end{bmatrix}, \quad L_2^{rol} \in \mathbb{R}^{m \times m} \tag{A.16}$$

L_2^{rol} has rank $m-1$. It is clearly seen that $r_{m-1} = r_m$.

The reflective offset boundary conditions applied on the two top rows:

$$L_2^{rou} = \begin{bmatrix} -1 & 1 & & & & & \\ -1 & 1 & & & & & \\ 1 & -2 & 1 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & & 1 & -2 & 1 & \\ & & & & 1 & -2 & 1 \end{bmatrix}, \quad L_2^{rou} \in \mathbb{R}^{m \times m} \tag{A.17}$$

L_2^{rou} has rank $m-1$. It is clearly seen that $r_1 = r_2$.

The reflective offset boundary conditions applied on the top and bottom rows:

$$L_2^{roul} = \begin{bmatrix} -1 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -1 \end{bmatrix}, \quad L_2^{roul} \in \mathbb{R}^{m \times m} \tag{A.18}$$

L_2^{roul} has rank $m-1$, since $r_1 + r_2 + ... + r_{m-1} + r_m = \underline{0}^T$.

As with the first derivatives it is bad news regarding the reflective second derivatives. Again a parameter, α, can be used to ensure the matrix has full rank. Looking into what caused the rank deficiency in the matrices, α can be inserted in either the top or bottom row. There is no motivation for inserting α e.g. r_2 in L_2^{rou}. Then it can easily be shown that this gives four matrices depending on which of the 6 are used as basis. The four matrices are:

$$L_2^{rl\alpha l} = \begin{bmatrix} 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -1 \\ & & & & & \alpha \end{bmatrix}, \quad L_2^{rl\alpha l} \in \mathbb{R}^{m \times m} \tag{A.19}$$

$$L_2^{ru\alpha u} = \begin{bmatrix} \alpha & & & & & \\ -1 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -2 & 1 \end{bmatrix}, \quad L_2^{ru\alpha u} \in \mathbb{R}^{m \times m} \tag{A.20}$$

$$L_2^{ru\alpha l} = \begin{bmatrix} -1 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & & \alpha \end{bmatrix}, \quad L_2^{ru\alpha l} \in \mathbb{R}^{m \times m} \tag{A.21}$$

$$L_2^{rl\alpha u} = \begin{bmatrix} \alpha & & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -1 \end{bmatrix}, \quad L_2^{rl\alpha u} \in \mathbb{R}^{m \times m} \tag{A.22}$$

It is noted that it has no effect whether or not the mirror line is offset, which is a nice property.

We will now look at the periodic boundary conditions, which gives three matrices, where the boundary conditions are applied either on the two bottom rows, two top rows or on the top and bottom rows, i.e. end points.

The periodic boundary conditions applied on the two bottom rows:

$$L_2^{pl} = \begin{bmatrix} 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ 1 & & & & 1 & -2 \\ -2 & 1 & & & & 1 \end{bmatrix}, \quad L_2^{pl} \in \mathbb{R}^{m \times m} \tag{A.23}$$

L_2^{pl} has rank $m-1$. It is clearly seen that $r_1 + r_2 + ... + r_{m-1} + r_m = \underline{0}^T$.

The periodic boundary conditions applied on the top and bottom rows:

$$L_2^{pu} = \begin{bmatrix} 1 & & & & 1 & -2 \\ -2 & 1 & & & & 1 \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -2 & 1 \end{bmatrix}, \quad L_2^{pu} \in \mathbb{R}^{m \times m} \tag{A.24}$$

L_2^{pu} has rank $m-1$. It is clearly seen that $r_1 + r_2 + ... + r_{m-1} + r_m = \underline{0}^T$.

The periodic boundary conditions applied on the two top rows:

$$L_2^{pul} = \begin{bmatrix} -2 & 1 & & & & 1 \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ 1 & & & & 1 & -2 \end{bmatrix}, \quad L_2^{pul} \in \mathbb{R}^{m \times m} \tag{A.25}$$

L_2^{pul} has rank $m-1$. It is clearly seen that $r_1 + r_2 + ... + r_{m-1} + r_m = \underline{0}^T$.

The periodic boundary conditions are naturally very alike no matter which end points they are applied. Using the α parameter to ensure full rank, we get the following four options:

$$L_2^{pl\alpha l} = \begin{bmatrix} 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ 1 & & & & 1 & -2 \\ & & & & & \alpha \end{bmatrix}, \quad L_2^{pl\alpha l} \in \mathbb{R}^{m \times m} \tag{A.26}$$

$$L_2^{pu\alpha u} = \begin{bmatrix} \alpha & & & & & \\ -2 & 1 & & & & 1 \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -2 & 1 \end{bmatrix}, \quad L_2^{pu\alpha u} \in \mathbb{R}^{m \times m} \tag{A.27}$$

$$L_2^{pu\alpha l} = \begin{bmatrix} -2 & 1 & & & & 1 \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & & \alpha \end{bmatrix}, \quad L_2^{pu\alpha l} \in \mathbb{R}^{m \times m} \tag{A.28}$$

$$L_2^{pl\alpha u} = \begin{bmatrix} \alpha & & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ 1 & & & & 1 & -2 \end{bmatrix}, \quad L_2^{pl\alpha u} \in \mathbb{R}^{m\times m} \tag{A.29}$$

Just as for the first derivative we can use the zero boundary condition, giving the following three full rank matrices:

$$L_2^{zl} = \begin{bmatrix} 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \\ & & & & & 1 \end{bmatrix}, \quad L_2^{zl} \in \mathbb{R}^{m\times m} \tag{A.30}$$

$$L_2^{zu} = \begin{bmatrix} 1 & & & & & \\ -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -2 & 1 \end{bmatrix}, \quad L_2^{zu} \in \mathbb{R}^{m\times m} \tag{A.31}$$

$$L_2^{zul} = \begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix}, \quad L_2^{zul} \in \mathbb{R}^{m\times m} \tag{A.32}$$

Also the zero boundary conditions can be used together with an α parameter, giving four matrices:

$$L_2^{zl\alpha l} = \begin{bmatrix} 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \\ & & & & & \alpha \end{bmatrix}, \quad L_2^{zl\alpha l} \in \mathbb{R}^{m \times m} \tag{A.33}$$

$$L_2^{zu\alpha u} = \begin{bmatrix} \alpha & & & & & \\ -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -2 & 1 \end{bmatrix}, \quad L_2^{zu\alpha u} \in \mathbb{R}^{m \times m} \tag{A.34}$$

$$L_2^{zu\alpha l} = \begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & & \alpha \end{bmatrix}, \quad L_2^{zu\alpha l} \in \mathbb{R}^{m \times m} \tag{A.35}$$

$$L_2^{zl\alpha u} = \begin{bmatrix} \alpha & & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix}, \quad L_2^{zl\alpha u} \in \mathbb{R}^{m \times m} \tag{A.36}$$

A.3 A priori based L-matrices

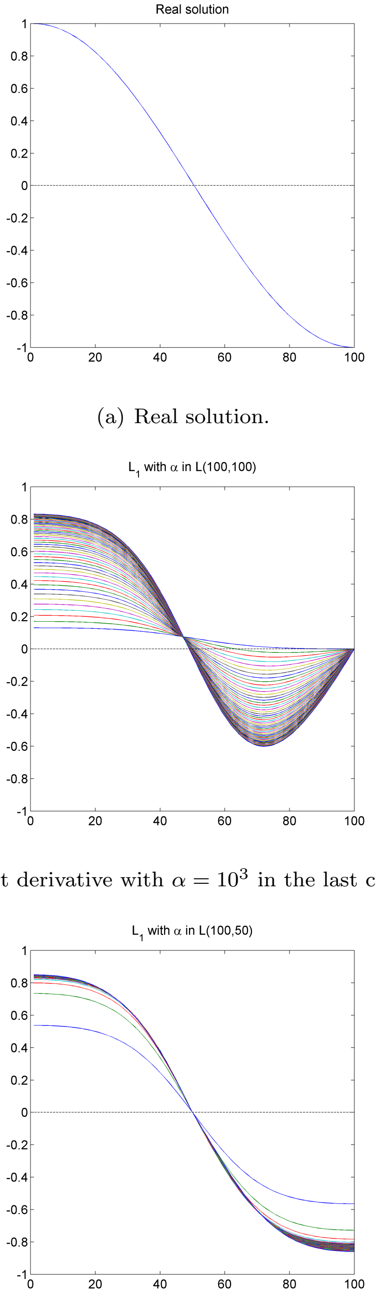
In the preceding sections some L-matrices has been chosen as approximations to first and second derivatives. Because of rank deficiency with the classical boundary conditions some α-values has been used as a workaround. Naturally these have been put at the end points since these are difficult to estimate.

However it has been shown section 4.2.4 that the choice of α-value can force the solution to be close to 0 at that point. That is if an $\alpha = \mathcal{O}\left(10^3\right)$ is inserted in the first column of L, the first element in the solution will roughly become 0.

This means that an a priori knowledge of the solution is useful when choosing L-matrices, e.g. if we know the solution should be zero in the last end point, it is a good idea to chose a L-matrix with an "large" α-value in the last column. However this knowledge of the influence of α can also be used when the solution is not 0 on the boundaries, but perhaps in the middle. In that case the α-parameter should be placed in the middle column. Not only do we obtain a good solution but in few iterations.

To illustrate this we consider the cosine test-problem in figure A.5. It is clearly seen that the knowledge of where the solution is 0 is basis for a much better solution. In the example the (A.9) and (A.37) was used.

$$L_1^{\alpha l50} = \begin{bmatrix} -1 & 1 & & & & \\ & -1 & 1 & & & \\ & & \ddots & \ddots & & \\ & & & -1 & 1 & \\ & \cdots & \alpha & \cdots & & \end{bmatrix}, \quad L_1^{\alpha l50} \in \mathbb{R}^{100\times100} \qquad \text{(A.37)}$$



(a) Real solution.

(b) First derivative with $\alpha = 10^3$ in the last column.

(c) First derivative with $\alpha = 10^3$ in the middle column.

Figure A.5: Illustration of how the placement of the α-value can be used when there is an a priori knowledge of the solution. The only difference in the two solutions is that α is in the last column or the middle column.

Appendix B

get_full_l.m

Contents

```
function L = get_full_l(n, d, ul, alpha, col, zrp)
```

```
% GET_FULL_L Computes discrete derivative operators.
%
%   L = get_full_l(n, d)
%   L = get_full_l(n, d, ul, alpha)
%   L = get_full_l(n, d, ul, alpha, col)
%   L = get_full_l(n, d, ul, alpha, col, zrp)
%
% Computes the discrete approximation L to the derivative operator
% of order d on a regular grid with n points.
%
% Input:
%   n          Dimensions of L. dim(L) = n x n.
%   d          Degree of differentiation (1 or 2).
%   ul          Value to determine if boundary conditions should be applied
%               in the upper or lower row. 1 for upper, 2 for lower and 3
%               for both (i.e. zero boundary conditions for 2. derivative).
%               For 2. derivative ul can be an array. The the first element
%               specifies where to put the boundary conditions and the
%               second element specificies where to put alpha.
%   alpha       The alpha value. If alpha = 0 the zero boundary condition is
%               applied.
%   col         Integer specifieing which column the alpha parameter should
%               be placed.
%   zrp         Zero, reflective or periodic boundary conditions when using
%               alpha for 2. derivative matrices. 0 for zero, 1 for
%               reflective and 2 for periodic.
% Output:
%   L           Sparse matrix approximating the derivative operator of
%               degree d.
%
% Default values:
%               d = 1
%               ul = 2
%               alpha = 0
%               col = 1 if ul = 1 and col = n if ul = 2
%               zrp = 0

% Lars Holtse Bonde and Per Christian Hansen, July 2011, IMM DTU.
```

Input checks and initilization

```
% Cheking n
if nargin < 1
    error('Too few inputs')
elseif n <= 0 || round(n) ~= n
    error('n should be a positive integer')
end

% Checking d or applying default value
if nargin < 2
    d = 1;
elseif d <= 0 || round(d) ~= d || ~any(d == [1, 2])
    error('d should be either 1 or 2')
end
```

```
% Checking ul or applying default value
if nargin < 3
    ul = 2;
elseif (length(ul) == 1 && ~any(ul == [1,2,3])) || ...
        (length(ul) == 2 && (~any(ul(1) == [1,2,3])) || ...
        ~any(ul(1) == [1,2,3]))
    error('elemnts in ul should be either 1, 2 or 3')
end
if ~any(length(ul(:)) == [1, 2])
    error('ul should be either scalar or array of length 2')
end
if length(ul) == 2 && (~any(ul(1) == [1, 2]) || ~any(ul(2) == [1, 2]))
    error('elements in ul do not match')
end
if length(ul) ~= 1 && d ~= 2
    error('cannot have multiple elements in ul for first derivative')
end
if d == 1 && ul == 3
    error('cannot have ul = 3 when d = 1')
end
% Ensuring ul to be a scalar if d == 1 and an array if d == 2
if d == 1
    ul = ul(1);
elseif d == 2 && length(ul) == 1
    ul = [ul, ul];
end

% Checking alpha or applying default value
if nargin < 4
    alpha = 0;
elseif length(alpha(:)) ~= 1
    error('alpha should be a scalar')
end

% Checking col or applying default value
if nargin < 5
    if ul == 1
        col = 1;
    else
        col = n;
    end
elseif col <= 0 || col > n ||col ~= round(col)
    error('col should be an integer in the range [1; n]')
end
```

```
% Checking zrp or applying default value
if nargin < 6
    zrp = 0;
end
```

Creating the matrix

```
% Obtaining the basic matrix using GET_L.
L = get_l(n, d);

% 1. derivative.
if d == 1
    % If boundary conditions in top row.
    if ul == 1
        L= [spalloc(1, n, 1); L];
        % If an alpha value, it is set. Otherwise using zero boundary
        % conditions.
        if alpha
            L(1, col) = alpha;
        else
            L(1, 1) = 1;
        end
    % If boundary conditions in bottom row.
    else
        L = [L; spalloc(1, n, 1)];
        L(n-1, n) = -1;
        % If an alpha value, it is set. Otherwise using zero boundary
        % conditions.
        if alpha
            L(n, col) = alpha;
        else
            L(n, n) = -1;
        end
    end
% 2. derivative.
else
    % If boundary conditions in top rows.
    if ul(1) == 1 && ul(2) == 1
        L = [spalloc(2, n, 4); L];
        % If an alpha value is set
        if alpha
```

```
            L(1, col) = alpha;
            % If reflective boundary condition
            if zrp == 1
                L(2, 1) = -1;
                L(2, 2) = 1;
            % If periodic boundary condition
            elseif zrp == 2
                L(2, 1) = -2;
                L(2, 2) = 1;
                L(2, n) = 1;
            % If zero boundary condition
            else
                L(2, 1) = -2;
                L(2, 2) = 1;
            end
        % If no alpha it must be zero boundary conditions in 1. and 2. row
        else
            L(1, 1) = 1;
            L(2, 1) = -2;
            L(2, 2) = 1;
        end
    % If boundary conditions in bottom rows.
    elseif ul(1) == 2 && ul(2) == 2
        L = [L; spalloc(2, n, 4)];
        % If alpha value is set
        if alpha
            L(n, col) = alpha;
            % If reflective boundary condition
            if zrp == 1
                L(n-1, n-1) = 1;
                L(n-1, n) = -1;
            % If periodic boundary condition
            elseif zrp == 2
                L(n-1, 1) = 1;
                L(n-1, n-1) = 1;
                L(n-1, n) = -2;
            % If zero boundary condition
            else
                L(n-1, n-1) = 1;
                L(n-1, n) = -2;
            end
        % If no alpha it must be zero boundary conditions in last two rows
        else
            L(n-1, n-1) = 1;
            L(n-1, n) = -2;
            L(n, n) = 1;
        end
    % If boundary conditions in both top and bottom rows.
    else
        L = [spalloc(1, n, 3); L; spalloc(1, n, 3)];
        % If an alpha value is set
        if alpha
            % If alpha is in top row
            if ul(2) == 1
                L(1, col) = alpha;
                % If reflective boundary condition
                if zrp == 1
                    L(n, n-1) = 1;
                    L(n, n) = -1;
                % If periodic boundary condition
                elseif zrp == 2
                    L(n, 1) = 1;
                    L(n, n-1) = 1;
                    L(n, n) = -2;
                % If zero boundary condition
                else
                    L(n, n-1) = 1;
                    L(n, n) = -2;
                end
            % Else alpha must be in bottom row
            else
                L(n, col) = alpha;
                % If reflective boundary condition
                if zrp == 1
                    L(1, 1) = -1;
                    L(1, 2) = 1;
                % If periodic boundary condition
                elseif zrp == 2
                    L(1, 1) = -2;
                    L(1, 2) = 1;
                    L(1, n) = 1;
                % If zero boundary condition
                else
                    L(1, 1) = -2;
                    L(1, 2) = 1;
                end
            end
        % If no alpha it must be zero boundary conditions in first and
        % last row
        else
            L(1, 1) = -2;
            L(1, 2) = 1;
            L(n, n-1) = 1;
            L(n, n) = -2;
        end
    end
end


end % function end
```

Appendix C

precondlandweber.m

Contents

- Input checks, initilization and preperation
- The landweber iterations

```
function X = precondlandweber(A, L, b, K, x0, lambda)


% PRECONDLANDWEBER Preconditioned Landweber method.
%
%   X = precondlandweber(A, L, b, K)
%   X = precondlandweber(A, L, b, K, x0)
%   X = precondlandweber(A, L, b, K, lambda)
%   X = precondlandweber(A, L, b, K, x0, lambda)
%
% Implements the preconditioned Landweber iteration for the linear system
% Ax = b, using constant lambda.
%
%       xi^{k+1} = x^k + lambda*L^(-1)*L^(-T)*A^T*(b-A*x^k)
%
% Input:
```

```
%   A          m times n matrix.
%   L          n times n matrix for preconditioning the system. Assumed
%              created using GET_FULL_L.
%   b          m times 1 vector containing the right-hand side.
%   K          Number of iterations. If K is a scalar, then K is the maximum
%              number of iterations and only the last iterate is saved.
%              If K is a vector, then the largest value in K is the maximum
%              number of iterations and all iterates corresponding to the
%              values in K are saved.
%   x0         n times 1 starting vector. Default: x0 = 0.
%   lambda    The relaxation parameter. Has to be a scalar.
% Output:
%   X          Matrix containing the saved iterations.
%
% Notes:
% * If 5 inputs and fifth input is a scalar it is assumed to be lambda.
%   Otherwise it assumed to be x0.
% * Accepts b and x0 even though they are not column vectors, but
%   converts them.
% * Accepts if elements in K are not sorted, but then sorts them.

% Lars Holtse Bonde, Maria Saxild-Hansen and Per Christian Hansen, July
% 2011, IMM DTU
```

Input checks, initilization and preperation

```
% Enough input arguments
if nargin < 4
    error('Too few inputs')
end

% L square and full rank
[mL, nL] = size(L);
if mL ~= nL
    error('L must be square')
elseif rank(L) ~= mL
    error('L must be have full rank')
end

% A matching L
[mA, nA] = size(A);
if nA ~= mL
```

```
    error('Number of coloumns in A must match dimension of L')
end

% A matching b
b = b(:);
if length(b) ~= mA
    error(['Size of A and b do not match (number of rows in A must ' ...
        'be equal to length of b'])
end

% Values used to estimate if lambda is ok, or create default value
% Approximating gamma(1) since that is all we need, and eases
% computation for large problems
gamma = sqrt(eigs(A'*A, L'*L, 1, 'LM', struct('disp', 0)));

% Default x0 or checking if x0 matches A.
% Checking if x0 is to be interpreted as lambda.
if nargin < 5
    x0 = zeros(nA,1);
    lambda = 1/gamma^2;
% If 5 inputs, check if x0 is not lambda and create lambda
elseif nargin < 6 && (size(x0,1) ~= 1 ||size(x0,2) ~= 1)
    % Checking x0
    x0 = x0(:);

    % Creating default lambda
    lambda = 1/gamma^2;
% If 5 inputs and x0 is lambda, specify it and create default x0
elseif nargin < 6
    lambda = x0;
    x0 = zeros(nA,1);
else
    x0 = x0(:);
end
% Otherwise 6 inputs and lambda is already defined.

% Checking x0
if length(x0) ~= nA
    error('Length of x0 does not match number of coloumns in A')
end

% Checking if lambda is within interval from MSH
if lambda < 0 || lambda > 2/gamma^2
    warning('MATLAB:UnstableRelaxationParameter',...
        'lambda is outside the suggested interval [0; 2/gamma^2]')
end

% Checking dimensions of and elements in K
if length(size(K)) ~= 2
    error('K must be either scalar or vector')
else
    K = K(:);
    % Checking if K has duplicate values
    K = sort(K, 'ascend');
    for i = 2:length(K)
        if K(i-1) == K(i)
            error('K has duplicate values')
        end
    end

end

% Creating X to hold iterate solutions
X = zeros(nA, length(K));

% Checking if L is a triangle matrix
if isequal(L,triu(L)) || isequal(L,tril(L))
    TF = 1;
    LT = L';
else
    TF = 0;
    R = triu(qr(L));
    RT = R';
end

% Preparring iterate values
AT = A';
xk = x0;
i = 0;

% Clearing unused values
clear mL nL mA nA gamma x0
```

The landweber iterations

```
for k = 1:K(end)
```

```
    h = AT*(b-A*xk);
    if TF
        h = L\(LT\h);
    else
        h = R\(RT\h);
    end
    xk = xk + lambda*h;

    % Checking if xk should be saved
    if any(k == K)
        i = i+1;
        X(:,k) = xk;
    end
end


end % function end
```

Appendix D

precondcimmino.m

Contents

```
function X = precondcimmino(A, L, b, K, x0, lambda)
```

```
% PRECONDCIMMINO Preconditioned Cimmino method.
%
%   X = precondcimmino(A, L, b, K)
%   X = precondcimmino(A, L, b, K, x0)
%   X = precondcimmino(A, L, b, K, lambda)
%   X = precondcimmino(A, L, b, K, x0, lambda)
%
% Implements the preconditioned Cimmino iteration for the linear system
% Ax = b, using constant lambda.
%
%       xi^{k+1} = x^k + lambda*L^(-1)*L^(-T)*A^T*(M.*(b-A*x^k))
%
% Input:
```

```
%   A          m times n matrix.
%   L          n times n matrix for preconditioning the system. Assumed
%              created using GET_FULL_L.
%   b          m times 1 vector containing the right-hand side.
%   K          Number of iterations. If K is a scalar, then K is the maximum
%              number of iterations and only the last iterate is saved.
%              If K is a vector, then the largest value in K is the maximum
%              number of iterations and all iterates corresponding to the
%              values in K are saved.
%   x0         n times 1 starting vector. Default: x0 = 0.
%   lambda    The relaxation parameter. Has to be a scalar.
% Output:
%   X          Matrix containing the saved iterations.
%
% Notes:
% * If 5 inputs and fifth input is a scalar it is assumed to be lambda.
%   Otherwise it assumed to be x0.
% * Accepts b and x0 even though they are not column vectors, but
%   converts them.
% * Accepts if elements in K are not sorted, but then sorts them.

% Lars Holtse Bonde, Maria Saxild-Hansen and Per Christian Hansen, July
% 2011, IMM DTU
```

Input checks, initilization and preperation

```
% Enough input arguments
if nargin < 4
    error(’Too few inputs’)
end

% L square and full rank
[mL, nL] = size(L);
if mL ~= nL
    error(’L must be square’)
elseif rank(L) ~= mL
    error(’L must be have full rank’)
end

% A matching L
[mA, nA] = size(A);
if nA ~= mL
```

```
    error('Number of coloumns in A must match dimension of L')
end

% A matching b
b = b(:);
if length(b) ~= mA
    error(['Size of A and b do not match (number of rows in A must ' ...
        'be equal to length of b'])
end

% Values used to estimate if lambda is ok, or create default value
% Approximating gamma(1) since that is all we need, and eases
% computation for large problems
gamma = sqrt(eigs(A'*A, L'*L, 1, 'LM', struct('disp', 0)));

% Default x0 or checking if x0 matches A.
% Checking if x0 is to be interpreted as lambda.
if nargin < 5
    x0 = zeros(nA,1);
    lambda = 1/gamma^2;

% If 5 inputs, check if x0 is not lambda and create lambda
elseif nargin < 6 && (size(x0,1) ~= 1 ||size(x0,2) ~= 1)
    % Checking x0
    x0 = x0(:);
    % Creating default lambda
    lambda = 1/gamma^2;

% If 5 inputs and x0 is lambda, specify it and create default x0
elseif nargin < 6
    lambda = x0;
    x0 = zeros(nA,1);
else
    x0 = x0(:);
end
% Otherwise 6 inputs and lambda is already defined.

% Checking x0
if length(x0) ~= nA
    error('Length of x0 does not match number of coloumns in A')
end

% Checking if lambda is within interval from MSH
if lambda < 0 || lambda > 2/gamma^2
```

```
    warning('MATLAB:UnstableRelaxationParameter',...
        'lambda is outside the suggested interval [0; 2/gamma^2]')
end

% Checking dimensions of and elements in K
if length(size(K)) ~= 2
    error('K must be either scalar or vector')
else
    K = K(:);
    % Checking if K has duplicate values
    K = sort(K, 'ascend');
    for i = 2:length(K)
        if K(i-1) == K(i)
            error('K has duplicate values')
        end
    end

end

% Creating X to hold iterate solutions
X = zeros(nA, length(K));

% Checking if L is a triangle matrix
if isequal(L,triu(L)) || isequal(L,tril(L))
    TF = 1;
    LT = L';
else
    TF = 0;
    R = triu(qr(L));
    RT = R';
end

% Preparring iterate values

% Caculating the norm of each row in A. This calculation can require a
% lot of memory. The commented lines can be used instead. They are
% slower, but uses less memory!
AL = A/L;
normAi = full(abs(sum(AL.*AL,2)));
%normAi = zeros(m,1);
%for i = 1:m
%    ai = full(AL(i,:));
%    normAi(i) = norm(ai)^2;
%end
```

```
% Defining the M matrix.
M = 1/mA*(1./normAi);
I = (M == Inf);
M(I) = 0;

AT = A';
xk = x0;
i = 0;

% Clearing unused values
clear mL nL mA nA gamma x0
```

The cimmino iterations

```
for k = 1:K(end)
    h = AT*(M.*(b-A*xk));
    if TF
        h = L\(LT\h);
    else
        h = R\(RT\h);
    end
    xk = xk + lambda*h;

    % Checking if xk should be saved
    if any(k == K)
        i = i+1;
        X(:,k) = xk;
    end
end


end % function end
```

Bibliography

[1] Lars Eldèn. *Matrix Methods in Data Mining Pattern Recognition.* SIAM, Philadelphia, 2007.

[2] Lars Eldèn, Linde Wittmeyer-Koch, and Hans Bruun Nielsen. *Introduction to Numerical Computation - analysis and* MATLAB *illustrations.* Studentlitteratur, Lund, 2004.

[3] Per Christian Hansen. *Discrete Inverse Problems - Insight and Algorithms.* SIAM, Philadelphia, 2010.

[4] Maria Saxild-Hansen. *AIR Tools - A* MATLAB *Package for Algebraic Iterative Reconstruction Techniques.* DTU Informatics, Department of Informatics and Mathematical Modeling, Kongens Lyngby, 2010.