Iterative methods for large-scale problems in 3D-tomography

Christian True

Kongens Lyngby 2011
IMM-PHD-2011-70

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-PHD: ISSN 0909-3192

Summary

In this thesis we compare different iterative methods to find pros and cons for both 2D and 3D problems. The problems will be defined as a tomography test problem, and an introduction to the theory is presented. Throughout the thesis we will define new measurement methods to determine if a reconstruction is good or bad, and we will look at certain constraints to investigate whether we should use them.

Both noise and error will be simulated in a feasible way to see if the methods can handle the noise and error, and an investigation of limiting- or extreme - values will be considered to determine what is needed to secure a good reconstruction.

Also we will consider different stopping criteria and examine how they influence the reconstruction obtained from the iterative methods. We will use our knowledge of the grains to formulate a new stopping criterion, and test if this is applicable.

Resumé

I dette eksamensprojekt sammenligner vi forskellige iterative metoder for at finde fordele og ulemper ved anvendelse på 2D og 3D problemer. Opgaverne vil blive defineret som et tomografi test problem, og en kort indførsel i teorien gives. I projektet definerer vi nye målemetoder til at afgøre om en rekonstruktion er brugbar eller ej, og vi indfører grænser til bestemmelse af brugbarheden.

Både støj og fejl vil blive simuleret for at undersøge om metoderne kan håndtere støjen og fejlene, og en undersøgelse af grænse- eller ekstreme - værdier gennemføres for at fastslå hvad der er en god rekonstruktion.

Vi undersøger også forskellige stopkriterier og hvorledes de påvirker rekonstruktionen der er fundet ved de iterative metoder. Vi bruger vores kendskab til kornenes egenskaber til at formulere et nyt stopkriterie og afprøver dets anvendelighed.

Preface

This master thesis is prepared at the Department of Informatics and Mathematical Modeling, Technical University of Denmark (DTU). It marks the finishing of a master degree in *Mathematical Modeling and Computations.* It represents a workload of 30 ETCS points and has been prepared for a sixth month period from February 1st to August 1st 2011. The study has been conducted under the supervision of Professor Per Christian Hansen.

I would like to thank my supervisor Per Christian Hansen for being supportive from the start of the project until the end. From him new ideas and directions from where to go on with this project arose. Also i would like to thank my family and friends for being supportive throughout the whole project.

List of Symbols

Symbol	Quantity	Dimension
$\|\cdot\|_1$	1-norm	
$\|\cdot\|_2$	2-norm	
$\|\cdot\|_F$	Frobenius-norm	
$\overrightarrow{(\cdot)}$	geometric vector	$(2,1)^T_{2D}, \quad (3,1)^T_{3D}$
$(\cdot)^{opt}$	optimal solution	
$\langle\cdot,\cdot\rangle$	inner product	
A	coefficient matrix	$m \times n$
$E(\cdot)$	expected value	
$\mathcal{H}_i$	i'th hyperplane	
I	identity matrix	
$\mathcal{K}_k$	Krylov subspace of dimension k	
M, T	Symmetric positive definite matrices	$m \times m$, $n \times n$
$NNZ(\cdot)$	number of non-zero elements	scalar
$\mathcal{P}_i(\cdot)$	i'th projection	
U_i	left singular matrix	$m \times n$
V_i	right singular matrix	$n \times n$

b	right hand side	m
b_{exact}	exact right hand side	m
$cond(\cdot)$	condition number of a matrix	
$delta_{per}$	upper bound on solution norm	scalar
ϵ	perturbation on the right hand side	m
η	relative noise level	scalar
k	iteration number and truncation parameter	scalar
λ	regularization parameter and relaxation parameter	
m, n	matrix dimensions	scalars
Ω	our domain	
ϕ_i^λ	Tikhonov filter factor	scalar
r^k	k'th residual vector	m
σ_i	singular value of a matrix	scalar
Σ_i	diagonal matrix consisting of the singular values	$n \times n$
τ	threshold level	scalar
u_i	left singular vector	m
v_i	right singular vector	n
x_{exact}	the exact solution	n
x_{naive}	the naive solution	n
x^{rec}	reconstruction of x	n
$x^{rec,bin}$	binary reconstruction of x	n
x_k	TSVD solution	n
x^k	solution of the k'th iteration	n

List of Abbreviations

Text	Meaning
"w" in tables and figures	with
"w0" in tables and figures	without
"nn" in tables and figures	the non-negativity constraint

Contents

Chapter 1

Introduction

Tomography is a very important instrument for non-destructive investigations of elements and bodies. The most well-known application is in medicine where it is extensively used to interpret pictures obtained by scanning for tumors in the fight of cancer. Scanning makes use of tomography in order to obtain 3D pictures. In material sciences it can be used in a similar way to find inclusions or vacancies. These pictures are for several reasons often blurred e.g. due to weak contrast or noisy reflections in the material. It is therefore desired to improve the quality of the image. Numerical iterative methods can be applied to the images in order to improve their quality.

These iterative methods are formulated in computer programs and it is desirable to optimize these with respect to computation time and quality of the reconstructions. It is possible to prove mathematically convergence of the methods, but convergence does not imply optimization. Therefore the methods are tested on various cases and parameter changes are performed in order to find optimal combinations of the parameters. These optima may be local. The dream of finding a global optimum remains a dream.

What is an optimum? When we look at an image or a picture, what distinguishes a good picture? It is a subjective criterium. Humans are individuals - we do not necessarily prefer the same adjustment. Some like red whine, some like white, and some does not like whine at all. Therefore we need objective criteria, that can be expressed in mathematical terms.

Many tests have been performed and certain criteria have been formulated on the basis on the results of the tests. But the criteria may be limited in the scope, and may not be applicable for complex problems.

The real world is very complicated so it may happen that the theory which has been developed under certain assumptions does not apply, because one or more of the assumptions can not be strictly satisfied.

The objective of this thesis is to perform further tests with application of known and new criteria that are developed and compare and assess the results.

1.1 Structure of the Thesis

The goal of this thesis is to compare iterative methods against each other for carefully specific-chosen tests. In order to do these comparisons, new measurement methods have been discussed, analyzed and implemented all in order to seek out the advantages and disadvantages of the iterative methods. Both old and new stopping criteria will be considered to simulate real-world problems, giving a more realistic view upon the iterative methods. Also discussion of how our test problems should be implemented will be discussed and analyzed, with different ways of simulating error and noise within our data.

Typically theory and tests follow the same pattern in 2D as it does for 3D. Most of the theory is easiest to explain in 2D, but the same theory is used for 3D. Therefore most of the theory throughout this report is formulated and shown in 2D, giving the best visualization and confirmed in 3D.

When different tests are made for our iterative methods, we have decided not to illustrate all. Therefore in several cases, features are shown in 2D only (or 3D), and often only 1 iterative method per group of methods is shown. The reader is informed that all the different test have been made for all the methods, but only one is shown since the others yield the same results.

The structure of the thesis is based on the following

- **Chapter 2:** We start off by introducing the term "Inverse Problem" and give the needed information regarding this term. Also the basic concepts like SVD will be introduced before proceeding.

- **Chapter 3:** In this chapter we introduce all the iterative methods. This regards the SIRT, the ART and the CGLS method. The concept of semi-convergence will be shown along with an explanation of the term ”measuring time”.
- **Chapter 4:** In this chapter we introduce and show what Tomography is. Mathematically, it will be shown how a tomography test problem can be created and formulated to a set of equations. Also the geometrical aspect of our grains will be analyzed.
- **Chapter 5:** In this chapter we introduce the first concepts, which will be used to measure the quality of our reconstructions. Different constraints, which can be applied to our iterative methods will be discussed and analyzed.
- **Chapter 6:** In this chapter we will consider the A matrix and the creation of this. Different tests will be made in order to observe how we can choose A and still obtain good quality measurements. Also tests for ”extreme values” will be performed.
- **Chapter 7:** The concept ”Geometrical error” will be explained and shown in this chapter. Different tests for this phenomenon will be simulated and performed, in order to investigate how the iterative methods handles this.
- **Chapter 8:** In this chapter we introduce the concept stopping criterion. Both old as well as newly found stopping criteria will be considered and the flaws of these. Discussion on how the newly formed stopping criterion should be implemented is included, with a discussion of ideas to new stopping criteria.
- **Chapter 9:** This chapter contains the conclusion and the summary of the thesis. Also suggestions for future work have been made.

Chapter 2

Inverse Problems

In this chapter it will be explained what a "inverse problem" is. From this the issues about these certain problems will be defined along with possible solutions to these inverse problems. First a overall definition will be given of these problems, followed by how it can be applied to the cases that the report consists of. This chapter is based on [2].

2.1 Explanation of an Inverse Problem

When speaking about an "inverse problem" these are often seen in different branches of science and mathematics. The problems appear when one wishes to compute certain unavailable data from accessible measurements [12]. A typical way to formulate a simple mathematical inverse problem is on the form

$$Ax = b. \tag{2.1}$$

Here A is a matrix multiplied on a vector x resulting in a vector b. In a typical reconstruction example x corresponds to the "true" image while b is the image we are dealt with. In our case we have these data on vector form, and our solution vector x is reshaped into a $N \times N$ image. Generally b will be noisy, and therefore it is difficult to obtain a proper reconstruction by solving (2.1).

One might ask the question, "Why is this so hard to reconstruct?" In a mathematical way the solution is obtained by saying

$$x_{exact} = A^{-1}b.$$

To understand this we first need to know some definitions for matrices, and as we will see later, the noise in the vector b will completely ruin the reconstructions. The solution obtained from (2.2) will be referred to as the "naive solution" throughout the report, indicating

$$x_{naive} = A^{-1}b. \tag{2.2}$$

.

2.2 Well-Posed and Ill-Posed Problems

When a mathematical problem involves matrix and vector calculations, the problem can be well-posed or ill-posed. A numerical solution is difficult to compute with precision when a problem is ill-posed, and therefore we need to do something about this. That is why we will look upon a way to reformulate an ill-posed problem, because if this is doable, we have the possibility of obtaining a decent solution to our problem.

The feature that usually defines an inverse problem is that the problem is "ill posed", and if it does not fulfill certain criteria, the problem is definitely ill-posed. Jacques Hadamard has made following definition for a well-posed problem:

A problem is said to be well-posed if it has the following 3 properties:

- **Existence:** A solution exists
- **Uniqueness:** The solution is unique, it's the only one to the problem
- **Stability:** The solution depends continuously on the data.

If a problem does not fulfill all of these properties, the problem is said to be ill-posed, which usually is the case for an inverse problem. Often we are faced with the fact that in order to satisfy those criteria, one has to reformulate the given problem.

So how can we reformulate a given problem in such a way that it fulfills all the above criteria? The first criterion seems obvious to satisfy, since it would be perfect if a solution could exist to any given problem. The fact is that for many problems there does not exist a solution. If we consider example (2.3), it is clearly seen that there does not exist a solution to this given problem.

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} x = \begin{pmatrix} 1 \\ 2.2 \end{pmatrix} \quad \Rightarrow x = 1 \quad \text{and} \quad 2x = 2.2 \tag{2.3}$$

So for this simple example we have to reformulate the problem in such a way that the existence criteria can be obtained. For this example a solution would be to consider the solution to the least squares problem given by

$$\min_x \left\| \begin{pmatrix} 1 \\ 2 \end{pmatrix} x - \begin{pmatrix} 1 \\ 2.2 \end{pmatrix} \right\|_2^2 . \tag{2.4}$$

Here the exact solution would be $x = 1.08$, and thereby the existence criteria would be satisfied. The same holds for the uniqueness criteria, namely that it's the only solution to the problem. We consider example (2.5).

$$x_1 + x_2 = 1. \tag{2.5}$$

As one can see there are infinitely many solutions to this underdetermined problem, but again we can obtain the second criterion by a reformulation. The solution here would be that the 2-norm of x is minimal, meaning that we wish to solve the given problem

$$\min_x \|x\|_2 \quad \text{s.t.} \quad x_1 + x_2 = 1. \tag{2.6}$$

Hereby the obtained solution will be $x_1 = x_2 = \frac{1}{2}$, and this is the only solution to the minimization problem. So the last criterion we need to check is "Stability". Stability means that a small change in the observations or data one has measured, will resemble a small change in the solution. So if this is not obtained, a small change in observations could mean we end up with a very bad solution to the problem. Therefore we need to reformulate the problem, which is best illustrated with example (2.7).

$$\min_x \|Ax - b\|_2, \quad A = \begin{pmatrix} 0.16 & 0.10 \\ 0.17 & 0.11 \\ 2.02 & 1.29 \end{pmatrix}, \quad x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad b = Ax + \epsilon \tag{2.7}$$

Here ϵ will simulate a small perturbation to b. We now let $\epsilon = [0.01\ -0.03\ 0.02]^T$ denote the small perturbation added to b and we end up with the least squares result

$$x = \begin{pmatrix} 7.01 \\ -8.40 \end{pmatrix} \quad \Rightarrow \quad \|Ax - b\|_2 = 0.022. \tag{2.8}$$

So as we can see from this, even though the residual is very low, and the perturbation was small, we end up with a completely different solution, namely

$$\begin{pmatrix} 7.01 \\ -8.40 \end{pmatrix} \text{compared to} \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

This is exactly due to the reason that the matrix is ill-conditioned, and therefore we have to make a reformulation of the problem.

A solution to this is to make use of equation (2.9)

$$\min_x \|Ax - b\|_2 \quad \text{s.t.} \quad \|x\|_2 \leq \delta_{per}. \tag{2.9}$$

By choosing an appropriate value of δ_{per}, one would be able to find an approximately acceptable solution which can be seen below.

$$x_{0.1} = \begin{pmatrix} 0.08 \\ 0.05 \end{pmatrix}, \quad x_1 = \begin{pmatrix} 0.84 \\ 0.54 \end{pmatrix}, \quad x_{1.37} = \begin{pmatrix} 1.16 \\ 0.74 \end{pmatrix}, \quad x_{10} = \begin{pmatrix} 6.51 \\ -7.60 \end{pmatrix}.$$

As we can see the solution $x_{1.37}$ seems reasonable, but the main problem about this method is to choose an appropriate $\delta_{per} - value$. Usually δ_{per} is unknown for these kinds of problems.

One of the main problems we have regarding inverse problems is that we know that our data contains noise. Before going any further let us first have a look at what this noise consists of and how it does appear.

2.3 Relative Noise Level

Whenever one is faced with a problem, one has to make calculations for noise within the data. If everything was perfect we would have the following formula

$$b^{exact} = Ax^{exact},$$

and everything would seem nice. As we saw earlier a small perturbation could force our solution to be far from the exact solution, and exactly this small perturbation can correspond to the noise within the data. The fact is that we do not obtain b^{exact} but instead we have b where

$$b = b^{exact} + e, \quad E(e) = 0. \tag{2.10}$$

For this reason we need to be able to simulate noisy data, and therefore we introduce the term "relative noise level". Relative noise level is a term indicating how much noise we have added to the data. By doing this we have a way of simulating noise within data, and then compare our numerical methods for different noise levels. Also we can see if the noise plays a role on certain variables, which we will look upon later. Throughout the report the relative noise level is defined as

$$\eta = \frac{\|e\|_2}{\|b^{exact}\|_2}, \tag{2.11}$$

where e and b^{exact} are defined from (2.10). Throughout the report, if something is written like "we have added 40% noise to the data", it means that $\eta = 0.4$ in this example, and whenever we refer to the relative noise level within the data, η is referred to. In MATLAB we have the following code which can be used to add noise

$$\texttt{e} = \texttt{randn(size(b\_exact));} \tag{2.12}$$
$$\texttt{e} = \texttt{e/norm(e);} \tag{2.13}$$
$$\texttt{b} = \texttt{b\_exact} + \texttt{noise} * \texttt{norm(b\_exact)} * \texttt{e;} \tag{2.14}$$

By now we have looked at different kinds of inverse problems, and which criteria they should maintain. Also the noise term has been defined. In the next section we will look into how to make use of all this in a numerical sense, namely how this should be applied to a computer.

2.4 SVD (Singular Value Decomposition)

Whenever we are faced with a difficult problem which needs a numerical solution, it is not enough to have it defined as a continuous problem. We first have to discretize the problem, hereby making the problem suitable for digital computation, and also make the problem finite, meaning that we have a limited number of elements.

When one is working with matrices on digital computers, one is automatically also working in finite dimensions. Luckily there is a powerful tool for matrix analysis known as the SVD (Singular Value Decomposition). Throughout this report, when speaking about a SVD analysis, we will consider matrices of any size, meaning that we will look upon matrices of size $A \in \mathbb{R}^{m\times n}$ where we will look at all three cases that can occur, namely $m > n, m < n$ or $m = n$. So for these types of matrices the SVD will take the following form

$$A = U\Sigma V^T = \sum_{i=1}^{\min m,n} u_i\sigma_i v_i. \tag{2.15}$$

The matrices U and V consists of the singular vectors, meaning that

$$U = (u_1, \ldots, u_m), \quad V = (v_1, \ldots, v_n),$$

and furthermore, $U \in \mathbb{R}^{m \times n}$, $V \in R^{n \times n}$ and since these matrices consists of orthonormal columns we can write $U^T U = V^T V = I$, where I is the identity matrix. The matrix Σ will be a diagonal matrix consisting of all the singular values in decreasing order, meaning that

$$\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n), \quad \sigma_1 \geq \ldots \geq \sigma_n \geq 0.$$

As we shall see, a high condition number, defined from [9] can result in poor reconstructions due to noise. If we want to compute the condition number of a matrix we use norms, and therefore the following two norms are defined:

$$\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2 = \sigma_1 \tag{2.16}$$

$$\|A\|_F = (\sigma_1^2 + \sigma_2^2 + \ldots + \sigma_n^2)^{\frac{1}{2}} \tag{2.17}$$

By these norms we are able to define the condition number of a matrix as

$$cond(A) = \frac{\sigma_{max}(A)}{\sigma_{min}(A)}. \tag{2.18}$$

Here $\sigma_{max}(A)$ and $\sigma_{min}(A)$ indicates the largest and smallest eigenvalues of the matrix A. And furthermore it is known that if the matrix has a large condition number it is said to be ill-posed, whereas a small condition number refers to a well-posed matrix. If we consider equation (2.1), the condition number can be roughly thought of as how x will change with a change in the data b. In other words, if the condition number is small, then a small change in b will only produce a small change in the solution x (Stability). On the other hand if the condition number is large then a small change in b can produce a large change in the solution x. From example (2.7) we can compute the condition number which is $cond(A) \simeq 1.1e3$. This is a large condition number and is the reason that example (2.7) gave different solutions for small perturbations. The proofs of the norms are omitted in this report.

2.4.1 The Discrete Picard Condition

Before going into depth with how the SVD can be applied, there is one important thing to investigate first. As we saw from definition 2.2, there are certain criteria that should be satisfied for a problem to be well-posed. Considering the SVD, the criteria "Existence" will hold if the solution obeys the Discrete Picard Condition, which is defined in [2].

The Discrete Picard Condition. Let τ denote the level at which the computed singular values σ_i level off due to rounding errors. The Discrete Picard condition is satisfied if, for all singular values larger than τ, the corresponding coefficients $|u_i^T b|$, on average, decay faster than the σ_i

So if this holds, there will exist a solution. This is best illustrated with some figures and is seen in figures 2.1 and 2.2. If we observe the first figure, 2.1

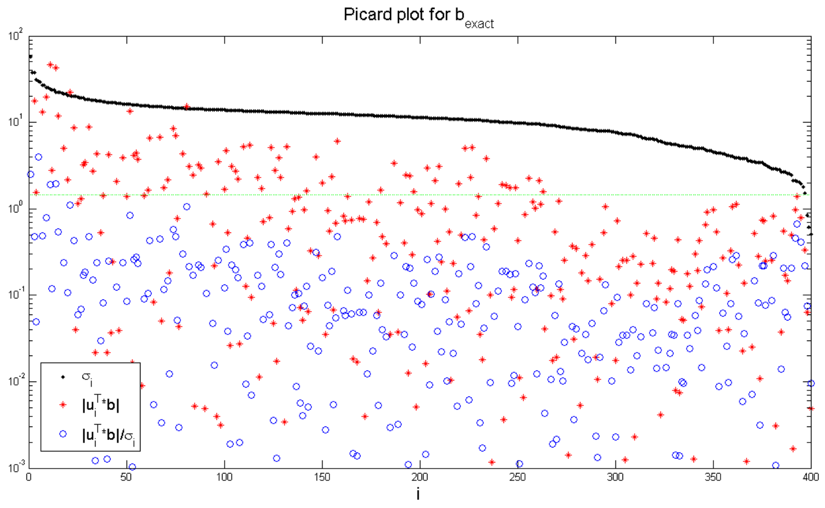


Figure 2.1: Plot of the Picard condition for a tomography test problem which we will discuss in a later chapter, namely Chapter 5. The Picard plot is created with 0 noise added, and the Picard condition is satisfied for this figure.

we notice that the Picard condition is satisfied since the SVD coefficients $|u_i^T b|$ on average decay faster than the singular values σ_i. If we observe figure 2.2 something else appears to the eye. The same SVD coefficients, $|u_i^T b|$ are still decaying faster than the singular values, but only until around $i = 125$ meaning that until this point the Picard condition is satisfied. At this level the coefficients starts to level off, which is due to the fact that we have added some noise to the image. Also it is noticed that the SVD coefficients $\frac{|u_i^T b|}{\sigma_i}$ are increasing as i is increased, when noise is added to the test problem. These SVD coefficients are, at this level, starting to become dominated by the noise. So to sum up the main part from these plots, we see that the large SVD coefficients from $|u_i^T b|$ are almost equal for both plots, whereas the small coefficients from $|u_i^T b|$ are dissimilar due to the perturbation.

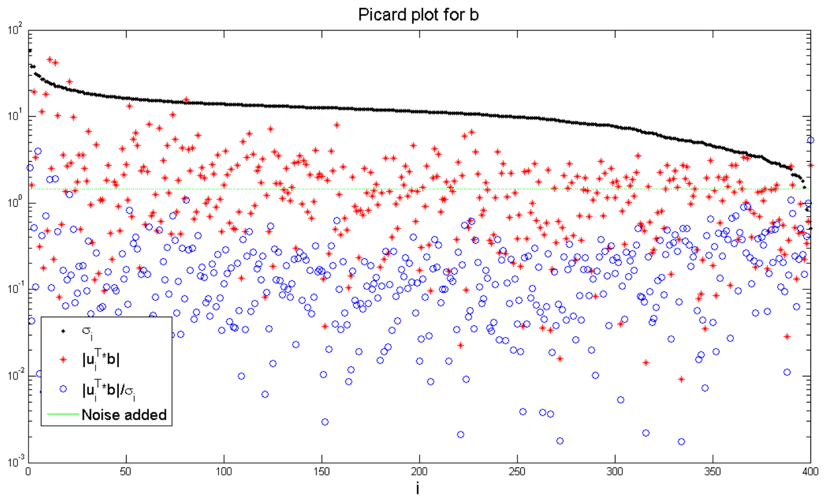


Figure 2.2: Plot of the Picard condition for a tomography test problem which we will discuss in a later chapter, namely Chapter 5. The Picard plot is created with noise level = 0.5. The green line indicates the error that has been added to the plot, and the noticeable thing here is that the coefficients $|u_i^T b|$ are "moved up" to this line, which is due to the error we have added to our test problem. In figure 2.1 the values were much lower.

Earlier we saw that the naive solution can be a poor choice regarding reconstructions, since small perturbations resulted in solutions to far from the exact solution. From the SVD we saw how the error contribution could become to large and start dominate our signal, which resulted in the poor solutions. The fact is that there exist a way of deriving equation (2.2) by the SVD, which will become handy when trying to make a decent reconstruction.

2.4.2 Usage of the SVD

The most important relations between the singular values and vectors are given below

$$\begin{aligned} Av_i &= \sigma_i u_i, \quad \|Av_i\|_2 = \sigma_i, \quad i = 1,\dots,n \\ A^{-1}u_i &= \sigma_i^{-1} v_i, \quad \|A^{-1}u_i\|_2 = \sigma_i^{-1}, \quad i = 1,\dots,n. \end{aligned}$$

The last equation only holds if A is square and nonsingular. Since V is a orthogonal matrix, we can write x in such a way that

$$x = VV^T x = \sum_{1}^{n} (v_i^T x) v_i$$

and it also holds for b implying that

$$b = \sum_{i}^{n} (u_i^T b) u_i.$$

By substituting this into the equation for the SVD we obtain

$$Ax = \sum_{i}^{n} \sigma_i (v_i^T x) u_i$$

and from this the naive solution is given by

$$x = A^{-1} b = \sum_{i}^{n} \frac{u_i^T b}{\sigma_i} v_i.$$

Now we have derived an expression for the naive solution consisting of a sum of "something". This is of importance due to the fact that it is possible to "cut off" the terms which are dominated by Gaussian noise, and this might improve the quality of our reconstruction. In the next sections, examples of methods of "cutting off" will be shown.

2.5 Regularization

Whenever we are trying to solve equation (2.1) we are actually trying to solve the least squares problem given by

$$\min_x \|b - Ax\|_2^2.$$

And as we saw earlier the naive solution, see (2.2), is not a proper solution to our problem. The main reason is that the reconstruction we obtain lies to far away from the exact solution that we seek, due to a small perturbation on the right hand side. In other words, we do not have b^{exact}, since

$$b \text{ (which is the vector we obtain) } = b^{exact} + e.$$

Therefore we need a technique that in some way is able to remove this noise, and for that we apply regularization. Regularization is a way of adding or removing information given our specific problem. In our case it would be appropriate to remove some of the noise. In the following we will be looking at two regularization methods that are able to do this.

2.5.1 TSVD

One of the methods is the so-called TSVD, which stands for Truncated SVD. It is very easy to implement, and the idea is simple. First let us have a look at the formula for the TSVD.

$$x_k = \sum_{i=1}^{k} \varphi_i \frac{u_i^T b}{\sigma_i} v_i,$$

where φ_i are the filter factors for this method. And with this let us introduce the so-called filter factors.

The thing is, as we saw on the Picard plot, that when noise is added to the test problem, our SVD coefficients will start to increase when the singular values becomes small. These small singular values arises from the perturbation and will start to dominate our reconstructions. Therefore, if possible, we would like to cut these components off, so in theory we would only have singular values from the "true data" left. From the two Picard plots it is noticed that the large singular values are more or less equal to each other, and therefore we wish to keep these. The idea here is then to choose an appropriate value of k, to ensure that we cut off all the SVD components which are dominated by the perturbation, and we keep all the SVD coefficients which should represent the true data.

This is exactly what we can do with the filter factors, since for the TSVD solution they are defined by

$$\varphi_i = \begin{cases} 1 & , i \leq k, \\ 0 & , i > k. \end{cases}$$

By adding this term we can cut off all the small SVD coefficients while keeping the large ones.

This method is very intuitive, and the solutions are easy to find, once the SVD components, or at least the first k singular values and vectors have been computed. This turns out to be a problem though, since the SVD is a very long-time-taking calculation and it is not at all suitable for large-scale problems. Even for small-scale problems the SVD coefficients can take a lot of time to calculate. Therefore we need different regularization methods which are more suitable for these types of problems. And from this we introduce the Tikhonov regularization.

2.5.2 Tikhonov Regularization

One of the most successful regularization methods is Tikhonov's method. The formula is given below

$$\min_x \|Ax - b\|_2^2 + \lambda^2 \|x\|_2^2, \tag{2.19}$$

where the Tikhonov solution x_λ, is the solution to equation (2.19), and λ is a positive regularization parameter which controls the weighting between the two terms in 2.19.

The first term $\|Ax - b\|_2^2$, as seen before, measures the "goodness-of-fit". The problem here is that if this term becomes to large, our solution x is far away from the exact solution. But if we fit this term so well that the residual will become closer to 0, we would actually have fitted the noise as well. Therefore we should fit the term in such a way that the residuals will correspond to the average size of the errors found in b. Then, intuitively, we would not fit the noise in this term.

The second term $\|x\|_2^2$ measures the solution's regularity. Since the naive solution for our type of problems is dominated by high frequency components, then if we are able to control the norm of this term, we might be able to get rid of these high frequency components.

The last term works in a similar way as a weight, the larger λ, the more the minimization of the solution norm $\|x\|_2$ will influence the regularization. Also, the less λ, the less $\|x\|_2$ influences the minimization of the solution norm, indicating that we are weighting in favor of the noisy data. If we set $\lambda = 0$ we obtain the original problem with the naive solution.

We saw in TSVD that given a number k, we decided where to "cut-off" our SVD coefficients and hereby obtain a reconstruction. Tikhonov works in a similar way, but with the difference that it has a more sophisticated way of deciding if a term should be removed or not. This is done by a different and more appropriate choice of the filter-factors.

$$\varphi_i^\lambda = \frac{\sigma_i^2}{\sigma_i^2 + \lambda_2} \approx \begin{cases} 1 & , \sigma_i \gg \lambda, \\ \sigma_i^2/\lambda^2 & , \sigma_i \ll \lambda. \end{cases} \tag{2.20}$$

The idea with these filter-factors is that when the singular values σ_i are larger than λ, the filter-factor will equal 1, meaning that the given SVD coefficient will contribute fully. Whereas if the opposite occurs, meaning that σ_i is much smaller than λ, the contribution from the SVD coefficient will be close to 0. So instead of just cutting off the SVD coefficients, we get a large contribution from the large singular values, whereas we get a small contribution from the small singular values. So both Tikhonov and TSVD resembles each other by filtering the solution. From Tikhonov, one can filter with appropriate values of λ, and one does the same by choosing k in TSVD.

So far the methods that have been discussed regards the calculation of the SVD coefficients or computing the Tikhonov solution via least squares formulation. The problem is that especially the SVD coefficients takes very long time to compute, even for a small matrix. Since these methods uses factorization, they are not suitable for a large-scale problem, since the time it would take to compute these coefficients is too long. Also there is the problem of storing the matrix A. As we shall see later, when we are facing the large-scale problems, we shall see that A is a sparse matrix, meaning that a lot of the values inside A are equal to 0, which we want to exploit. And due to this we do not want to make use of factorizations of the A matrix. When we use regularization methods we must ensure two things:

- To avoid a matrix factorization we must ensure that we make use of matrix-vector multiplication for our large-scale method.
- Since our methods has a regularization parameter we can adjust, a large-scale method must ensure that the user can choose this parameter in such a way that we do not have to solve the problem from the beginning when a new value for the parameter is chosen.

This is why we will look upon iterative methods, since instead of using a regularization parameter, the iterative methods rely on the current iteration. Also since the iterative methods use matrix-vector multiplications, the methods satisfy the second requirement from above.

2.6 Summary

Until now we have barely scratched the surface regarding this thesis. The term "inverse problem" has been defined and we have seen which problematic can arise when solving this type of problems. In the real world, noise exist and therefore we showed how we can simulate this within our problems, to make our test cases resemble the real world in a better way. An introduction followed to how we can discretize our problem, and with this we introduced a method (SVD) for matrix analysis. From this the beginning of iterative methods (Tikhonov) was shown, and the theory was explained. With this we should have our basic tools ready for further investigation of the iterative methods.

CHAPTER 3

Iterative Methods

In this chapter we will go into depths with how the iterative methods works. Some methods will resemble each other, and some will have their own advantages and disadvantages. Also there are certain criteria a method must apply before it can be used as an appropriate solver to our given problem, which we will look into. Throughout the report we will look upon seven different methods, where these methods can be characterized into three groups. But more on this will follow. This chapter is based on [6] and [2].

3.1 Usage of Iterative Methods

As a start one should know how an iterative method usually works, and here the special feature is that an iterative method uses iterations to converge towards a solution. In other words, we always start with a user-specified starting vector x^0, which usually is a vector consisting only of zeroes. The optimal choice of starting vector will be adressed in the test section of this report. From this vector the method finds x^1, x^2, $\ldots$. It does not mean that the sequence converges to the exact solution or a ”nearby” solution. Therefore we must ensure that an iterative method will fulfill the ”Semi-Convergence” criteria, or else we can end up with divergence towards the naive solution which we saw was not an appropriate solution in Chapter 2.

3.1.1 Semi-Convergence

As mentioned above, the semi-convergence criterion is of real importance when considering iterative methods. We will show that if a method fulfills this criteria, we know that at some iteration number, x^i for $i = 1, \dots, k$, we will be very close to the exact solution. But the longer we iterate from this point on, the closer the scheme gets to the naive solution. Let us illustrate this with an example, we consider figure 3.1 and figure 3.2. On the first figure the curve illustrates the

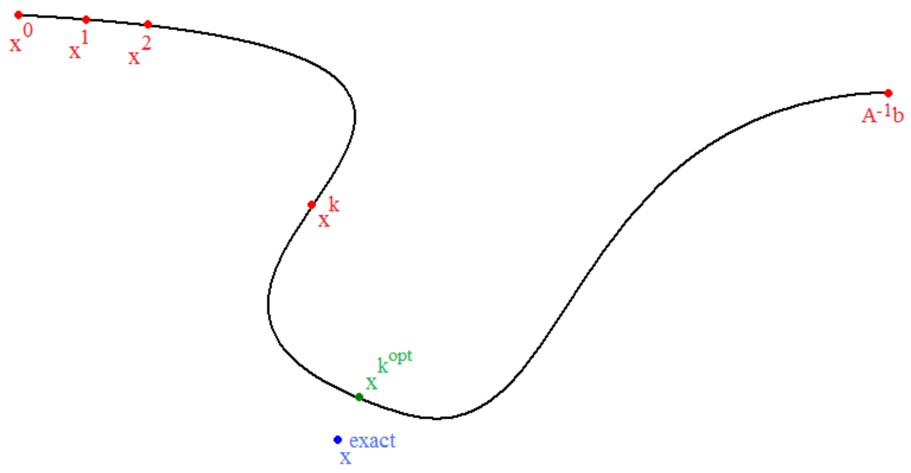


Figure 3.1: Plot of the concept of semi-convergence. We start at some point x^0, and then iterate towards the optimal solution, $x^1, x^2, \dots, x^{k^{opt}}$. From this point on, $x^{k^{opt}}$, the solution will diverge towards the naive solution $A^{-1}b$.

concept of semi-convergence. We start at a user-specified vector and start the iteration. At some point we reach the optimal iteration $x^{k^{opt}}$, and if we keep on iterating we will diverge towards the naive solution. On figure 3.2 four examples with different iteration numbers are shown. As we can see, the top left picture is still a bit blurry since an insufficient number of iterations has been used. The bottom two pictures are still decent, but more and more noise in the background appears, indicating that we are diverging from the exact solution. The top right picture shows the optimal reconstruction found. On a semi-convergence plot, the number of iterations will be along the x-axis whereas the relative error will be along the y-axis. When the criterion semi-convergence is obtained, we mean that a minimum of the relative error has been found which corresponds to the iteration $x^{k^{opt}}$. Therefore we introduce the term "relative error". This is defined

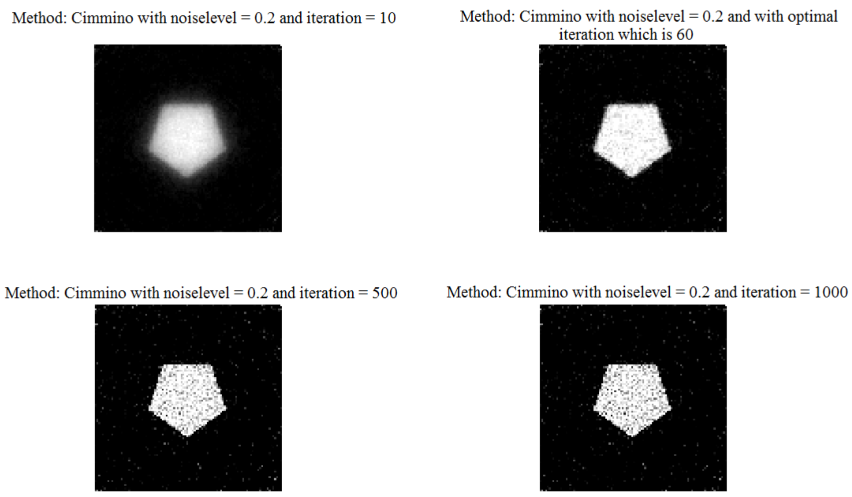

Method: Cimmino with noiselevel = 0.2 and iteration = 10
Method: Cimmino with noiselevel = 0.2 and with optimal iteration which is 60
Method: Cimmino with noiselevel = 0.2 and iteration = 500
Method: Cimmino with noiselevel = 0.2 and iteration = 1000

Figure 3.2: Plot of some reconstructions, which illustrates the concept of semi-convergence. The top left picture is still blurry since only 10 iterations has been used which is not enough. The down right and left figures show how more noise appear in the background as we tend to diverge from the optimal solution. The top right figure shows the optimal solution found for this problem.

as

$$
\begin{aligned}
rel_{error}(i) &= \frac{\|x^i - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100, \quad i = 1, 2, \ldots, k. \\
rel_{error}(i) &= \frac{\|x^i - x^{exact}\|_2}{\|x^{exact}\|_2} \cdot 100, \quad i = 1, 2, \ldots, k.
\end{aligned}
$$

Later in the report we will consider which norm to make use of.

In the next sections we will go into depth with every iterative method that we will look upon in this report.

3.2 Regularization Methods

In Chapter 2 the need for regularization was discussed and some methods were looked upon. A method we will use in this project is the so-called CGLS method (Conjugate Gradient Least Squares). The method is defined in the following way:

$$x^k = \underset{x}{\operatorname{argmin}} \|Ax - b\|_2 \quad \text{s.t.} \quad x \in \mathcal{K}_k, \tag{3.1}$$

where $\mathcal{K}_k$ is the Krylov subspace. For more information we refer the reader to [2]. The CG algorithm solves a system of normal equations given by $A^T Ax = A^T b$ with the restriction from above. We have to ensure though, that we have a symmetric positive definite coefficient matrix, else the solution may diverge. As an extra feature to this CG algorithm, the option "non-negativity" has been implemented. But more on this criterion later. The last feature we need to observe for this method is if it fulfills the semi-convergence criteria. To do this we look upon figure 3.3. From this figure it follows that the method fulfills

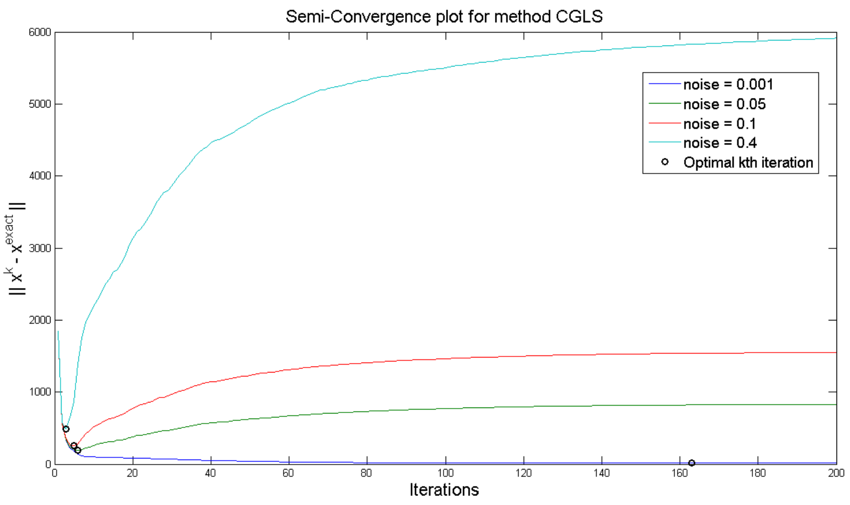


Figure 3.3: Plot of the CGLS method to illustrate semi-convergence. As we can see the method converges to wards a minimum which is our solution. Afterwards the solution diverges.

semi-convergence. The reason for this is that given a specific noise level, the curves on the graph converges towards a specific solution. Afterwards the curve start diverging towards the naive solution. This minimum obtained corresponds to $x^{k^{opt}}$, and the point with semi-convergence was to find an optimal solution before diverging towards the naive solution.

3.3 SIRT Methods

The next type of methods are the so-called SIRT (Simultaneous Iterative Reconstruction Techniques) methods. The SIRT methods use all the equations at one time in an iteration, and the name ”simultaneous” comes from this fact. A property of the SIRT methods is that they can be written in a general formula, which is defined as

$$x^{k+1} = x^k + \lambda_k T A^T M(b - Ax^k), \quad k = 0, 1, 2, \ldots . \tag{3.2}$$

Here x^k is the current iteration vector, x^{k+1} is the new iteration vector, λ_k is a relaxation parameter, and the matrices M and T are symmetric positive definite. The SIRT methods differ between each other by the matrices M and T. By choosing different values of these matrices, one can go from one method to another. For methods defined as (3.2), with $T = I$ where I is the identity matrix, we have the following convergence theorem.

Theorem 1. *The iterates on the form (3.2) with $T = I$ converge to a solution x^* of $\min_x \|Ax - b\|_M$ if and only if*

$$0 < \epsilon \leq \lambda_k \leq 2/\frac{2}{\sigma_1^2} - \epsilon,$$

where ϵ is an arbitrarily small but fixed constant and σ_1 is the largest singular value of $M^{\frac{1}{2}}A$. If in addition $x^0 \in \mathcal{R}(A^T)$ then x^* is the unique solution of minimum T^{-1}-norm (minimum 2-norm if $T = I$).

Now that we can ensure converge by the above theorem, let us have a look at the different methods. It should be mentioned that from [6], weights have been implemented for some of the methods. Therefore in this report, the weights have been included in the formulas, but use and tests of these weights will not be performed. The methods that have weights incorporated within them are Cimmino's method, CAV and DROP, and every time tests are being performed with these methods, a default value of $weights = 1$ will be used.

3.3.1 Landweber's Method

The first type of SIRT method is Landweber's method. The method is listed below

$$x^{k+1} = x^k + \lambda_k A^T(b - Ax^k), \quad k = 0, 1, 2, \ldots \tag{3.3}$$

where $M = I$ and $T = I$ from (3.2). From this formula we can express the iterates as filtered SVD coefficients and therefore obtain filter factors. This is done by writing the iterate as

$$x^k = V\Phi^k\Sigma^{-1}U^Tb,$$

where the elements in the matrix $\Phi^k = \mathrm{diag}(\varphi_1^k, \dots, \varphi_n^k)$ correspond to the filter factors given as

$$\varphi_i^k = 1 - (1 - \lambda\sigma_i^2)^k.$$

We notice that for small values of σ_i, we have that $\varphi_i^k \approx k\lambda\sigma_i^2$ which has the same decay rate as (2.20). As an example of semi-convergence for a SIRT method, we observe figure 3.4. This example illustrates that our method satisfies the criterion semi-convergence, and this is seen due to the fact that the method first converges towards a solution $x^{k^{opt}}$ and then starts diverging towards the naive solution. Recall that this was the concept of semi-convergence. The rest of the SIRT methods do also satisfy this criterion.

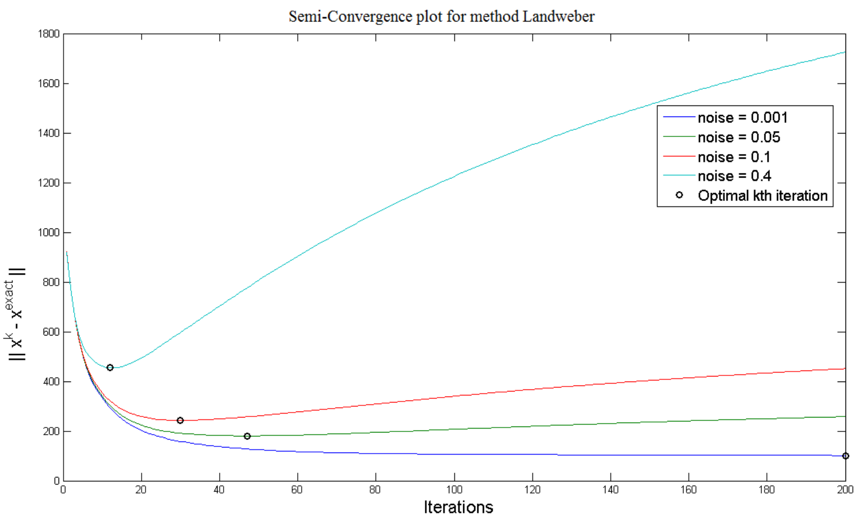


Figure 3.4: Plot of Landweber's method to illustrate semi-convergence. As we can see the method converges to wards a minimum which is our solution. Afterwards the solution diverges.

3.3.2 Cimmino's Method

Cimmino's method is based on reflections on hyperplanes, and therefore we first need to introduce the definition of a reflection on a hyperplane

$$R_i(z) = z + 2\frac{b_i - \langle a^i, z\rangle}{\|a^i\|_2^2}a^i.$$

The affine hyperplane is defined as $\mathcal{H}_i = x \in \mathbb{R}^n | \langle a^i, x\rangle = b_i$ for each of the linear equations from $Ax \simeq b$, where $A \in R^{m\times n}$. Hereby the orthogonal projection on the hyperplane is given by $\mathcal{P}_i(z) = z + (b_i - \langle a^i, z\rangle)\|a^i\|_2^{-2}a^i$ where z is a vector projected on the hyperplane. By taking an average of the projections from the previous iterate, x^k, from all the hyperplanes we obtain the next iteration x^{k+1} for Cimmino's method, which is given below

$$\begin{aligned} x^{k+1} &= \frac{1}{m}\sum_{i=1}^{m}\mathcal{P}_i(x^k) = \frac{1}{m}\sum_{i=1}^{m}\left(x^k + \frac{b_i - \langle a^i, x^k\rangle}{\|a^i\|_2^2}a^i\right) \\ &= x^k + \frac{1}{m}\sum_{i=1}^{m}\frac{b_i - \langle a^i, x^k\rangle}{\|a^i\|_2^2}a^i. \end{aligned}$$

As mentioned for Cimmino's method, the possibility of positive weights has been implemented. When a relaxation parameter λ_k is added we obtain the formula

$$x^{k+1} = x^k + \lambda_k\frac{1}{m}\sum_{i=1}^{m}w_i\frac{b_i - \langle a^i, x^k\rangle}{\|a^i\|_2^2}a^i, \quad k = 0, 1, 2, \ldots. \tag{3.4}$$

As we can see this resembles the general form of the SIRT methods with $M = D$ and $T = I$, where D is defined as $D = \frac{1}{m}\text{diag}\left(\frac{w_i}{\|a^i\|_2^2}\right)$.

3.3.3 Component Averaging (CAV) Method

This leads us to the next method. Since Cimmino's original method uses equal weighting of the contributions from the projections, CAV was introduced as an extension of Cimmino's method, which takes the sparsity of our matrix A into consideration. If we denote the number of nonzero elements in column j by s_j we get that

$$s_j = \text{NNZ}(a_j), \quad \text{j = 1, \ldots, n.}$$

Then the CAV algorithm will take the following form

$$x^{k+1} = x^k + \lambda_k\sum_{i=1}^{m}w_i\frac{b_i - \langle a^i, x^k\rangle}{\|a^i\|_S^2}a^i, \quad k = 0, 1, 2, \ldots, \tag{3.5}$$

where we define a diagonal matrix $S = \text{diag}(s_1, \dots, s_n)$ and the norm $\|a^i\|_S^2 = \langle a^i, Sa^i \rangle = \sum_{j=1}^n a_{ij}^2 s_j$ for $i = 1, \dots, m$. If A is dense then $S = mI$ and we obtain Cimmino's method once again. So the CAV algorithm will take the form of (3.2) with the changes that $T = I$ and $M = D_s$ where $D_s = \text{diag}\left(\frac{w_i}{\|a^i\|_S^2}\right)$.

3.3.4 Diagonally Relaxed Orthogonal Projections (DROP) Method

A further expansion of Cimmino's method is the DROP method. The new feature here is that the factors s_j are incorporated differently, meaning that our next iterate x^{k+1} is found from.

$$x^{k+1} = x^k + \lambda_k S^{-1} \sum_{i=1}^{m} w_i \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|_2^2} a^i. \tag{3.6}$$

To compare the DROP method with (3.2) we now have that $T = S^{-1}$ and $M = mD$, where $D = \frac{1}{m}\text{diag}\left(\frac{w_i}{\|a^i\|_2^2}\right)$. If A is dense we obtain Cimmino's method, where $S^{-1} = m^{-1}I$. Since we have that $T \neq I$, we cannot use Theorem 1, and therefore we will make use of the following theorem.

Theorem 2. *Assume that $\omega_i > 0$ for all $i = 1, \dots, m$. If for all $k \geq 0$,*

$$0 < \epsilon \leq \lambda_k \leq (2 - \epsilon)/max\{\omega_i | i = 1, \dots, m\},$$

where ϵ is an arbitrarily small but fixed constant, then any sequence generated by 3.6 converges to a weighted least squares solution $x^* = argmin\{\|Ax - b\|_D | x \in \mathbb{R}^n\}$. If in addition $x^0 \in R(S^{-1}A^T)$, then x^* is the unique solution of minimum S-norm

The proofs are omitted.

3.3.5 Simultaneous Algebraic Reconstruction Technique (SART) Method

The last method we will be looking at is the SART method. The algorithm is given below.

$$x^{k+1} = x^k \lambda_k D_r^{-1} A^T D_c^{-1} (b - Ax^k). \tag{3.7}$$

Here we have defined the matrices D_r and D_c in terms of the sum taken in the rows or the columns, implying that $D_r = \text{diag}(\|a^i\|_1)$ and $D_c = \text{diag}(\|a_j\|_1)$.

The SART method has $T \neq I$ meaning that we are not guaranteed convergence from Theorem 1. But from [6], we know that it has been proven that convergence is guaranteed for the interval $\lambda_k \in (0, 2)$.

3.4 ART Methods

The last type of methods we will consider are the so-called ART (Algebraic Reconstruction Techniques) methods. Of special interest for these methods is that they perform an update of the iteration, and treat the equations one at the time for all the iterations, whereas we saw that the SIRT methods used all the equations at one time in one iteration.

3.4.1 Kaczmarz's Method

The first type or ART method we will consider is Kaczmarz's method. Below is given the formula for how Kaczmarz's method "sweeps" through the rows in A

$$x \leftarrow x + \lambda_k \frac{b_i - \langle a^i, x \rangle}{\|a^i\|_2^2} a^i, \quad i = 1, 2, \ldots, m. \tag{3.8}$$

Here λ is a relaxation parameter and (3.8) is the typical step in how we update the iteration vector. Kaczmarz method is best used for a fixed $\lambda_k = \lambda \in (0, 2)$. To ensure convergence for the ART methods we have the following theorem.

Theorem 3. *Assume that $0 < \lambda_k < 2$. If the system $Ax \simeq b$, where $A \in R^{m \times n}$ is consistent then the iteration (3.8) converges to a solution x^*, and if $x^0 \in \mathcal{R}(A^T)$ then x^* is the solution of minimum 2-norm. If the system is inconsistent then every subsequence associated with a^i converges, but not necessarily to a least squares solution.*

Furthermore an example of Semi-convergence to Kaczmarz's method is given on figure 3.5 and as we can see from the figure the method fulfills the criterion. This is seen due to the fact that the method converges towards its minimum $x^{k^{opt}}$ and then start diverging.

As an extension to the ART methods, we know that there exist two different Kaczmarz methods, which are the so called Randomized Kaczmarz and the Symmetric Kaczmarz.

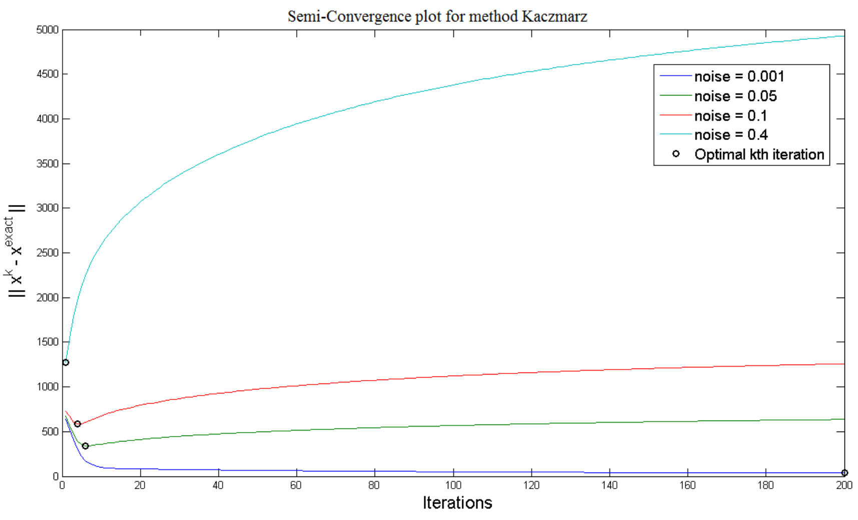


Figure 3.5: Plot of Kaczmarz's method to illustrate semi-convergence. As we can see the method converges to wards a minimum which is our solution. Afterwards the solution diverges.

3.4.2 Symmetric Kaczmarz

As mentioned another variant of the Kaczmarz method is the Symmetric Kaczmarz method. The idea is that after Kaczmarz has done the first "sweep" through our matrix A, it sweeps though the matrix once again, but uses the equations in reverse order. Therefore one iteration for this type of method corresponds to two regular Kaczmarz iterations. Therefore the formula will be the same as 3.8, but now we have a new i corresponding to $i = 1, 2, \ldots, m-1, m, m-1, \ldots, 2$.

3.4.3 Randomized Kaczmarz

This brings us to the last method, namely Randomized Kaczmarz. The formula takes the following form

$$x^{k+1} = x^k + \frac{b_{r(i)} - \langle a^{r(i)}, x^k \rangle}{\|a^{r(i)}\|_2^2} a^{r(i)}, \tag{3.9}$$

where the index $r(i)$ is chosen at random from the set $1, 2, \ldots, m$ with probability proportional with $\|a^{r(i)}\|_2^2$. Also it follows that no convergence result exists for the relaxation parameter, and therefore a default value of $\lambda_k = 1$ is chosen.

A semi-convergence plot has been made for all the ART methods on figure 3.6. As we can see the semi-convergence criterion is fulfilled. Due to the fact that Kaczmarz's method seems to find a lower relative error, we will make use of this as ART method throughout the report.

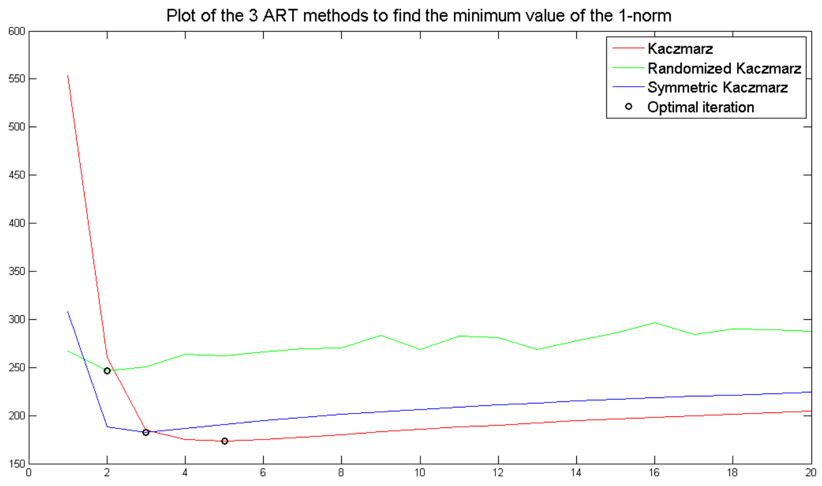


Figure 3.6: Plot of the different ART methods to illustrate semi-convergence. As we can see the criterion is fulfilled for all of the methods.

3.4.4 Default Values of λ_k

For all the methods, different values of the relaxation parameter have been implemented as a default value. Throughout this report we do not consider different values of λ_k and therefore the default value will always be used. Below is listed the different default values which are found from [6].

$$\lambda_k^{Landweber} = 1/\hat{\sigma}_1^2,$$

where $\hat{\sigma}_1$ is an estimate of the largest singular value of A.

$$\lambda_k^{Cimmino} = 1/\hat{\sigma}_1^2,$$

where $\hat{\sigma}_1$ is an estimate of the largest singular value of $M^{\frac{1}{2}A}$, where $M = \frac{2\omega_i}{m}\text{diag}(\|A(i,:)\|_2^{-2})$ for $i = 1, \dots, m$.

$$\lambda_k^{CAV} = 1/\hat{\sigma}_1^2,$$

where $\hat{\sigma}_1$ is an estimate of the largest singular value of $D_S^{\frac{1}{2}}A$, where $D_S = \sum_{j=1}^n s_j a_{1j}^2, \dots, \sum_{j=1}^n s_j a_{mj}^2$, and s_j is the number of nonzero elements in column j of A.

$$\lambda_k^{DROP} = 1/\hat{\rho},$$

where $\hat{\rho}$ is an estimate of the spectral radius of $S^{-1}A^T DA$. Here $S^{-1} = \text{diag}(s_j^{-1}$ and s_j is the number of nonzero elements in column j of A. $D = \text{diag}(\frac{\omega_i}{\|A(i,:)\|_2^2})$ for $i = 1, \dots, m$.

$$\lambda_k^{SART} = 1/\hat{\rho},$$

where $\hat{\rho}$ is an estimate of the spectral radius of $V^{-1}A^T WA$. Here $V = \text{diag}(\sum_{i=1}^m a_j^i)$ for $j = 1, \dots, n$ and $W = \text{diag}(\frac{1}{\sum_{j=1}^n a_j^i})$ for $i = 1, \dots, m$.

$$\lambda_k^{Kaczmarz} = 0.25.$$

$$\lambda_k^{R,Kaczmarz} = 1.$$

With this we are now ready to go more into detail with how tomography works and what it is. But first we need to take a look at how we can compare the different methods.

3.5 Measuring Time

Throughout this report different measurements will be made to compare the methods with each other. One important feature is that we cannot directly measure in minutes and seconds whether one method is faster than another. The reason for this is, that the ART method, Kaczmarz, uses a for-loop in MATLAB, which takes a huge amount of time to compute. If this was implemented in e.g. C++, the iterations would take the same amount of time to compute. Therefore our measure of time in this report will be given in how many iterations the different methods use. The reason for this is due to the fact that in each iteration we have a matrix-vector multiplication and some "other stuff" going on, but that multiplication is what takes time to perform. So later on we are doing measurements of the quality of the reconstructions. If we have

$$Method_1 \text{ used } k_1 \text{ iterations }, \quad Method_2 \text{ used } k_2 \text{ iterations }, \quad k_1 \ll k_2,$$

then it does not imply that $Method_1$ is better than $Method_2$. It only implies that $Method_1$ finds its optimal reconstruction quicker than the optimal reconstruction found with $Method_2$.

As a final note it should be mentioned that the methods Cimmino, Landweber, Kaczmarz, DROP, CAV, SART and CGLS all use 2 matrix-vector multiplications per iteration. Only Symmetric Kaczmarz uses 4 for comparison. This is also the reason that we represent the ART methods by Kaczmarz only. Symmetric Kaczmarz converged towards the same relative error as Kaczmarz from the semi-convergence plot, see figure 3.6. But Symmetric Kaczmarz takes twice the time to compute, and therefore we will only consider Kaczmarz.

3.6 Summary

With this chapter, we have taken a deeper step towards understanding the methods we shall make use of. First a definition of an iterative method was given. With this followed a phenomenon called semi-convergence, which we need to have fulfilled to ensure convergence for our iterative methods. All the equations for the iterative methods have been explained, and the concept of "time measure" has been discussed. With this we have a greater understanding of the underlying in the iterative methods, and can proceed to the concept Tomography.

Chapter 4

Tomography

In this chapter it will be explained what tomography is and how a tomography problem arises. The chapter is based on [2].

4.1 Explanation of Tomography

Tomography is about seeing or observing what the naked eye cannot see. Especially in the medical industry tomography is widely used to reconstruct images. As an example take a tumor in the human body. The tumor is barely visible, but with e.g. CT-scanners, which use tomography, one is able to see the position of a tumor in a human body.

In this report we are mainly concerned with applications of tomography to examine the structure of different materials. Many materials and metals found in the nature consist of so-called "grains" with a certain form of crystal-like structure. Whenever one performs tests on metal, e.g. bending an iron rod, it is of interest to see what is happening to these grains and the structure without damaging the material by e.g. cutting the material. Therefore tomography is of interest, since with this technique we might be able to observe changes in the material without damaging it.

So how does tomography work? X-rays are send through a material in certain specified directions, and from projections of these X-rays, in this case it would be data that is obtained by integration along these X-rays, we are able to reconstruct what is inside a 2D or 3D image. We will go more into detail with the mathematical aspect of tomography in the next section.

4.2 Mathematical Interpretation of Tomography

We shall see how tomography can be applied to our case. This section is based on [2], and a classic example of tomography is shown on figure 4.1. As we can see, the X-rays penetrate our object and by projection of these rays, we obtain data where the value inside this data corresponds to how much ”material” the specific ray has penetrated. The idea is now that we send a lot of X-rays through the material and then gather all the information received from the rays, to represent the object we penetrated.

We once again consider figure 4.1. Here the red numbers indicate our domain Ω, which is an $[0,1]\times[0,1]$ square, and the numbers represent an arbitrary length from 0 to 1. The blue numbers indicate how the problem is discretized, but more on that later. Inside this domain we wish to find an unknown function $f(t_1,t_2)$, where we assume that this function represents the density or the absorption coefficient of the material that the ray has passed on its path. From the figure, ray_2 would be damped since it has passed through some material (green circle) whereas ray_1 would not be damped at all since it did not pass through anything. So ray_1 would have a value equal to 0 and ray_2 would be greater than 0.

We now direct our attention towards the rays, to see how these can be expressed in such a way that they can satisfy the set of equations $Ax = b$. We have the following assumption about our unknown function $f(t_1,t_2)$

We assume that this function represents some material parameter, in such a way that the damping of a signal penetrating an infinitesimally small part $d\tau$ of a ray at position t is proportional to the product to $f(t)d\tau$.

Given the above assumption, we are able to locate all the points that lie within the ray. This is given by the formula

$$t^i(\tau) = t^{i,0} + \tau d^i, \quad \tau \in \mathbb{R}.$$

Here t^i indicates a given point on ray_i, $t^{i,0}$ indicates an arbitrary point on ray_i and d^i is a unit vector lying in the direction of ray_i, ensuring that we move along

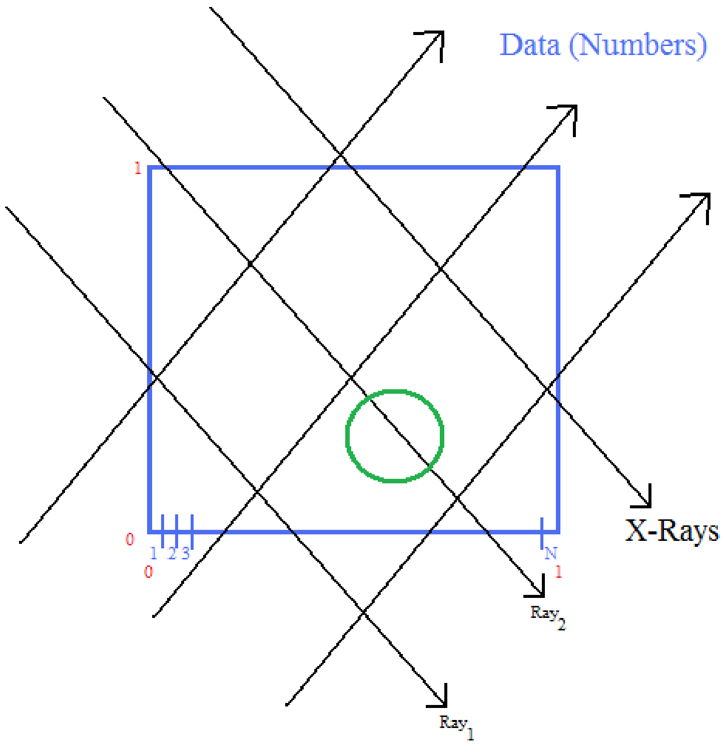


Figure 4.1: An illustration of how tomography works. The red numbers indicate our domain which we are discretizing into pixels or voxels for the 3D case. In this case the domain is $\Omega = [0\ 1] \times [0\ 1]$. The black lines corresponds to the X-rays we send through the domain. The blue numbers tell how we discretize our domain into pixels, and the green circle indicates the object within our domain we wish to locate with our X-rays.

the correct ray. By this formula and the assumption, the material penetrated by a ray is proportional to the integral $f(t)$ along the ray, and therefore we arrive at the equation

$$b_i = \int_{-\infty}^{\infty} f(t^i(\tau))d\tau, \quad i = 1, \dots, m$$

where b_i corresponds to the left hand side of equation (2.1), which we wish to reconstruct, m corresponds to the total number of measurements and $d\tau$ denotes a differential along the ray.

The next step is to discretize the integral so we can implement it on a computer. First our domain Ω is divided into $N \times N$ pixels. The function $f(t)$ is assumed to be a constant, f_{kl}, inside every pixel (k, l), meaning that we have

$$f(t) = f_{kl} \quad \text{for} \quad t_1 \in N_k^1 \text{and } t_2 \in N_l^2. \tag{4.1}$$

This is illustrated in figure 4.2. Now with the assumption that the function $f(t)$

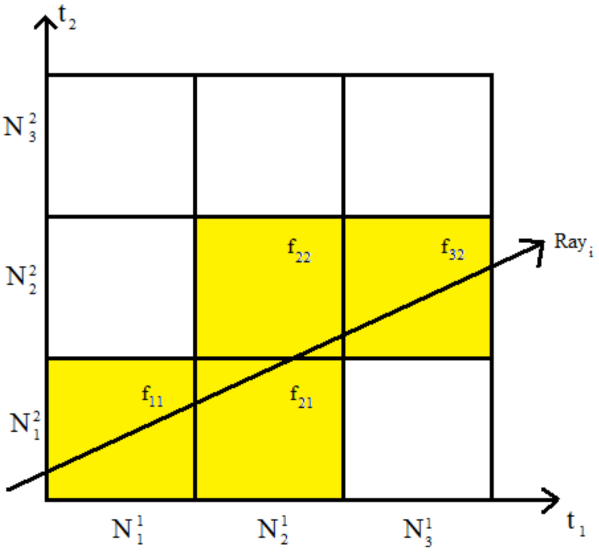


Figure 4.2: An example of how a domain Ω can be discretized. Here $N = 3$ and the ray intersects 4 points corresponding to the yellow pixels. The domain is discretized into 3 pixels on each axis, indicating that we have a total of 9 pixels.

is piecewise constant, we get the following expression for the kth measurement:

$$\begin{aligned} b_i &= \sum_{(k,l)\in ray^i} f_{kl}\Delta L_{kl}^{(i)}, \quad i = 1, \ldots, m, \text{ where} \\ \Delta L_{kl}^{(i)} &= \text{length of } ray_i \text{ in pixel } (k, l) \end{aligned}$$

So, by looking at our unknown function $f(t)$, we realize that we have a total of $N \times N$ unknown values, which equals $f(k, l)$. These corresponds to the number of pixels inside our domain. We then arrange all the unknown values $f(k, l)$ in a vector x in such a way that our new vector is equal to

$$x_j = f_{kl}, \quad \text{with the ordering } j = (l - 1)N + k.$$

By doing this our new vector x_j corresponds to take the second column and stack it under the first, take the third and stack it under the second, and so on.

Hereby we obtain a linear system of equations and can therefore always write

$$b_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1, \ldots, m, \quad n = 1, \ldots, N^2.$$

From this we end up with what we wanted to, and that is to show how tomography can be rewritten into something on the form from (2.1). We end up with an $m \times n$ matrix whose elements are given by

$$a_{ij} = \begin{cases} \Delta L_{kl}^{(i)} & ,(k,l) \in ray_i, \\ 0 & ,\text{else} \end{cases}$$

So by putting it in words the matrix b are our measurements, x consists of our solution to the problem and A will be built in the following. The ith row corresponds to the ith ray and j indicates which pixel we are within. We once again consider figure 4.2. In this case the first row in A would look like the following

$$A_{1j} = \begin{pmatrix} x & 0 & 0 & x & x & 0 & 0 & x & 0 \end{pmatrix},$$

where x corresponds to a value different from 0, since the ith ray penetrates this pixel. As one can imagine, if N is large (e.g. 50) the matrix A will contain of a lot of zeroes, meaning that we have a very sparse matrix. This is an advantage we will exploit.

4.3 Matlab-Code for 2D and 3D Problems

In this section it will be explained how the code for setting up the tomography test problems works, and it will be shortly described which variables there are and how they influence the test problems. Furthermore it should be mentioned that all the theory applied above for tomography is basically the same for 2D and 3D.

4.3.1 2D Tomography

For generating test problems for a tomography problem in 2D, the main point of view is taken from the MATLAB-code *paralleltomo.m* which is found in AIRtools from [13]. There are also other ways of creating a tomography test problem, e.g by *seismictomo.m* and *fanbeamtomo.m*, but in this project we only consider *paralleltomo*. The program has been modified for this project with some new features, but these features will be described later. The basic call for *paralleltomo* is

[A b x theta p w] = paralleltomo(N,theta,p,w,isDisp).

N is the size of the image, p is the number of rays we send through our image given the angle *theta*. As an example, if $p = 5$ and $theta = [0 \quad 90]$, 5 rays will be send through our object at an angle equal to 0 and 90. w indicates the length between the first and the last ray, and *isDisp* is a option that, if chosen, displays how the rays will be created. At an angle equal to 0 the rays will be vertical, going from top to bottom. For a clarification see figure 4.3.

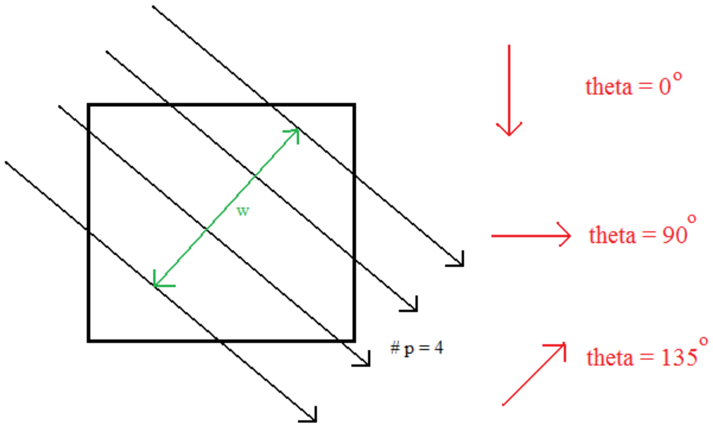


Figure 4.3: Illustration of how *paralleltomo* works. As we can see w corresponds to the length between the first and the last ray. The black rays indicate the actual number of rays we have, which is p. The red arrows indicate in which direction we are sending our rays, that is θ. As we can see zero degrees correspond to an arrow going from top to down.

Throughout this report, when speaking about "standard variables" for the 2D reconstructions, this implies that $N = 100$, $\theta = 0 : 1 : 179$, $p = round(\sqrt{2}N)$ and $w = \sqrt{2}N$ has been chosen. Furthermore, for the test problem shown in the Picard plots, a test problem was generated with $N = 32$, $\theta = 0 : 1 : 179$, $p = round(\sqrt{2}N)$ and $w = \sqrt{2}N$.

The tests for these variables will be described in the next chapter. For now it is only important to know how the algorithm works. The setup of the matrix A is basically the same in 2D as it is for 3D. Therefore it will be explained in the 3D section.

4.3.2 Tomography 3D

When we generate a test problem in tomography for 3D, the main point of view is taken from Tomobox from [10] and [3]. The most important functions from this package which we will consider are *buildSystemMatrix.m* and *traceRays.m* with special interest on *traceRays*. The matrix A is constructed in principle in 3D in the same way as for 2D, with the exception that in 3D our rays penetrate a voxel (cube), and in 2D our rays penetrate a pixel (square). But how exactly this matrix is set up will be explained in a further section. First let us have a look at how the rays will be created. See figure 4.4.

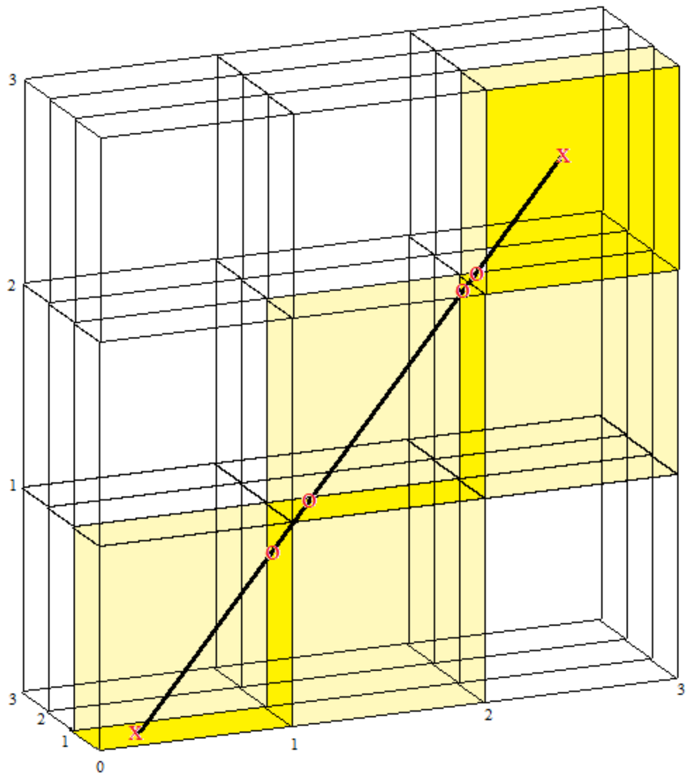


Figure 4.4: Illustration of how a line penetrates the voxels in 3D. The red X's indicate where our ray starts and ends. The red O's indicate where we hit an edge of our voxels. The transparent edges indicate which voxel we have hit, and the yellow edges indicate which edge of the grain we are going through. The black line corresponds to a ray.

Once again all the ray_i's are being sent through our object, but instead of sending them with a given angle θ, which we did in 2D, we send them with a unit direction vector instead, defined as $(a, b, c)^T$. The point is that we still have to find the length of the line, which hits the target voxel. To calculate this distance, we need the intersection points on each boundary of the voxels. To illustrate this see figure 4.5.

On the figure it is seen that at some value, g, we should find the intersection

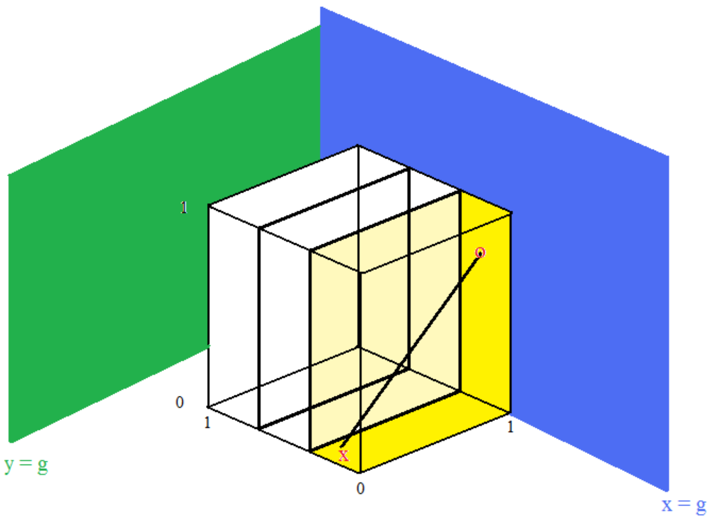


Figure 4.5: Shows a voxel from figure 4.4. The figure illustrates how we find the length of ray_i given the intersection points. The intersection points corresponds to the red O's and the black X is where our ray initiates from. The transparent edges show which voxel we are hitting, and the yellow edges indicate the actual edge of our voxel we are going through. The blue and green planes are the planes through which we calculate the intersection points. From this we can calculate the distance that the ray has moved through our voxel. The black line indicate a ray.

between the plane and the line, given by $x = g, y = g$ or $z = g$, $g = 1, \ldots, N$. We notice that this value g corresponds to the discretization of a voxel, indicating that it also corresponds to the boundary of one voxel. In figure 4.5 the first voxel has been taken out. The call for *traceRays* is listed below

$$[\text{voxels,vals,rows,xxyz,yxyz,zxyz}] = \text{traceRays(x0,y0,z0,abc,dims,Imod)}.$$

Here $dims$ is the dimension of the cube, which means it will correspond to $N \times N \times N$. The $\overrightarrow{abc}$ corresponds to the given direction vector for all the lines and $x0, y0, z0$ corresponds to $P = (x0, y0, z0)$ where P is a point on ray_i. The last variable $Imod$ is a input variable which ensures efficiency if *traceRays* is called multiple times. The default value ensures this efficiency, and therefore it is always set to default. $voxels$, $vals$ and $rows$ tell us which voxels are hit, the length of the path of the rays and which row and column the corresponding voxels are in. $xxyz$, $yxyz$ and $zxyz$ contain the intersection points between the rays and the boundary of the voxels. The point that ensures that $ray_i \neq ray_j$, for $i \neq j$ is therefore the point P, which indicates $P_i \neq P_j$ for $i \neq j$. Then for every point P, all the rays are being sent through every given direction vector.

Now given the point P, and the direction vector of the line, we can write down the parameter representation for the line for which we need to find the intersection points,

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = P + t \cdot \overrightarrow{abc}, \quad t \in \mathbb{R}. \tag{4.2}$$

To find the intersection points, we only need to insert the formula for the plane $(x = g), (y = g)$ or $(z = g)$ into (4.2). This is what is done in *traceRays*.

Now we have found all the intersection points, all the path lengths of the rays and which voxels they hit. This leads us to the next step, which is to construct the matrix A. *traceRays* is a "helping function" to *buildSystemMatrix*, which sets up the matrix A. The call for the function is

[A,p_all] = buildSystemMatrix(r1_max,u_max,v_list,nuv,vpRatio).

Here $r1\_max$ corresponds to our N, which is the size of our discretization. In this case $N = r1\_max \cdot 2 + 1$, which of course indicates that we have a total of $(r1\_max \cdot 2 + 1)^3$ voxels. v_list contains all our direction vectors, and the length of v_list corresponds to the number of projections, m. The variable u_max indicates how we discretize our projections since every projection, m, is discretized into a 2D plane consisting of $(2 \cdot u\_max + 1)^2$ pixels. *nuv* and *vpRatio* are set to default and used with this value throughout the report. The output variables are somewhat obvious, since A corresponds to our sought matrix A. p_all is a three-dimensional array where the k'th layer contains the k'th center pixels corresponding to the k'th projection.

Since we now know how to construct the matrix A, and we know what the different values corresponds to in 2D and 3D, then we can perform the tests in 2D or 3D and do comparisons of the variables. The last thing we need is to learn how v_list is being created, which we will look into in the next section.

4.3.3 Setting up *v_list*

We want to obtain a certain structure in the way the rays are sent through the object, and for this we will use a function called *getLebedevSphere.m* from [5]. For an illustration see figure 4.6. The objective is to create the direction vectors $\overrightarrow{abc}$ in a proper way, therefore we should not generate them in a random way. From all our starting points P, from (4.2), we now want to "fill" out a sphere as densely as possible with these directions. This indicates that we should have the same amount of directions for every axis. As an example, if we send 10 rays through our 3D object, it would be bad if all had negative z-axis value, since we would not hit any part of the object that lies above the P_z index. From the function *getLebedevSphere.m*, there is an input parameter called *degree*. This value corresponds to how densely the direction vectors are created, and by changing this we can choose how many different direction vectors we are sending out through our sphere. Figure 4.6 illustrates 4 rays being sent through the sphere (it is an example so it might not be the optimal way for $degree = 4$), and their corresponding 2D planes, which correspond to the right hand side b. Not all values are accepted for *degree*, and to find out which values are acceptable, we refer to the code found at [11].

Throughout this report, when speaking about "standard variables" for the 3D test problems, we imply that $r1\_max = 17$, $u\_max = 23$ and $degree = 38$ has been chosen. Now everything should be set and ready for creating our test grains. In the next section we will look upon the geometrical aspects of these grains.

4.4 Geometric Aspects of a Grain

As mentioned early in the report, we are specially interested in the investigation of these so-called metal grains and their behavior. Therefore we need a program that can create such grains. For this we have the functions called *grain2d.m* and *grain3d.m*. The concept of a grain is, that when looking at an image of a grain, in MATLAB, the grain will consist of a value equal to 1, and the surroundings are set to 0, namely the background. This is a feature which we want to exploit, and therefore we need some geometric definitions before we can construct the grain. This fact only applies for our test problems. In the real world we do not know the value of the grain, but we know that every value inside a grain is equal to a constant, whereas the background is equal to 0. Recall the equation for a plane in 3D given by

$$l_{3D} : ax + by + cz + d = 0. \tag{4.3}$$

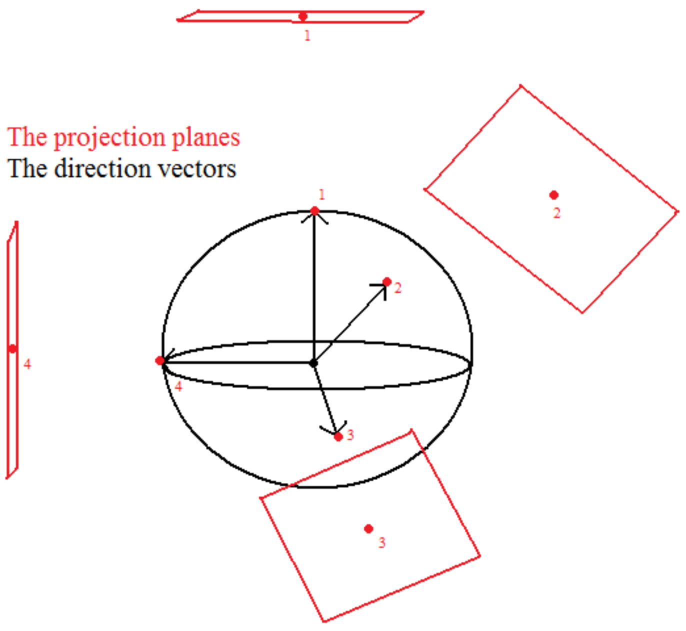


Figure 4.6: Example of our direction vectors and their corresponding projection planes. In this case we are sending our rays in 4 different directions. The red planes illustrates how we "capture" the rays to obtain the data.

In 2D this can be simplified by removing the z-term, meaning we end up with the following equation for a line

$$l_{2D} : ax + by + c = 0. \tag{4.4}$$

Furthermore the way to find a line in 2D or a plane in 3D is given by the following formula

$$\text{Equation of a line/plane } = \underline{\underline{n_1}} \cdot (x - x_1), \tag{4.5}$$

where $x = (x, y)^T$ in 2D, $x = (x, y, z)^T$ in 3D, $x_1 =$ Known point on our line or plane and $\underline{\underline{n_1}}$ is the normal to our line or plane.

Now that we have our definitions we turn our attention to figure 4.7 and figure 4.8. Figure 4.7 shows geometrically how the grain will be created. We search for a line l when certain variables are given. These variables consist of a vector $\underline{\underline{n_1}} = \begin{pmatrix} a \\ b \end{pmatrix}$, where a and b corresponds to the constants from (4.4). $\underline{\underline{x_0}}$ is the vector to the center point, $\underline{\underline{x'}}$ is the vector to the point where the line l is

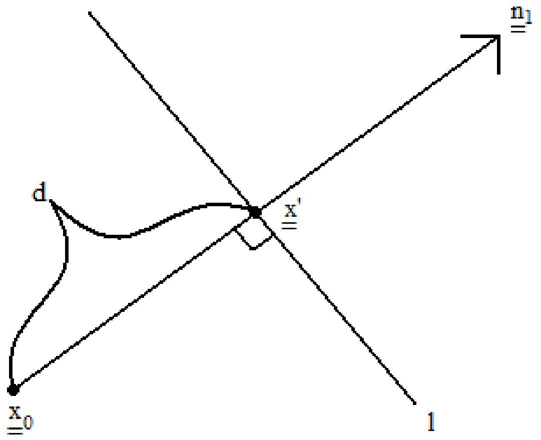

$\underline{\underline{n}}_1$
d
$\underline{\underline{x}}'$
$\underline{\underline{x}}_0$
l

Figure 4.7: Illustration of the geometric aspect of how a grain should be created. The principe is exactly the same for 2D as it is for 3D. Here l is a given line in 2D, $\underline{\underline{n_1}}$ is the normal to the line l, d corresponds to the length of vector $\underline{\underline{x'}} - \underline{\underline{x_0}}$, $\underline{\underline{x_0}}$ is our center point and $\underline{\underline{x'}}$ is the point where l is perpendicular to the vector $\underline{\underline{x'}} - \underline{\underline{x_0}}$. All these variables are used to create 1 edge in our grain in 2D or to create one plane of our grain in 3D which corresponds to a side of the grain.

perpendicular to the vector $\underline{\underline{x'}}-\underline{\underline{x_0}}$, and $d\cdot|\underline{\underline{x_0 x'}}|$ is the length of the vector $\underline{\underline{x'}}-\underline{\underline{x_0}}$.

Figure 4.8 illustrates how the grain will be created, and the idea is that everything below a given line l will be set to 1 and everything above will be set to 0. From the figure it follows that all the green area would be equal to 1 whereas all the red area would be equal to 0. Hereby we end up with a grain consisting of only 1's in MATLAB and zeroes everywhere else. This is our definition of the grain. The point $\underline{\underline{x'}}$ can be found by the 3 other unknowns in the following way

$$\underline{\underline{x'}} = x_0 + \frac{d}{\|\underline{\underline{n_1}}\|} \cdot \underline{\underline{n_1}}.$$

By substituting this into (4.5) we obtain the equation for the line l we seek:

$$\begin{aligned} \underline{\underline{n_1}} \cdot (x - x_1) \quad &\Leftrightarrow \quad \underline{\underline{n_1}} \cdot x - \underline{\underline{n_1}} \cdot x_1 \\ &\Leftrightarrow \quad \underbrace{\underline{\underline{n_1}} \cdot x}_{\text{The Slope}} \quad \underbrace{-\underline{\underline{n_1}} \cdot x_0 - d\|\underline{\underline{n_1}}\|}_{\text{The Constant}}. \end{aligned}$$

If we want to be able to create n lines, we "only" need the vector $\underline{\underline{n_1}}$, the length d and the center point x_0. Note that exactly the same construction can be

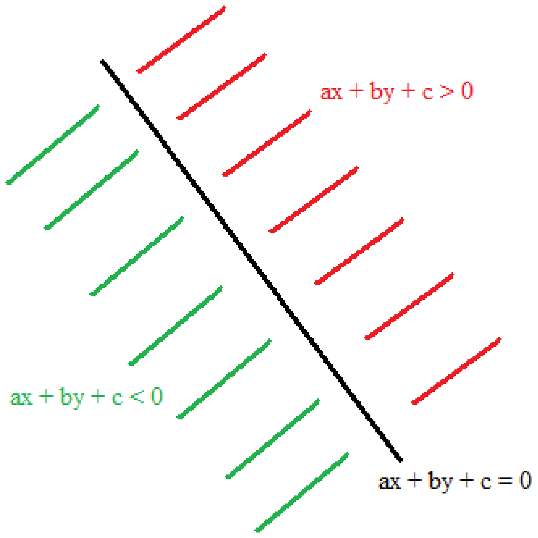


Figure 4.8: Illustration of the geometric aspect of how a grain should be created. The principe is exactly the same for 2D as it is for 3D. When we have created a line in 2D or a plane in 3D, all pixels or voxels below this will be set to 1 (the green area). Everything else will be set to 0 (the red area).

applied in 3D. The only difference is a z-term, which from (4.3) is equal to cz, must be added.

4.5 Creation of a Grain in Matlab

We want to use the three user-specific inputs above, n, x_0 and d, in a program that should be able to create an n-edged figure. Therefore we formulate the following call for our *grain2d* function:

$$\text{X} = \text{grain2d(N,x0\_x,x0\_y,n,scalar)}.$$

The code for *grain2d.m* can be found in Appendix C. Here N is the size of the image, $(x0\_x, x0\_y)$ denotes the vector $\underline{\underline{x_0}}$, namely where the center of the grain will be placed. The value is a scalar between 0 and 1, and the number is independent of N, meaning that if $(x0\_x, x0\_y) = (\frac{1}{2}, \frac{1}{2})$, then the center of the figure will always be in the center of the image, no matter the value of N. n corresponds to how many edges we want on our grain, e.g. 3, 5 or 20. The last variable, *scalar*, indicates how big the figure should be compared with the image, since it reflects the length variable d from 4.7. It should be set between 0 and 1, but for small values of n the image can appear big, since there are "fewer" planes to create the grain. Examples of what the different grains can look like are shown on figure 4.9. When talking about "standard values" throughout this report $scalar = 0.35$, $\underline{\underline{x_0}} = (0.5, 0.5)$ and $n = 3$.

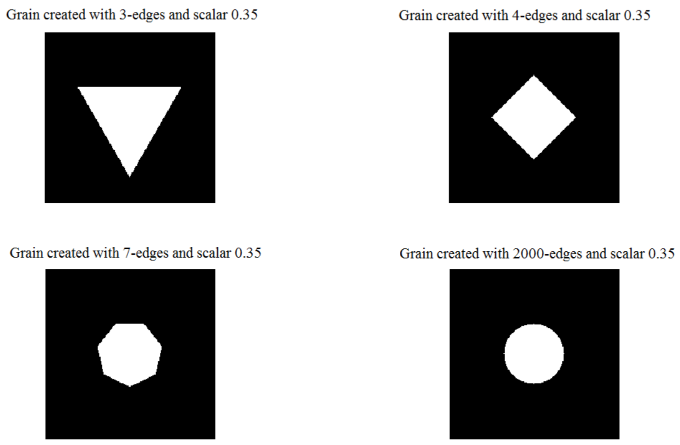

Grain created with 3-edges and scalar 0.35
Grain created with 4-edges and scalar 0.35
Grain created with 7-edges and scalar 0.35
Grain created with 2000-edges and scalar 0.35

Figure 4.9: Examples of different types of grains. Top left is a triangle where $n = 3$. Top right is a square where $n = 4$. Down left is a 7-edged figure where $n = 7$. Down right shows a circle where $n = 2000$. These are examples of different grain types which we can create.

As one notices, the grains are uniform meaning that all the edges are of equal length. This is done on purpose, and therefore all the figures in this report will have the same side length. Figure 4.10 illustrates how it has been implemented. In practice, we divide a circle (360 degrees) up into the specified number n of edges, and then we let an edge span over $\frac{360^\circ}{n}$ degrees. Hereby we obtain n edges of equal length. Note that in 3D the only difference in the construction from the one in 2D, is that there is one more input $x0\_z$, which of course defines the z-coordinate of the center point. Also in 3D it can be shown that there exist only 5 regular polyeder. The proof is omitted here, but it is shown by Eulers theorem regarding polyeder in [14]. These 5 polyeder consist of $4, 6, 8, 12$ and 20 edges, and examples of these figures can be seen in Appendix B.

All of the polyeder have been created with some code from Maple. The code can be found in Appendix A. The package *with*(*geom*3*d*) has been used, and

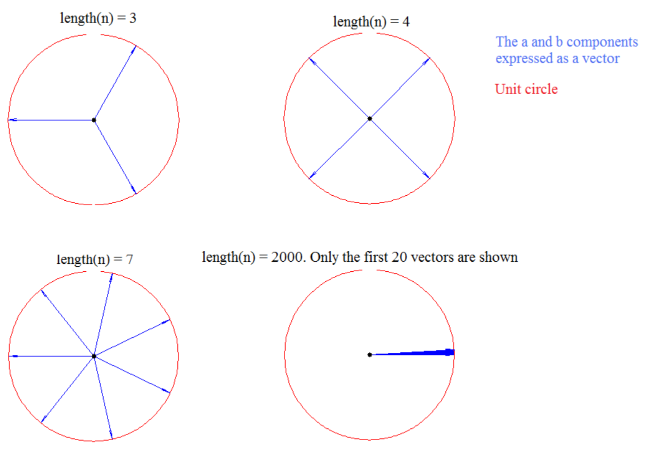
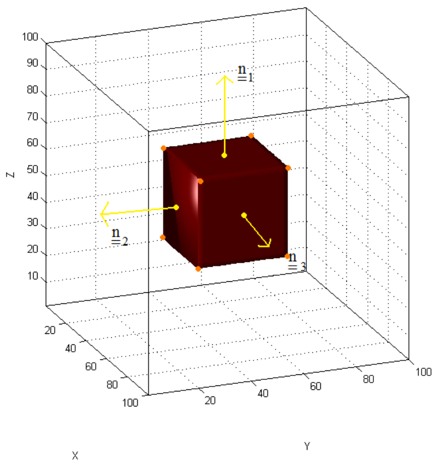

length(n) = 3
length(n) = 4
The a and b components expressed as a vector
Unit circle
length(n) = 7
length(n) = 2000. Only the first 20 vectors are shown

Figure 4.10: Illustration of how we create the different direction vectors in 2D. As one can see the vectors have equal distance between each other and they span over the same amount of degrees. These vectors were used to create the 4 figures from figure 4.9

in this package there is a command, which can give the coordinates to a regular polyeder specified by the user. Figure 4.11 illustrates the concept that has been used. The idea is simple. We need to calculate the normal to all the planes that combined creates the figure we seek. On 4.11 our 3D-example is a cube, indicating that we need to find 6 normals. The command in MAPLE gives the coordinates to the corner points, in this example it would be the 7 (8 total) points corresponding to the orange dots, and from these corner points it is easy to calculate the vectors that span the sought planes inside. When these vectors are found, we know from basic geometry that the vector product of two vectors yields another vector that is normal to the plane, which the two vectors span. The normals are what we seek from (4.5), and this concept has been implemented in the code *grain3d.m*. The code can be found in Appendix C

Given the different variables we are now ready to create our grains. First we look upon the different measurement methods, that can determine if a grain or a reconstruction of a grain is appropriate or not.



$\underline{\underline{n}}_1$
$\underline{\underline{n}}_2$
$\underline{\underline{n}}_3$
Z
X
Y

Figure 4.11: Illustrates which normals we seek to create our grain. In this example we have a cube, and the orange lines indicate the normals we are seeking. When these normals are known we can create our grain.

4.6 Summary

With this chapter done, we should be able to understand most of the theory underlying this project. It has been shown what tomography consists of, and how it is used for creation of test problems. The mathematical aspect of tomography has been shown, and this was further used to set up the equations, which we wish to solve. Code already existed in order to simulate a tomography 2D and 3D test problem, and explanation of this code has been made. Furthermore it has been discussed how our grains should be created in both 2D and 3D, and a special way of setting up our direction vectors was made with help from the function *getLebedevSphere*. We are now ready to start creating our test problems.

Chapter 5

Quality Measurements of the Reconstructions

As mentioned before, a problem with the storage and computation time can arise when our problem from (2.1) becomes too big. Therefore we need to ensure that a proper choice of variables are specified, so we do not waste any time on what might appear to be useless computation. The point is that the size of A is proportional to the number of rays being sent through our object for each given angle. If we somehow can choose these parameters in such a way that the size of A will be reduced, without ruining the reconstruction, we could save computation time when reconstructing. But before looking into these parameters, we need to take a look at different constraints and their properties, and also find a good measure to determine when our reconstruction is "good enough". It would not be advisable to start making reconstructions if we did not have some methods to determine if these reconstructions are good or bad. Therefore we first will look into some of these measurement methods.

5.1 Sharpness of the Reconstructions

As mentioned, we need a value to indicate how sharp an edge in our reconstruction is, to determine if a reconstruction is sharp or not. To do this we have defined a value max_{slope} which is best illustrated by an example. Lets consider figure 5.1. This figure shows an edge taken out from two reconstructions and the exact image. Standard values of the variables has been used for these examples. The top figure illustrates an edge found from the exact image. The middle figure shows an edge from a reconstruction made with $noiselevel = 0.1$ and the bottom figure shows the same as the middle figure, but the reconstruction was made with $noiselevel = 0.6$. As we can see from these figure, the more noise we add to the image, the more the slope of the edge will be spread out. Therefore when we make these reconstructions, we should have a measure that shows how steep our slope is, indicating that we have a value showing the sharpness of our reconstructions.

With this we introduce the term max_{slope} which corresponds to the maximum slope value found within the figures from 5.1. From these figures it follows that the top figure should have the highest value, the middle figure should have a lower value of max_{slope} compared to the top figure, and the bottom figure should have the lowest value of max_{slope}.

Since we have the original image we can find the "optimal" value of max_{slope} from the original edge, which we can use for comparison of the reconstructions. The closer the value of max_{slope} is to the "optimal" value, the better our reconstruction is, when measuring the sharpness of our reconstructions.

The problem now is to define a value as a criterion for when our reconstructions are sharp. To find an estimate of the slope in the data points, we can fit a function to these points, and then obtain the parameters that are needed to calculate the slope. For this we use MATLAB's incorporated function *polyfit.m*. The call for *polyfit* is the following:

$$\text{p} = \text{polyfit(x,y,n2)},$$

where the function finds a polynomial, $p(x)$, of degree $n2$, that fits the data y in a least squares sense. p is a vector that contains the coefficients to the polynomial. Since we wish to fit a straight line to our data points, we need at least 3 points since we wish to fit

$$p_1 \cdot x + p_2. \tag{5.1}$$

If the amount of points is the same as the number of parameters which we wish to estimate, then the function will interpolate the data instead of fitting it. When the parameters are found, then the value p_1 is the one we seek, since equation (5.1) differentiated gives the exact slope, which corresponds to p_1.

Now let us have a look at what our values of max_{slope} should be. Once again we consider the top figure in 5.1. The slope in this figure consists of a jump

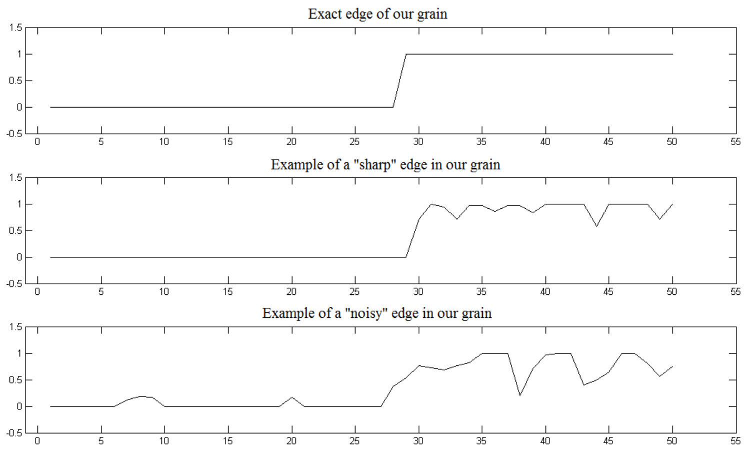


Figure 5.1: Illustration of what our value max_{slope} should measure. The top figure is an example of an edge taken from an original grain (triangle). The middle figure is an edge taken from a reconstruction where the relative noise level was 0.1. The bottom figure is an edge taken from a reconstruction made with a relative noise level equal to 0.6.

between 0 and 1. Since we use 3 points in fitting, our slope estimate will consist of two parts, where one part is found as the slope from vec_1 and the other part is found as the slope in vec_2. These two vectors will in this example be given as $vec_1 = [0 \quad 0 \quad 1]$ and $vec_2 = [0 \quad 1 \quad 1]$. Everywhere else the slope estimate will be equal to 0 and this is out of interest. By calling *polyfit*, $p = polyfit(1{:}3, vec_i, n), \quad i = 1, 2$, we end up with a slope corresponding to $p_1 = \frac{1}{2}$. As we can see, the correct slope is 1 for this test problem, due to

$$max_{slope} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{1-0}{1-0} = 1.$$

Therefore we now assume that the slope is found by some part of two vectors, vec_1 and vec_2. When doing this assumption, we can find the estimate of the slope by adding the two vector terms together, ending up with

$$slope_{vec_1} + slope_{vec_2} = \frac{1}{2} + \frac{1}{2} = 1,$$

which actually corresponds to the exact slope for this example. Had we taken some of the other neighboring slope values, we would get $\frac{1}{2} + 0$. This indicates that when we have found the maximum slope estimate, and we know that a term from one of the neighboring points should be added, we assume that if we take the largest neighboring term, we will have a good estimate of the slope. From the example above we saw that we should say $\frac{1}{2} + \frac{1}{2}$ instead of $\frac{1}{2} + 0$.

This is implemented in MATLAB. If we consider the middle and the bottom figure in figure 5.1, the idea is the following. We start by taking out a single half row-vector, *vec*, out of our reconstruction, see 5.2. This is due to the fact that we know where we have placed our grain (in the middle), and then we know that by going from the center and out, we will hit an edge. When this is done, we do exactly the same for the 5 neighboring vectors (on both sides) and take a mean of these 11 values. This should ensure that we have a usefull measure of the edge.

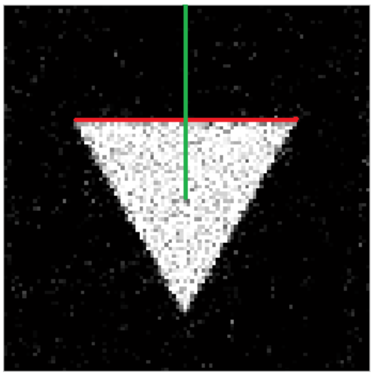


Figure 5.2: Illustrates the edge we are trying to find the slope of, and how we place our "half vector". The red line shows the edge we are using to find max_{slope} and the green line shows which vector we are taking out of the reconstruction.

Now we have some data points stored in a vector, the green line in 5.2, and from this vector we create a new vector, *slope_vec*, which has all the slope values in it. These values are created by taking out an index in *vec* with its 2 neighboring indexes and then make usage of *polyfit* on these 3 values. We hereafter find the maximum value of *slope_vec* by the function *max* in MATLAB. When the maximum value of max_{slope} has been found we locate which index i, it has. By doing this we can locate the neighboring points by the two indexes $i - 1$ and $i + 1$, and then we sort the 3 obtained values by descend. As we assumed earlier the slope was found over two slope terms, indicating that the slope found at the i'th pixel was not enough. We should add the largest neighboring slope value to obtain max_{slope}. For illustration of this concept see 5.3.

As an example of how this value resembles our reconstructions we refer to

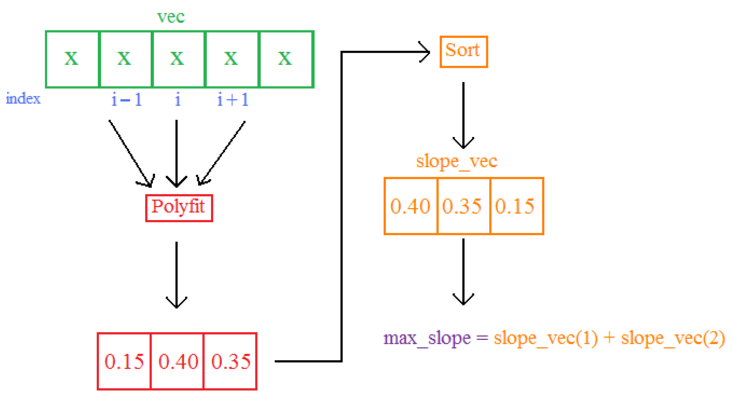


Figure 5.3: Illustrates how we find the value max_{slope} in MATLAB. We start with a vector with numbers from the reconstruction. *Polyfit* is used on 3 neighboring points, and the values are stored in a new vector *slope_vec*. This vector is then sorted by descend, and the two largest components are used to obtain the value max_{slope}. In this example $max_{slope} = 0.4 + 0.35 = 0.75$.

figure 5.4. As we can see the sharper the image, the larger the value of max_{slope}. So it seems that we have obtained a value which can indicate if a reconstruction is sharp or not.

To test that the value max_{slope} does not vary too much for the same problem, we consider figure 5.5. As we can see, 100 runs have been made with the method CGLS, and at every run the optimal reconstruction has been stored. From this reconstruction the slope has been calculated by the above method and all of the 100 values have been plotted. The noise level was equal to 0.05. It seems that almost all of the values lie inside the span of $0.82 - 0.86$ which is good, since it indicates that the value is not found in a total random way, and also that the value does not vary too much.

5.2 3D Sharpening

The case for the 3D sharpening is similar to the 2D case. For illustration one can see figure 5.6. The idea is to find a plane on which we can measure the sharpness. Therefore the best would be to take a 3D figure which has a plane parallel with the xy-plane, the yz-plane or the xz-plane, and for this example we have chosen a tetrahedron which has a plane parallel with the xy-plane. Now

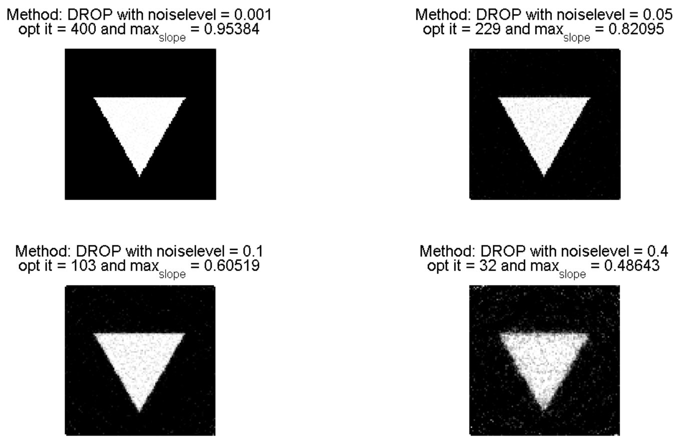

Method: DROP with noiselevel = 0.001
opt it = 400 and max$_{slope}$ = 0.95384
Method: DROP with noiselevel = 0.05
opt it = 229 and max$_{slope}$ = 0.82095
Method: DROP with noiselevel = 0.1
opt it = 103 and max$_{slope}$ = 0.60519
Method: DROP with noiselevel = 0.4
opt it = 32 and max$_{slope}$ = 0.48643

Figure 5.4: Illustrates how the max_{slope} value varies. We see from the figures that the more noise we add to our problem, the smaller the value of max_{slope} we obtain. Standard values have been used.

we know that we have a plane parallel with the xy-axis for which we wish to measure the sharpness off. From figure 5.6 we also know that we have a lot of planes parallel to the xy-plane, but the plane for which we wish to measure the sharpness off, will be the plane consisting of most voxels for this test problem. This means, that if we take the sum of every xy-plane, then the index to the largest number would indicate that our "side" of the tetrahedron has been located. This "side" corresponds to a plane. The principle now follows the principle from 2D, we let a half vector go through the found plane, and do the whole from figure 5.3 again. To ensure that it is not found completely at random, we take the neighboring 8 vectors and do a mean of these 9 values.

As a note, we mention that this way of determining a side for a figure in 3D is not appropriate for all figures. If we consider the case for $n = 20$, a side in this figure is not necessarily the plane which contains the most voxel values. A plane taken from the middle of this figure would contain a larger sum of the voxels, but we know that this does not correspond to a side in the figure.

Unfortunately there are problems with this kind of measurement method for the 3D case. It seems that it does not matter how much noise we add, the sharpness of the edges seems to be very high anyway. For illustration see figures 5.7 and 5.8. On these figures it is clearly seen that different noise levels has been added, since one of the figures is shown with a voxel value of 0.91 whereas

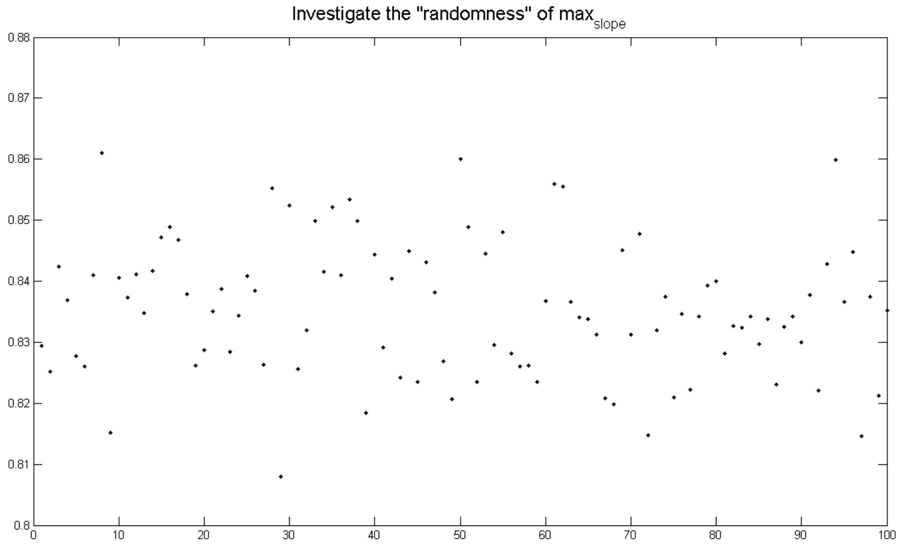


Figure 5.5: Plot of the max$_{slope}$ given different reconstructions made with the same variables. In this case it is standard variables. *noiselevel* = 0.05. As we can see the value is not found completely at random since the variation in the values is small.

the other has a voxel value of 0.70. But even though they are distinguished by *noiselevel*, they seem to have the same value of max_{slope}. For an illustration of *noiselevel* one can observe figure 5.9 and 5.7. These figures have different *noiselevel* but the same voxel value is shown. We can see that the figure with a high noise level added is the figure most "filled with holes".

For an example of the sharpness between two xy-planes see figure 5.10. On this figure we see the edge we have located within our figure, and the xy-plane that follows the edge. It is between these two planes that we wish to establish a measurement of the sharpness in the reconstruction, and since one of the planes are close to 0, whereas the other consists of high voxel values, we ought to obtain a high value of max_{slope}.

So it seems that this way of measuring sharpness in 3D can be problematic. In the example before we only looked through the xy-plane, and it could occur that the error is added in the xz-plane or the yz-plane instead of the xy-plane which we measured through before. Therefore we see figure 5.11. As we can see it does not matter which plane we measure our sharpness in, the value we

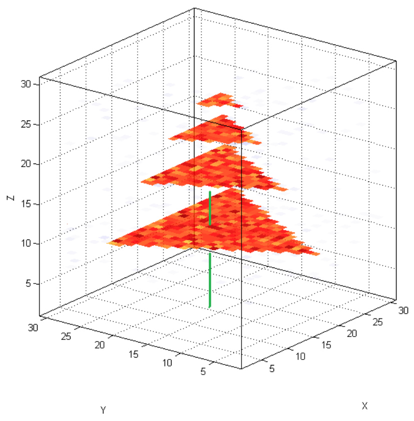


Figure 5.6: Illustrates the edge we are trying to find the slope of. As we can see when we move downwards in the figure, more and more pixels are located within the xy-plane until the actual edge of the figure is found. When this plane is found, a vector, the green line, is placed through this plane to measure the sharpness, just like we did for the 2D case.

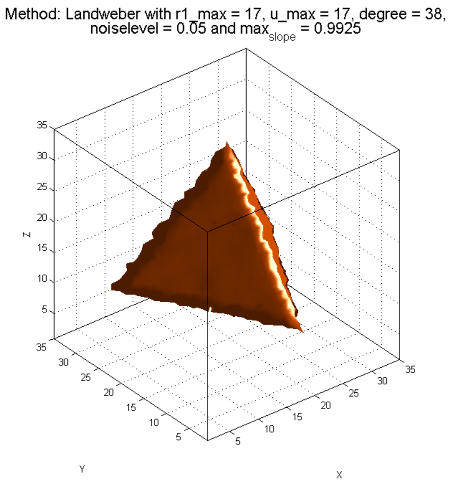

Method: Landweber with r1_max = 17, u_max = 17, degree = 38, noiselevel = 0.05 and max$_{slope}$ = 0.9925
Z
Y
X

Figure 5.7: Shows a reconstruction of a tetrahedron in 3D. The value inside the figure is around 0.91, and this is the breaking point before the figure becomes too ”full of holes”.

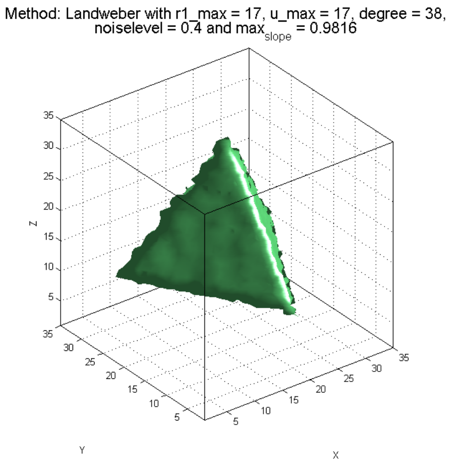

Method: Landweber with r1_max = 17, u_max = 17, degree = 38,
noiselevel = 0.4 and max$_{slope}$ = 0.9816
Z
Y
X

Figure 5.8: Shows a reconstruction of a tetrahedron in 3D. The value inside the figure is around 0.70, and this is the breaking point before the figure becomes too ”full of holes”.

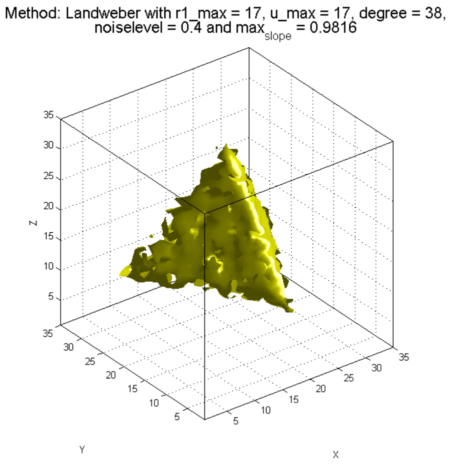

Method: Landweber with r1_max = 17, u_max = 17, degree = 38,
noiselevel = 0.4 and max$_{slope}$ = 0.9816
Z
Y
X

Figure 5.9: Shows a reconstruction of a tetrahedron in 3D. The value inside the figure is around 0.91. For comparison with figure 5.7, we see that *noiselevel* does have an impact on the reconstruction, since this reconstruction which has a high noise level added, is much more ”filled with holes”.

obtain of max_{slope} would still be large. Therefore it seems that when solving 3D problems, the iterative methods might be able to preserve the sharpness of our figures regardless of the noise level.

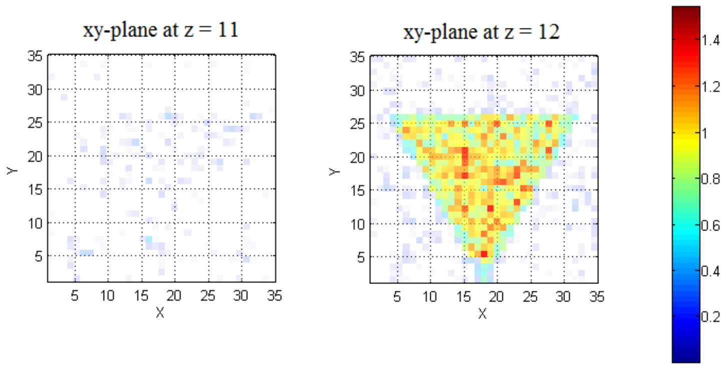


Figure 5.10: Shows the "sharpening" between two edges in 3D. The picture to the left is a plot of the xy-plane where z = 11, and as we can see it is close to 0. The figure to the right indicates the xy-plane at z = 12, where our edge is located. Both figures are a slice of a reconstruction where $noiselevel = 0.4$.

For this type of measurement we finally must know how the value max_{slope} varies as we vary the noise added to our test problem. For this purpose see figure 5.12. From this figure, it does not matter how much noise we add to the 3D test problem, we still obtain a large value of max_{slope}, which was not the case for 2D. This is apparently the way this type of measurement behaves for the 3D problems.

Now we have a measure for the quality of sharpness within a reconstruction, and in the following sections we will look upon other quality measurements. We will also test some constraints that can be applied to our iterative methods to investigate if they will improve the reconstruction.

5.2.1 Non-Negativity, Yes or No?

All the numerical methods in $AIR - Tools$ have been implemented in such a way that an option called non-negativity can be chosen. The question is if we should make use of it or not. In order to illustrate the function of the option, see figure 5.13. This figure illustrates the constraint for a SIRT method. The theory is from [6].

As we can see from these figures there are certain features that appeal the

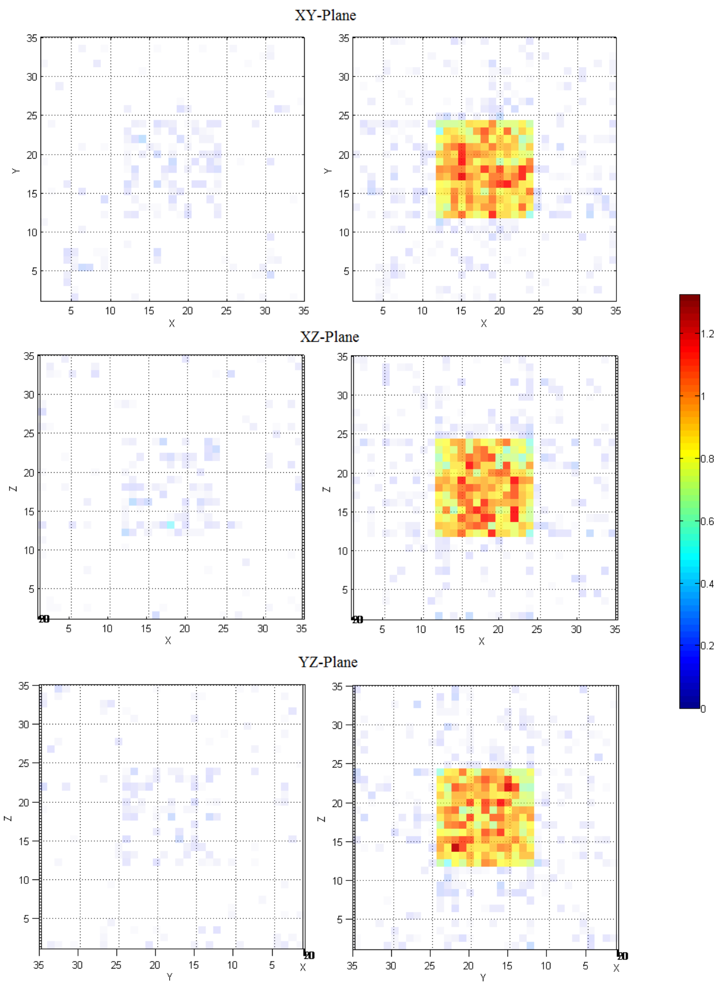


Figure 5.11: Shows the same as figure 5.10. Here we have plotted a cube to check if the error is created in a special way. The 3 figures illustrates the 3 planes we are measuring the sharpness off, the xy-plane, the xz-plane and the yz-plane. As we can see the figures to the left are close to 0. This indicates that a high value of max_{slope} would be obtained, since the figures to the right has large voxel values. The noise level in this reconstruction is $noiselevel = 0.4$.

eye. First of all, when the non-negativity criterion has not been used, the picture seems much more blurry, and the optimal iteration number is found rather fast. Whereas if we use the non-negativity criterion, we obtain a much sharper picture, and visually a much better reconstruction. It may be difficult to see,

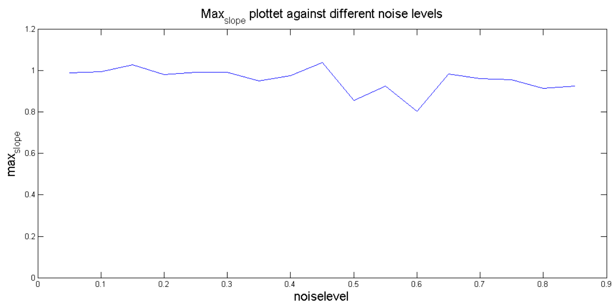


Figure 5.12: Shows the value max_{slope} plotted against different values of *noiselevel*. It is seen that a change in *noiselevel* does not reflect on a lower value of max_{slope} as we saw for the 2D case.

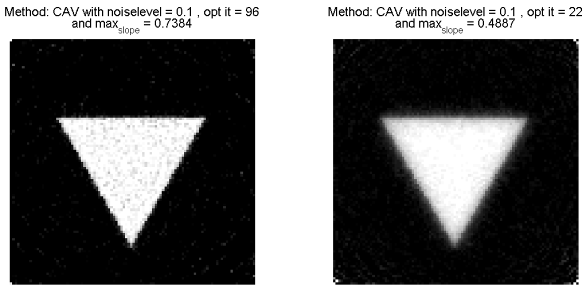

Method: CAV with noiselevel = 0.1 , opt it = 96 and max$_{slope}$ = 0.7384
Method: CAV with noiselevel = 0.1 , opt it = 22 and max$_{slope}$ = 0.4887

Figure 5.13: Plot of a solution made with the method CAV, with and without non-negativity. The figure to the left is with non-negativity and the figure to the right is without. As we see the figure with this constraint on is sharper, and it can be difficult to see, but the noise in the background is less.

but the fact is that there is a lot of "scrambled-noise" in the reconstructions when this constraint is not used. Therefore if we do impose this constraint, some of the noise will be removed resulting in a better reconstruction.

We now take a look on figure 5.14 which illustrates the constraint for the CGLS method. We notice that if we use the non-negative constraint the reconstruction is not that much sharper compared to if we did not impose the constraint. The figure to the left is with non-negativity and the figure to the right is without. Since the value of max_{slope} is slightly larger we will impose this constraint for better comparison with the other iterative methods.

If we see the last figure 5.15, reconstructions for the method Kaczmarz has been made. Again the picture to the left is with the constraint and the picture to the right is without. We notice that the reconstruction does not appear sharper with the constraint on, but on the other hand we lose much of the artefact generated in the reconstruction without the non-negativity constraint. Therefore for all the numerical methods we will impose this constraint when a reconstruction is made.

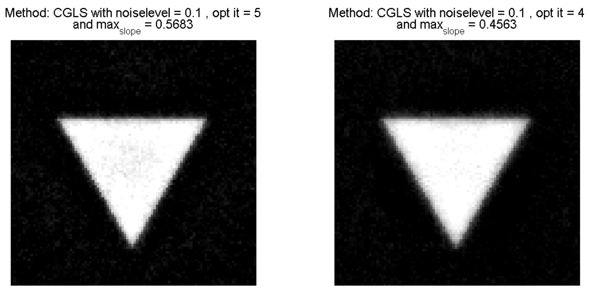

Method: CGLS with noiselevel = 0.1 , opt it = 5
and max$_{slope}$ = 0.5683
Method: CGLS with noiselevel = 0.1 , opt it = 4
and max$_{slope}$ = 0.4563

Figure 5.14: Plot of a solution made with the method CGLS, with and without non-negativity. The figure to the left is with non-negativity and the figure to the right is without. There is not a big difference between these two reconstructions, but the one with the non-negativity constraint on has a larger max_{slope} value.

How did this "scrambled-noise" appear? In the following the theory will be based upon Cimmino's method, (3.4). This is because the reconstruction was obtained with Cimmino's method, but the theory applies to all the numerical methods. Put in words, the non-negativity constraint ensures that our data will not be negative.

In a mathematical way the constraint is introduced by a projection of the solution. So given Cimmino's method

$$x^{k+1} = x^k + \lambda_k \frac{1}{m} \sum_{i=1}^{m} w_i \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|_2^2} a^i, \quad k = 0, 1, 2, \ldots, \ldots \tag{5.2}$$

Then by adding the constraint we obtain that

$$x^{k+1} = P\left[x^k + \lambda_k \frac{1}{m} \sum_{i=1}^{m} w_i \frac{b_i - \langle a^i, x^k \rangle}{\|a^i\|_2^2} a^i \right], \quad k = 0, 1, 2, \ldots, \ldots \tag{5.3}$$

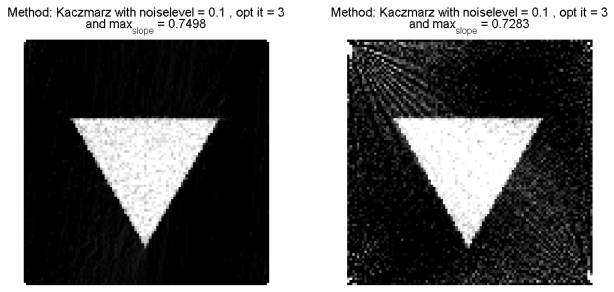


Figure 5.15: Plot of a solution made with the method Kaczmarz, with and without non-negativity. The figure to the left is with non-negativity and the figure to the right is without. As we see the figure with this constraint on is sharper, and obtain less artefact than the reconstruction without this constraint on.

where it holds for P that

$$z_i = \begin{cases} P(y_i), & y_i \geq 0 \\ 0, & y_i < 0 \end{cases}$$

In other words, we ensure that all the negative numbers in our reconstructions are set to zero. How the "scrambled-noise" will influence our reconstructions mathematically is best seen on figure 5.16. We can see from the middle figure, that we will get a new minima if the constraint is not used. Since we are scaling our axes in MATLAB by [0 to 1], our new minima will be set to 0, and all the small values slightly above the minima will be set to > 0 due to the scaling. Therefore a lot of the values will look like "noise" in the reconstructions, but it is seen that the application of this constraint, will remove all this "scrambled-noise". Therefore throughout the rest of the report, we will always use the non-negativity constraint to ensure the best possible reconstruction.

For the 3D case the conclusion still holds, and for a illustration we refer to figure 5.17. As one can see we obtain much better reconstructions with the constraint on, which corresponds to the 2D case. So for the 3D examples one has to use this constraint as well.

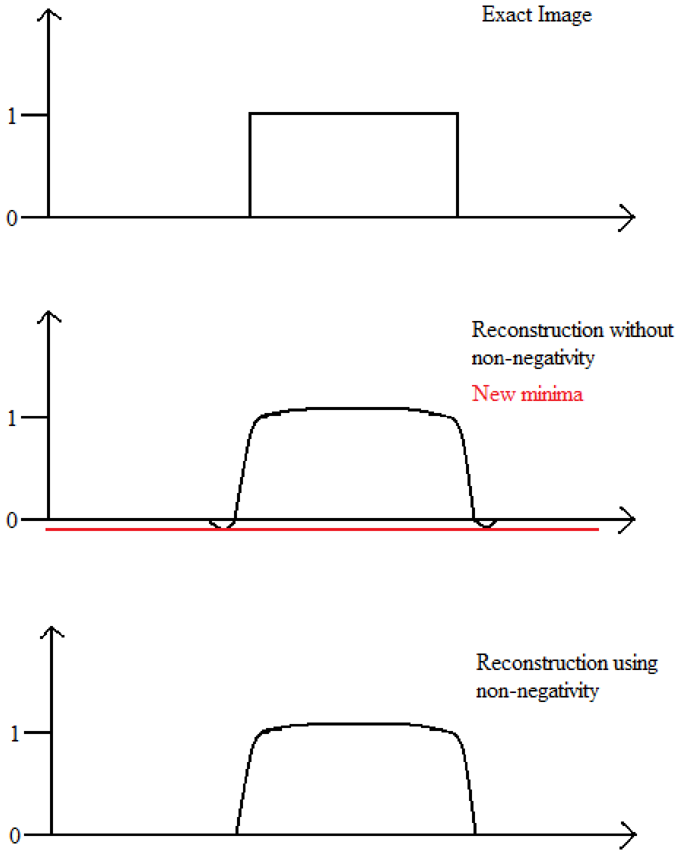

Exact Image
1
0
Reconstruction without non-negativity
New minima
1
0
Reconstruction using non-negativity
1
0

Figure 5.16: An example of why we should use the non-negativity constraint. The top figure illustrates a plot of an exact grain. The middle figure illustrates the reconstruction of this grain without non-negativity. As we notice a new minimum is obtained and every value slightly above this will resemble noise in the reconstruction. The bottom figure illustrates a reconstruction made with non-negativity. As we can see all the small values are set to 0, indicating that the "scrambled noise" in the background has been removed.

5.3 1-Norm vs 2-Norm

Another question is how do we measure the quality of the reconstructions? There are different ways to do this when one works with images. As an example SSIM (Structural Similarity Index Mapping) could be used, but throughout this

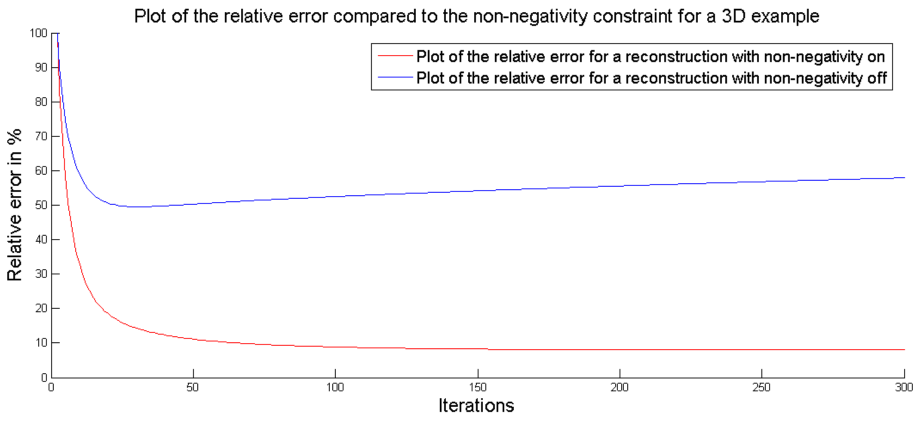


Figure 5.17: An example of two reconstructions made in 3D. The difference in the two graphs is the non-negativity constraint. The method used here is Landweber with $noiselevel = 0.05$. As we can see, the relative error obtained is much lower for the reconstruction with the non-negativity constraint. Therefore we will make use of this constraint.

report we will mostly use norms. If the reader wishes to learn more about the SSIM, we refer to [8].

When the test problem is known, it is rather easy to compare our reconstruction to the original data and look at how well the reconstruction resembles the original. To do this we can apply different norms. In the real world though, we do not obtain the original data, and we seek a reconstruction as good as possible. There are ways to measure if a reconstruction is decent or not. One of the methods is to use stopping-criteria to the iterative methods, but more on this later in the report. For now, when the original data is known, we will first decide which norm to make use of to determine if the reconstruction is appropriate or not.

The question is if we should use the 1-norm or the 2-norm. There exist other norms, e.g. $\infty - norm$, but for our cases we will only look at these two norms. They are defined in the following way in [4] on page 908

$$\begin{aligned} \|x\|_1 &= |x_1| + \ldots + |x_n| \\ \|x\|_2 &= \sqrt{x_1^2 + \ldots + x_n^2}. \end{aligned}$$

The idea is now that by using these norms we may estimate how far we are from the exact solution by

$$\begin{aligned} type_1 &: \quad \|x^k - x^{exact}\|_1 \\ type_2 &: \quad \|x^k - x^{exact}\|_2. \end{aligned}$$

The closer these estimates are to 0, the better, and recall that this is exactly what the semi-convergence plots were showing. Before we investigate how many iterations the two different types of norms need, lets observe if there is any visual difference between these 2 norms. To do this we observe figure 5.18, figure 5.19 and figure 5.20.

From figure, 5.18 a SIRT method has been shown. We see that it does

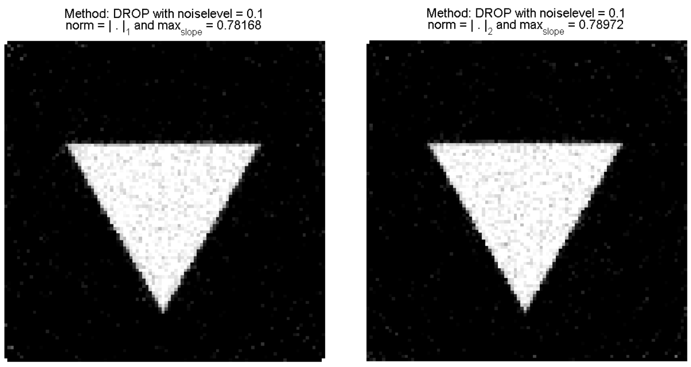

Method: DROP with noiselevel = 0.1
norm = | . |1 and maxslope = 0.78168
Method: DROP with noiselevel = 0.1
norm = | . |2 and maxslope = 0.78972

Figure 5.18: Example of 1 and 2-norm for a SIRT method. The method used here is DROP, and the reconstruction to the left is created with the 1-norm and the reconstruction to the right is made with the 2-norm. As we can see there is no visual difference between the two reconstructions.

not matter which norm we use. The visual difference is more or less the same, and the reconstructions have the same sharpness of edges. On figure, 5.19 the CGLS method has been shown. The visual difference seems the same in the reconstructions. The sharpness of the reconstructions varies, so it might be appropriate to use the 2-norm for this type of method. Still we will make use of the 1-norm for the CGLS method. The last figure, 5.20 is an illustration of the Kaczmarz method. It shows the same trend as the figure from the SIRT method. There is no visual difference between the two reconstructions.

Since the norms resembles each other visually the next step is to see how many iterations the different norms need. Note from Chapter 3 that if a method uses less iterations compared to another method, it does not necessarily mean that

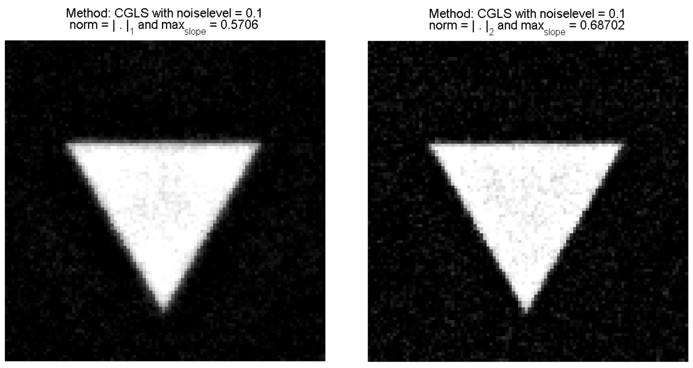

Method: CGLS with noiselevel = 0.1
norm = | . |$_1$ and max$_{slope}$ = 0.5706
Method: CGLS with noiselevel = 0.1
norm = | . |$_2$ and max$_{slope}$ = 0.68702

Figure 5.19: Example of 1 and 2-norm for the CGLS method. The reconstruction to the left is created with the 1-norm and the reconstruction to the right is made with the 2-norm. As we can see the reconstruction to the right has a larger max_{slope} value.

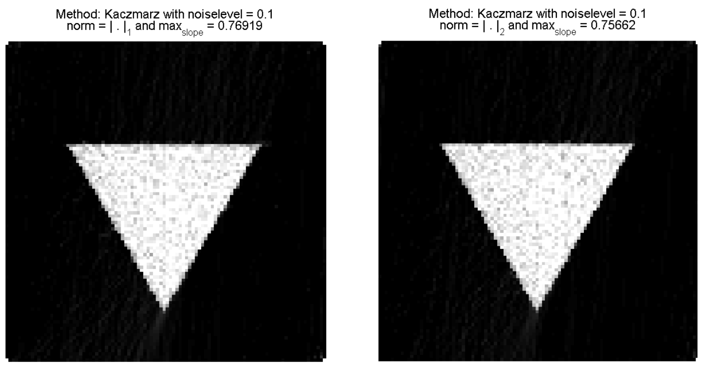

Method: Kaczmarz with noiselevel = 0.1
norm = | . |$_1$ and max$_{slope}$ = 0.76919
Method: Kaczmarz with noiselevel = 0.1
norm = | . |$_2$ and max$_{slope}$ = 0.75662

Figure 5.20: Example of 1 and 2-norm for Kaczmarz's method. The reconstruction to the left is created with the 1-norm and the reconstruction to the right is made with the 2-norm. As we can see there is no visual difference between the two reconstructions.

the method is better.

We now observe table 5.1. In this table *wo* refers to "without", *nn* refers to the non-negativity constraint and *w* means "with". As we can see from this table the number of iterations depend mainly on whether we use the non-negativity constraint. If the constraint is not on, the methods use few iterations before the best reconstruction is found. As we saw earlier this type of reconstruction was not appropriate. If we apply the constraint, more iterations are performed, implying an addition in computation time. But this will result in a better reconstruction. Concerning the number of iterations, we see that the 1-norm uses fewer iterations than the 2-norm, and since the quality of the reconstructions was the same, we will use the 1-norm from this point on. The reason that all iterations are equal to 800 in the first row is because we have chosen to stop the iterations at $k_{max} = 800$. When the noise is almost equal to 0, all of the methods will use many iterations to converge towards the optimal solution. We find the exact number of iterations to be unimportant, and instead of wasting computation time, we choose to stop at a given number, in this case 800, and inform the reader that more iterations would be needed.

Table 5.1: Shows the number of iterations used with the method DROP for different noise levels. $k_{max} = 800$.

Noise level	1-norm wo/nn	2-norm wo/nn	1-norm w/nn	2-norm w/nn
0.001	800	800	800	800
0.05	41	56	242	244
0.1	22	37	99	115
0.4	8	3	31	30

We now turn our attention towards table 5.2 which shows the CGLS method. The remarkable feature here is that the method seems to be very sensitive towards noise. When only very little noise is added, the method uses a lot of iterations before finding the optimal solution. But when some noise is added the method uses very few iterations to find the optimal solution. This applies for both norms and with or without the non-negativity constraint.

Table 5.2: Shows the number of iterations used with the method CGLS for different noise levels. $k_{max} = 800$.

Noise level	1-norm wo/nn	2-norm wo/nn	1-norm w/nn	2-norm w/nn
0.001	800	414	776	206
0.05	6	7	6	8
0.1	4	5	5	6
0.4	3	3	3	3

The last table we need to turn our attention to is for Kaczmarz's method, and for this we observe table 5.3. We see that the same trend in the data appears as it did for the CGLS method. Kaczmarz seems to be very sensitive to noise, but it is only weakly dependent on the noise level. If very little noise is added, Kaczmarz uses a lot of iterations before converging towards an optimal solution.

Table 5.3: Shows the number of iterations used with the method Kaczmarz for different noise levels. $k_{max} = 800$.

Noise level	1-norm wo/nn	2-norm wo/nn	1-norm w/nn	2-norm w/nn
0.001	237	271	175	400
0.05	5	5	5	19
0.1	3	3	3	4
0.4	1	2	1	2

The last feature we need to ensure is that the result of the norms follows each other. In other words, that they more or less return the same relative error value. For illustration we observe figure 5.21. As expected the relative errors are rather independent of the norms so the choice of norm hardly interferes with our solution. Therefore we can choose to use the 1-norm only.

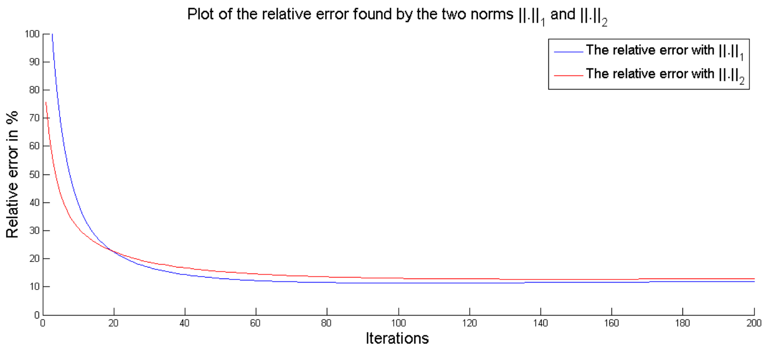


Figure 5.21: Plot of the two norms compared against each other. The two curves follow each other, but the 1-norm tends to be lower than the 2-norm generally.

By now we have seen how different methods work with the different norms as a measurement of the quality of the reconstructions. Finally, we will investigate the "value" that a method obtains when the optimal solution is found. This "value" corresponds to $\|x^k - x^{exact}\|_1$, but the relative error will also be given. The idea is that CGLS and Kaczmarz uses very few iterations compared to the SIRT methods, but if the SIRT methods find a much better solution to our problem, it would not be appropriate to use Kaczmarz or CGLS. Therefore see figures 5.22, 5.23 and 5.24. The first figure shows the first 20 iterations. In this interval CGLS and Kaczmarz find their optimal solution, while the 5 other methods still converge towards their optimal solution. Figure 5.23 shows the methods on a larger scale, namely 600 iterations where every method has found its optimal solution. The y-axis is cut at 1000 since CGLS diverges after the solution is found. Figure 5.24, shows a zoom of the large scale, which should indicate how our "values" lie relative to each other.

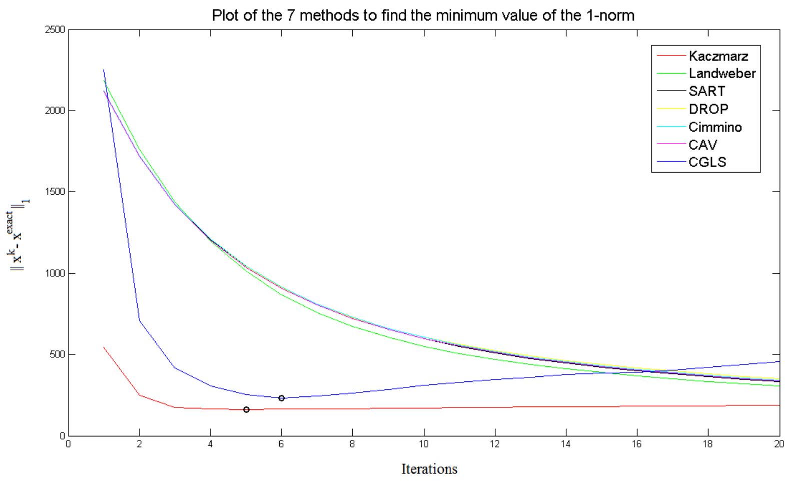


Figure 5.22: Plot of all 7 methods compared with each other. We have decided to make the first plot from iteration 1 to 30 only. The reason for this is that CGLS and Kaczmarz locate their minimum iteration value within this interval. The plot is made for a 2D example.

To see the exact values we look at table 5.4. As we notice from this table, the value obtained from the SIRT methods are in fact lower, so if one wishes the best possible reconstruction, and computation time is not an issue, a SIRT method would seem appropriate.

For the 3D case the same results apply. Instead repeating everything, we only show the last table with its corresponding figures. First let us have a look at 5.25, 5.26 and 5.27. The remarkable feature is that CGLS still seems to perform worse compared to the other methods. And furthermore it seems that Kaczmarz is actually able to perform as well as the SIRT methods for 3D. For the values we refer to table 5.5. From this table it follows that CGLS performs poorly compared to the SIRT methods, and that Kaczmarz performs just as well as the SIRT methods.

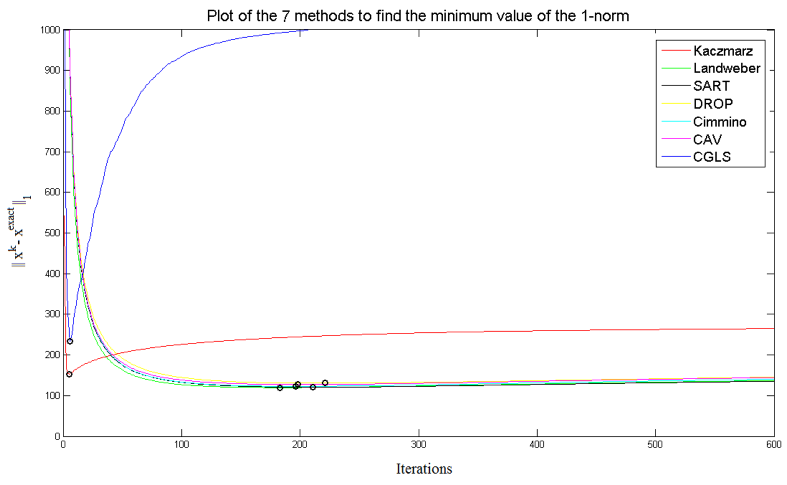


Figure 5.23: Plot of the 7 different methods compared with each other. The idea is to give an illustration of how good the methods are against each other. The y-axis has been "cut" since CGLS gave large values. The plot is made for a 2D example.

So now we know which norm we should use when the difference between the reconstructions and the original data are quantified, and also that we should apply the non-negativity constraint. One of the last things we need to check before we can obtain a best possible reconstruction, is whether a guess of the starting vector will affect the solution.

5.3.1 The Starting Vector

As we saw in Chapter 3, the usual value for a starting vector for our iterative methods is the 0-vector. If we observe table 5.6, an overview for this method with different starting vectors has been made. The top line indicates which starting vector that has been used. The clear trend is that the 0-vector, the 1-vector and the $\frac{1}{2}$-vector deliver very similar results. But if we create a starting vector consisting of randomly generated numbers between 0 and 1, which is done by the function *rand* in MATLAB, we actually obtain a quite different result. As we can see from the table a lot more iterations are being used when this starting vector is applied, meaning that more computation time will be used. But the reconstructions we obtain are quite different, measured in the max_{slope} context.

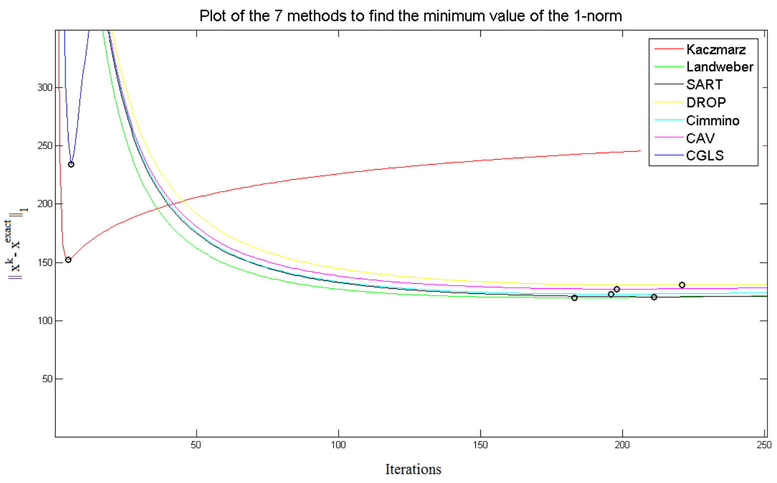


Figure 5.24: This figure is the same as 5.23 with the exception that we have zoomed in on the important things. Visually it should give a better indication of how good the methods are compared to each other. The plot is made for a 2D example.

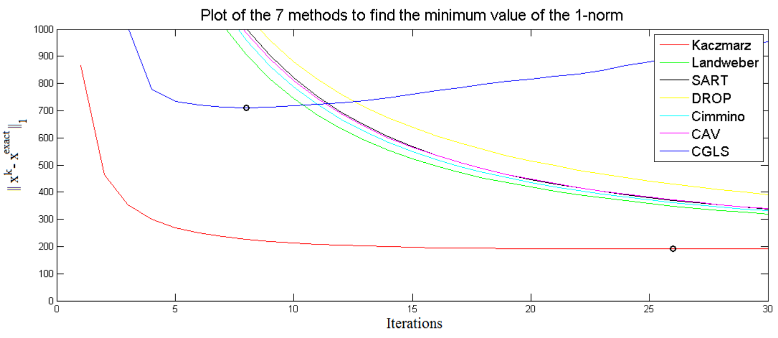


Figure 5.25: Plot of all 7 methods compared with each other. We have decided to make the first plot from iteration 1 to 30 only. The reason for this is that CGLS and Kaczmarz locate their minimum iteration value within this interval. The plot is made for a 3D example.

We consider figure 5.28 and 5.29

As we can see from these figures, if we use the starting vector *rand*, our reconstruction seems to have sharper contours than the reconstruction done with 0, but it might just be an optical illusion. The result is the same if we use the two

Table 5.4: The minimum value obtained by the 1-norm for the 7 different methods in 2D. Standard variables has been chosen with *noiselevel* = 0.05

Method used	$\|x^k - x^{exact}\|_1$	k^{opt}	Relative Error
CGLS	232	5	14.56%
Kaczmarz	155	6	9.73%
Landweber	121	181	7.60%
SART	119	204	7.47%
CAV	122	210	7.66%
DROP	129	219	8.10%
Cimmino	126	204	7.91%

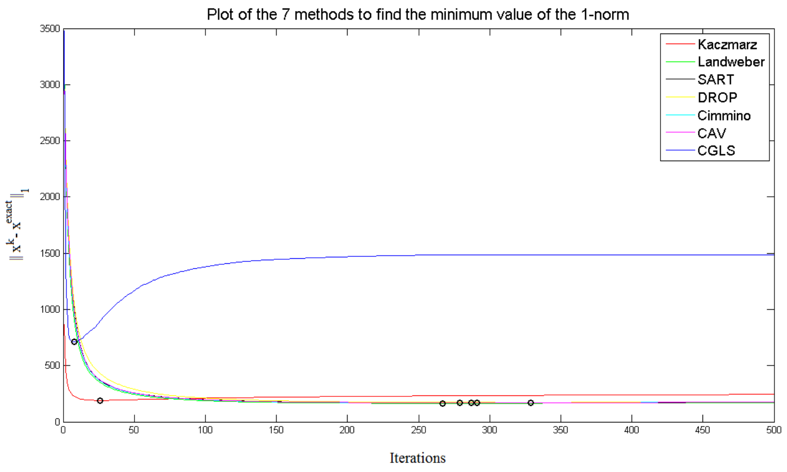


Figure 5.26: Plot of the 7 different methods compared with each other. The idea is to give an illustration of how good the methods are against each other. The plot is made for a 3D example.

other starting vectors. The slope number, max_{slope}, appears to be larger for the *rand* reconstructions. So for all the SIRT iterative methods, there might be a chance to obtain sharper reconstructions (w.r.t. value) with starting vector *rand*, but the number of iterations increase. Also by calculating the relative error at noise level 0.1 for Landweber's method we get

$$\begin{aligned} x_{0-vector}^{k^{opt}} &\Rightarrow rel_{error} = 12.19\% \\ x_{rand-vector}^{k^{opt}} &\Rightarrow rel_{error} = 16.90\%. \end{aligned}$$

We get a change of almost 5% in the relative error if we choose the *rand* as starting vector. So even if we have a chance to obtain a slightly sharper image,

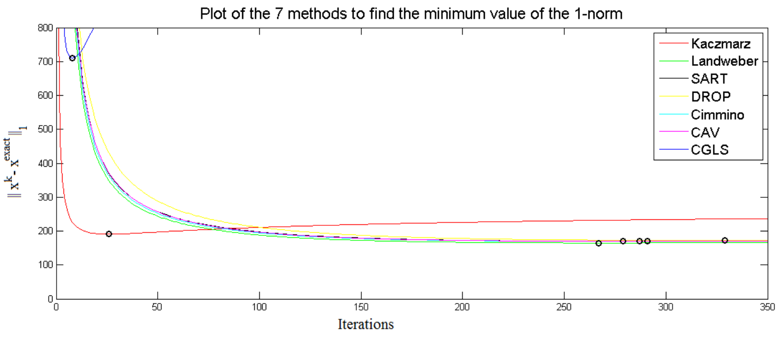


Figure 5.27: This figure is the same as 5.26 with the exception that we have zoomed in on the important things. Visually it should give a better indication of how good the methods are compared to each other. The plot is made for a 2D example.

Table 5.5: Shows the minimum value obtained by the 1-norm for the 7 different methods in 3D. Standard values has been chosen with *noiselevel* = 0.05

Method used	$\|x^k - x^{exact}\|_1$	k^{opt}	Relative Error
CGLS	710	8	31.74%
Kaczmarz	191	26	8.54%
Landweber	165	267	7.38%
SART	170	291	7.60%
CAV	170	287	7.60%
DROP	172	329	7.69%
Cimmino	170	279	7.60%

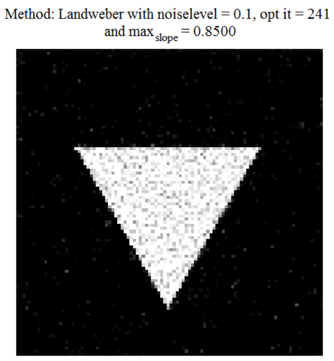
it will not be worth the cost of an increase in the relative error.

The CGLS method however works differently, if we choose a different starting vector than 0, our reconstructions will be of poor quality. See figure 5.30. If the other two starting vectors, 1 and $\frac{1}{2}$ are used in the CGLS method, we end up with the same reconstruction as for *rand*, which are terrible.

In order to see if the Kaczmarz's method is affected in the same way we observe table 5.7. It seems that if we apply the other starting vector, *rand*, the method uses more iterations, but only for small noise levels. When the noise level increases, the method uses the same amount of iterations. To see if a different choice of starting vector results in a visual change, we observe figures 5.31 and 5.32. These pictures illustrate the reconstruction done with noise level equal to 0.1 and 0.4. 5.31 shows the reconstructions done with *rand*-vector as starting

Table 5.6: Method: Landweber, w/nn and 1-norm. Shows how many iterations are needed for different noise levels to compute our solution. $k_{max} = 400$.

Noise level	$(0,\dots,0)^T$	$(1,\dots,1)^T$	$(\frac{1}{2},\dots,\frac{1}{2})^T$	$rand(n,1)$
0.001	400	400	400	400
0.05	173	192	190	400
0.1	79	81	86	271
0.4	28	28	28	43


Method: Landweber with noiselevel = 0.1, opt it = 241
and max$_{slope}$ = 0.8500

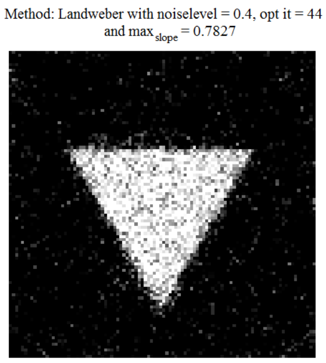

Method: Landweber with noiselevel = 0.4, opt it = 44
and max$_{slope}$ = 0.7827

Figure 5.28: Illustration of two reconstructions made with *rand* as starting vector. It seems that the value of max_{slope} is increased by using this starting vector if we compare to figure 5.29. Also it seems that more noise is present within the grain.

vector whereas in 5.32 the 0-vector is used. At $noiselevel = 0.1$ there is no visual difference and the value max_{slope} is almost equal. On the other pictures where the noise level is 0.4 only the value of max_{slope} differs. Since there is no change in the reconstructions, we will stick to the 0 vector since this worked out to be the best starting guess for the SIRT methods and the CGLS method.

To see if the same applies for the 3D case we observe table 5.8. As we can see by applying the 0 vector we always get a lower value of the relative error. So we obtain the same results in 3D as we do in 2D when considering starting vectors. Therefore we will use the 0 vector from now on for all methods.

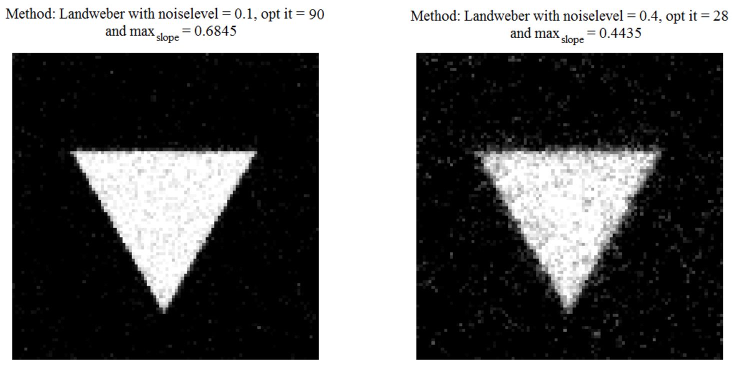


Figure 5.29: Illustration of two reconstructions made with 0 as starting vector. It seems that the value of max_{slope} is decreased by using this starting vector compared to figure 5.28. Also it seems that more noise is present within the grain.

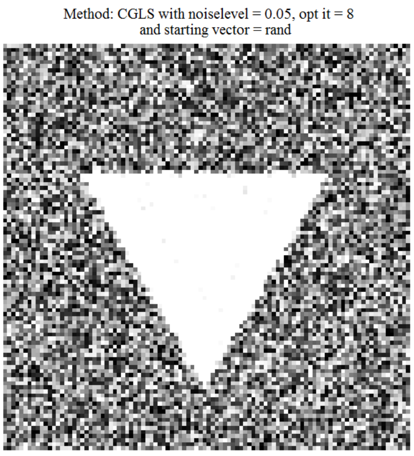


Figure 5.30: Shows that if we choose a different starting vector than 0 for the CGLS algorithm we end up with a poor reconstruction.

Until now we have examined different methods for measuring how good our

Table 5.7: Method: Kaczmarz, w/nn and 1-norm. Shows how many iterations are needed for different noise levels to compute our solution. $k_{max} = 400$.

Noise level	$(0,\ldots,0)^T$	$(1,\ldots,1)^T$	$(\frac{1}{2},\ldots,\frac{1}{2})^T$	$rand(n,1)$
0.001	237	271	175	400
0.05	5	5	5	19
0.1	3	3	3	4
0.4	1	2	1	2

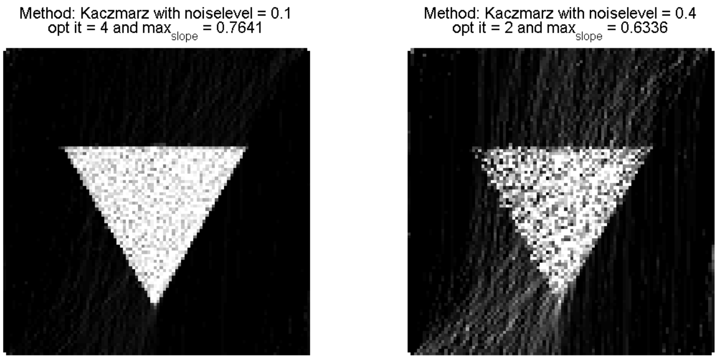
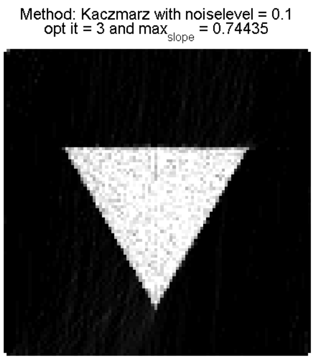

Method: Kaczmarz with noiselevel = 0.1
opt it = 4 and max$_{slope}$ = 0.7641
Method: Kaczmarz with noiselevel = 0.4
opt it = 2 and max$_{slope}$ = 0.6336

Figure 5.31: Shows two reconstructions made for Kaczmarz's method. Here we have chosen *rand* as starting vector.

reconstruction is compared to the original image. In the next section we will introduce another measurement method.

5.3.2 Miss Classified Pixels

So far we have looked at how visually and mathematically good a reconstruction is (1-norm) and how sharp we can reconstruct an edge (max_{slope}). One feature we will look further into, is to observe how many of our reconstructed pixels are miss-classified. The reason for this is that we wish to exploit the fact that our original image is binary, meaning it only consists of 0's and 1's. When we have obtained our reconstruction, given some parameter, τ_2, we set everything above this parameter equal to 1 and everything below it equal to 0. Hereby we can



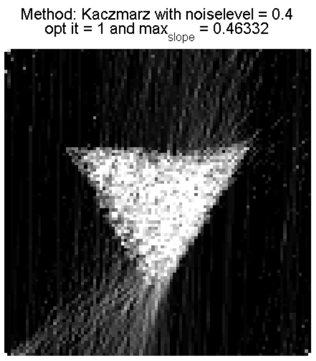


Figure 5.32: Shows two reconstructions made for Kaczmarz's method. Here we have chosen 0 as starting vector.

Table 5.8: Shows the relative error found with different guesses of the starting vector. The methods used are Kaczmarz,Landweber and CGLS. The test problem is made in 3D with standard variables.

Noise level	$(0,\dots,0)^T$	$(1,\dots,1)^T$
Kaczmarz	14.49	17.67
Landweber	11.09	14.83
CGLS	34.47	$N\backslash A$

count how many pixels, relative to the original image, that are wrongly set to 0 and 1. Hereby we end up with 3 different measures given as:

$$\begin{aligned}
A &: \quad \text{Relative Error } = \frac{\|x^{rec} - x^{exact}\|_1}{\|x^{exact}\|_1} \\
B &: \quad \text{Number of miss-classified pixels } = \frac{\|x^{rec,bin} - x^{exact}\|_1}{N^2} \cdot 100 \\
C &: \quad \text{Measure of sharpness of an edge } = max_{slope}.
\end{aligned}$$

5.3.3 Analysis of the Grain

Now when we are able to create our grain we need to test if different values of the grain will interfere with a given reconstruction. We know that the 1-norm and the 2-norm resembles each other, the idea with this section is just to illustrate that our methods are able to do reconstructions when we differ the variables we have at hand. These variables are the location of the grain, $(x0\_x, x0\_y)$, the size of the grain, *scalar* and the number of edges of the grain. To do this we look at tables 5.9 and 5.10 From these tables it is seen that if we add more

Table 5.9: Method: SART, noise level = 0.1.
Shows how many iterations are needed to compute the optimal solution given the variables needed to create the grain.

n	4		4	
(x_0, y_0)	$(0.5, 0.5)$		$(0.1, 0.9)$	
scalar	0.1	0.6	0.1	0.6
$iteration_{opt}$	500	103	500	172

Table 5.10: Method: SART, noise level = 0.1.
Shows how many iterations are needed to compute the optimal solution given the variables needed to create the grain.

n	200		200	
(x_0, y_0)	$(0.5, 0.5)$		$(0.1, 0.9)$	
scalar	0.1	0.6	0.1	0.6
$iteration_{opt}$	500	144	500	189

edges, increase in n, the SIRT methods will use some more iterations, but not that much, and the same applies if we move the center point towards one of the corners. On the other hand, scalar plays a role since a large grain takes "few" iterations compared to the number of iterations needed for a small grain. The small grains computation was stopped at k_{max} iterations. Here k_{max} was set to 500. To see if the same applies for different noise levels we look at tables 5.11 and 5.12. As we can see from these tables it is hard to say if the noise level has an impact on these variables, but the trend is the same as before. Also the noise added corresponds to 0.4 which can ruin our data.

So it seems that the only big difference is that if we have a small grain we need more iterations to compute the optimal solution. If we on the other hand

Table 5.11: Method: SART, noise level = 0.4.
Shows how many iterations are needed to compute the optimal solution given the variables needed to create the grain.

n	4		4	
(x_0, y_0)	(0.5, 0.5)		(0.1, 0.9)	
scalar	0.1	0.6	0.1	0.6
$iteration_{opt}$	500	22	500	26

Table 5.12: Method: SART, noise level = 0.4.
Shows how many iterations are needed to compute the optimal solution given the variables needed to create the grain.

n	200		200	
(x_0, y_0)	(0.5, 0.5)		(0.1, 0.9)	
scalar	0.1	0.6	0.1	0.6
$iteration_{opt}$	500	26	500	31

have a large grain the iterations needed to compute the optimal solution would decrease. The next thing is to investigate if the same applies for the CGLS method. We observe table 5.13 and 5.14. As we can see from these two tables, none of the variables seems to have a great impact on the reconstruction, since the optimal iteration number is more or less the same. The same tables with a noise level equal to 0.4 have been made but are omitted here. The data inside the tables were much similar to the above. Kaczmarz method was similar to the results obtained from here, and the same occurred for the 3D implementation of the grain.

Table 5.13: Method: CGLS, noise level = 0.1.
Shows how many iterations are needed to compute the optimal solution given the variables needed to create the grain.

n	4		4	
(x_0, y_0)	(0.5, 0.5)		(0.1, 0.9)	
scalar	0.1	0.6	0.1	0.6
$iteration_{opt}$	5	4	6	5

Until now we have looked upon what tomography is and how it works. Furthermore we are able to create certain grains, which should be of interests since

Table 5.14: Method: CGLS, noise level = 0.1.
Shows how many iterations are needed to compute the optimal solution given the variables needed to create the grain.

n	200		200	
(x_0, y_0)	$(0.5, 0.5)$		$(0.1, 0.9)$	
scalar	0.1	0.6	0.1	0.6
$iteration_{opt}$	6	4	6	5

they resemble real life material. In the next chapter we will look at how good the different iterative methods are at solving our inverse problems, and how different values of the input parameters will affect the found solution.

5.4 Summary

With this chapter, discussions and introductions to new measurement methods and concepts have been made. As one saw it could be difficult to state whether a reconstruction is sharp or not. Unfortunately it seemed that either our method was not good enough for the 3D case, or else the iterative methods made sharp edges when reconstructing a 3D test problem. Furthermore it was shown that it does not matter if we apply the 1-norm or the 2-norm, and therefore the 1-norm was chosen as a standard measure.

The non-negativity constraint was shown to be very useful, and therefore we should make use of this when doing reconstructions. A different choice of starting vector, compared to the 0-vector seemed inappropriate since this resulted in a larger relative error. At the end, different test were made for creation of the grain.

Chapter 6

Determination of the Lowest Number of Rays and Angles

Now we have seen how we can create our test problems, and we have looked at different ways of measuring the reconstructions. When we solve our problem (2.1), we have so far only considered one way of creating the matrix A, since a standard choice of parameters has been chosen. In this chapter we will look into the variables needed to create the matrix, and further investigate these. The reason for this is that A takes much time to compute. We want to decrease the computation time as much as possible, without ruining our reconstructions.

6.1 Setting up the Matrix A

One of the important features of the matrix A is that it is extremely large for a given problem, and therefore consume a lot of computer time. We want to be able to construct this matrix in such a way that the computation time is minimized, without risking a poor reconstruction.

To do this, we need to look into the structure of the matrix A, since this is the term that gives us the biggest problems w.r.t. computer time. From *paralleltomo.m* we know that we have different variables available, which we can adjust. We know from Chapter 4 that A consists of the number of rays that penetrate our domain given each angle θ times the number of pixels we have. From this it is clearly seen that the size of A corresponds to

$$\begin{aligned} row_A &= p \cdot length(\theta), \\ column_A &= N^2, \end{aligned} \tag{6.1}$$

where p is the number of rays being sent in each different direction θ, and N^2 is our total number of pixels (in 2D). The question is now how few rays we can send through our image given a number of angles before it will affect our reconstructions in a negative way. First let us have a look at the different variables that we can adjust.

6.1.1 Influence of w

As first variable we will look upon w. From *paralleltomo* and figure 4.3 we know that w indicates the distance between the first ray and the last ray. Since we want to be able to hit every pixel inside our domain, it is important to choose w appropriately, so that given any angle θ, we will hit the whole domain. From figure 4.1 we know that our domain is discretized into N pixels, which means that the length of our square will correspond to N. Furthermore the largest distance inside a square is known to be the diagonal which is found to be $\sqrt{N^2 + N^2} = \sqrt{2}N$. Therefore the default value of w is also set to this number, and as an example of what will happen if we choose another w, one can look at figure 6.1. From this we need to ensure a proper value of w, since a low value of w results in a blurry image, and the default value seems reasonable.

As a note, one should be careful when setting this value. If an extremely large value of w is chosen, the distance between each ray, $\frac{w}{p-1}$, will become to big, indicating that a lot of rays would not hit our domain. But overall we see that w does not have an impact on how A is being created, and therefore we will choose the default value implemented, $w = \sqrt{2}N$, when we do tests measuring the variables p and θ.

6.2 Influence of p and θ

The idea in this section is to adjust the number of rays we are sending through our object and adjust the number of angles, in such a way that p and θ will be as small as possible without resulting in poor reconstructions measured by our different measurement approaches. We saw that A consists of (6.1),

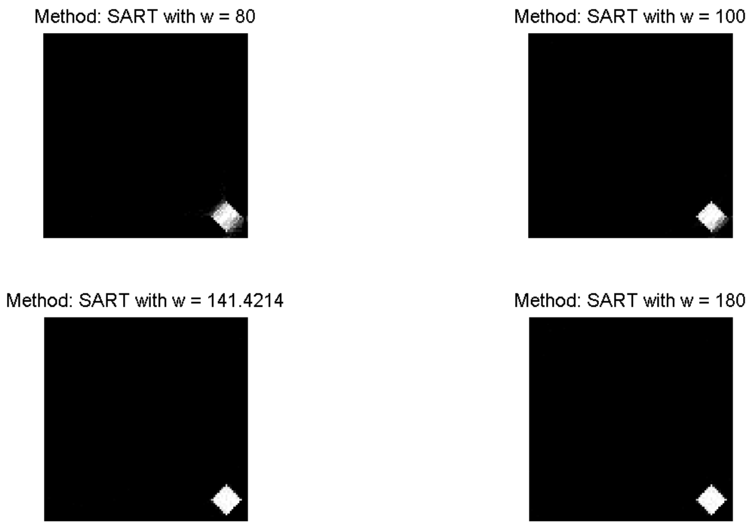


Figure 6.1: Shows how the reconstruction will be affected by different values of w. N is kept constant, p is kept constant, *noiselevel* is kept constant and the length of θ is kept constant. When w becomes too small, artifacts tend to appear.

meaning that when $A \in \mathbb{R}^{m \times n}$, an underdetermined system will be present when $p \times length(\theta) < N^2$ and an overdetermined system is present when $p \times length(\theta) > N^2$. From this we are able to test whether the methods can handle an underdetermined system or not. The different tests made are tabulated in table 6.1, 6.2, 6.3 and 6.4. One should notice that in this case, the total number of ones in the exact image corresponds to 16.21% of the total pixels for this example. Therefore the percent of miss-placed pixels should be viewed upon with respect to this number.

From table 6.1 we notice that almost no noise has been added to the reconstructions. This is reflected on the results, since we find both sharp images and no pixels are miss-placed. The relative error is very small, indicating that we have obtained a good reconstruction. Even for small values of p and θ we can still obtain a decent reconstruction. On figure 6.2 an example of the "corners" in the table has been plotted. With corners we mean the extreme values, that is $p = 0.8N, \theta = 0 : 7.5 : 179$, $p = 0.8N, \theta = 0 : 1 : 179$ and $p = 1.6N, \theta = 0 : 7.5 : 179$. As we see with almost no noise added, even though the table shows good values of the measurements, we see some artefact on the

Table 6.1: Method: Cimmino, with *noiselevel* $= 0.001$ and $N = 64$. Shows the different 3 types of measurement values obtained for different p and θ values. The table to the right is a helping table, indicating what measurement type the different measurement values corresponds to.

$\theta \backslash p$	0.8N	1N	1.2N	1.4N	1.6N
0:7.5:179	**0.27**	**0**	**0**	**0**	**0**
	0.24	*0.20*	*0.09*	*0.08*	*0.06*
	0.81	0.88	0.93	0.93	0.93
0:5:179	**0**	**0**	**0**	**0**	**0**
	0.19	*0.17*	*0.08*	*0.05*	*0.04*
	0.89	0.93	0.96	0.96	0.95
0:3.5:179	**0**	**0**	**0**	**0**	**0**
	0.17	*0.14*	*0.06*	*0.04*	*0.03*
	0.89	0.95	0.96	0.97	0.97
0:2.5:179	**0**	**0**	**0**	**0**	**0**
	0.16	*0.13*	*0.05*	*0.04*	*0.02*
	0.90	0.95	0.97	0.97	0.98
0:1:179	**0**	**0**	**0**	**0**	**0**
	0.15	*0.12*	*0.05*	*0.04*	*0.02*
	0.91	0.96	0.97	0.98	0.98

Relative error, A
Number of miss-classified pixels, B
Measure of sharpness, C

reconstruction. It seems that the artefact is most extreme when we decrease p and furthermore it seems that it does not matter if A is over- or underdetermined.
Table 6.2 shows the same tendency. The result shows that the relative error is low and we have almost none miss-classified pixels. The noticeable feature here is that for small values of θ, the sharpness value tends to decrease. Also it seems that it does not matter if A is over- or underdetermined. For an example of pictures, see figure 6.3. On these figures, we see that artefact tend to happen for the combinations of large and small values of p and θ. These combinations will be referred to as "extreme values" and tested later on in the report.

In table 6.3 we see that the trend changes. The relative error is larger than from the two tables created with *noiselevel* $= 0.001$ and *noiselevel* $= 0.05$, with approximately a factor ~ 2, and the reconstructions have a problem with placing all the reconstructed pixels correctly compared with the exact image. The sharpness value is still low for small values of p and θ, but also the noise level has been increased. As an example of figures for the extreme values in this case we refer to figure 6.4 and again it is noticed that it does not matter if A is over- or underdetermined.

Table 6.2: Method: Cimmino, with *noiselevel* $= 0.05$ and $N = 64$. Shows the different 3 types of measurement values obtained for different p and θ values. The table to the right is a helping table, indicating what measurement type the different measurement values corresponds to.

$\theta \backslash p$	0.8N	1N	1.2N	1.4N	1.6N
0:7.5:179	**0.81** *0.32* <u>0.78</u>	**0.24** *0.28* <u>0.86</u>	**0.02** *0.19* <u>0.89</u>	**0.07** *0.15* <u>0.79</u>	**0** *0.13* <u>0.82</u>
0:5:179	**0.15** *0.30* <u>0.86</u>	**0.15** *0.25* <u>0.89</u>	**0** *0.16* <u>0.89</u>	**0** *0.13* <u>0.85</u>	**0.07** *0.13* <u>0.88</u>
0:3.5:179	**0.12** *0.28* <u>0.86</u>	**0.05** *0.23* <u>0.88</u>	**0** *0.14* <u>0.96</u>	**0.02** *0.12* <u>0.92</u>	**0** *0.12* <u>0.93</u>
0:2.5:179	**0** *0.24* <u>0.87</u>	**0** *0.20* <u>0.92</u>	**0** *0.15* <u>0.94</u>	**0** *0.10* <u>0.92</u>	**0** *0.12* <u>0.94</u>
0:1:179	**0.05** *0.21* <u>0.89</u>	**0** *0.18* <u>0.97</u>	**0** *0.09* <u>0.96</u>	**0** *0.08* <u>0.95</u>	**0** *0.07* <u>0.97</u>

Relative error, A
Number of miss-classified pixels, B
<u>Measure of sharpness, C</u>

We now turn to the last table, 6.4. We first notice that a lot of noise has been added, which we saw from earlier chapters could ruin the reconstructions. This is exactly what the measurements show here. The relative error is quite high, and a high percentage of pixels has been miss-placed. Furthermore the sharpness value is very poor, which corresponds to what we found in Chapter 5. Again it seems that it does not matter if A is over- or underdetermined. For an example of reconstructions of this case, consider figure 6.5. As one can see the noise has completely ruined these reconstructions.

Once again the SIRT methods works in the same way for this kind of tests. Instead of listing it all once again in the report, we make a short summary of the important features here for the Kaczmarz method and the CGLS method. For the Kaczmarz method the same tend appear as for the SIRT method with one difference. Due to the artefact which Kaczmarz creates, when a lot of noise is added to our test problem, our reconstructions will have a very large number of the relative error. For Kaczmarz's method with *noiselevel* $= 0.4$ we refer to table 6.5. This once again show that Kaczmarz's method is very sensitive towards noise, since the relative error found for these reconstructions is very large.

Table 6.3: Method: Cimmino, with *noiselevel* = 0.1 and $N = 64$. Shows the different 3 types of measurement values obtained for different p and θ values. The table to the right is a helping table, indicating what measurement type the different measurement values corresponds to.

$\theta \backslash p$	0.8N	1N	1.2N	1.4N	1.6N
0:7.5:179	**1.46**	**0.71**	**0.34**	**0.27**	**0.10**
	0.37	*0.34*	*0.25*	*0.19*	*0.18*
	0.75	0.73	0.77	0.76	0.72
0:5:179	**0.93**	**0.44**	**0.15**	**0.15**	**0**
	0.31	*0.31*	*0.21*	*0.22*	*0.23*
	0.74	0.73	0.86	0.76	0.79
0:3.5:179	**0.39**	**0.24**	**0.20**	**0.07**	**0.02**
	0.29	*0.29*	*0.24*	*0.18*	*0.17*
	0.75	0.72	0.77	0.74	0.76
0:2.5:179	**0.29**	**0.20**	**0.02**	**0.05**	**0**
	0.28	*0.26*	*0.19*	*0.17*	*0.14*
	0.74	0.80	0.83	0.83	0.81
0:1:179	**0.12**	**0.07**	**0.05**	**0**	**0**
	0.26	*0.22*	*0.14*	*0.12*	*0.11*
	0.82	0.85	0.88	0.89	0.95

Relative error, A
Number of miss-classified pixels, B
Measure of sharpness, C

The CGLS method works similar to Kaczmarz and SIRT, but here a trend also appears. First of all it has the same feature as Kaczmarz, the CGLS method is sensitive towards noise. But furthermore it seems that CGLS have difficulties with keeping a reconstruction sharp. For this we have chosen to show the CGLS method for *noiselevel* = 0.1 since for *noiselevel* = 0.4 the results was dominated by noise. We observe table 6.6 for the values obtained, and as one notice, the values of the sharpness are much lower compared to the SIRT methods.

Therefore when one wishes to do a reconstruction one must ask the question "At which value can i accept the relative error"? When this is answered, one can look in the tables to find which variables are needed to obtain this value of the relative error.

A final feature to consider is if the size N affects the results in some way. For this we consider figure 6.6. As we can see, only by changing the size of the problem N, we still obtain the same relative error, and therefore we do not need to set up more tables like above for a different N value. On figure 6.7 we see the same trend, which is that the size of N in 3D does not interfere with the

Table 6.4: Method: Cimmino, with *noiselevel* $= 0.4$ and $N = 64$. Shows the different 3 types of measurement values obtained for different p and θ values. The table to the right is a helping table, indicating what measurement type the different measurement values corresponds to.

$\theta \backslash p$	0.8N	1N	1.2N	1.4N	1.6N
	5.32	**3.98**	**3.13**	**2.42**	**2.56**
0:7.5:179	*0.59*	*0.51*	*0.58*	*0.42*	*0.41*
	0.52	0.67	0.55	0.51	0.54
	3.96	**2.69**	**2.31**	**1.88**	**1.71**
0:5:179	*0.56*	*0.50*	*0.43*	*0.38*	*0.40*
	0.59	0.50	0.67	0.64	0.58
	3.08	**2.37**	**2.08**	**1.59**	**1.34**
0:3.5:179	*0.51*	*0.44*	*0.37*	*0.34*	*0.36*
	0.53	0.57	0.54	0.53	0.62
	2.29	**1.81**	**1.32**	**1.37**	**1.32**
0:2.5:179	*0.42*	*0.43*	*0.36*	*0.31*	*0.36*
	0.70	0.58	0.52	0.59	0.63
	1.07	**0.88**	**0.66**	**0.37**	**0.39**
0:1:179	*0.38*	*0.33*	*0.34*	*0.27*	*0.24*
	0.59	0.57	0.66	0.59	0.69

Relative error, A
Number of miss-classified pixels, B
Measure of sharpness, C

relative error of the reconstruction.

The tables for the 3D case are omitted here since the trend was the same as we saw it for 2D. In the next section we shall see how low values of p and θ our iterative methods can handle before the reconstructions are ruined for 2D and 3D problems.

6.2.1 Extreme Values of p and θ

Earlier we used the phrase "extreme values" to refer to the corners of our tables. Recall that the extreme values corresponded to $\theta = 0 : 7.5 : 179$ and $p = 0.8N$. In this section we wish to go even further and find the values where the iterative methods simply cannot create the reconstructions properly anymore. As we saw from the tables it is clear that the more noise we have added, the larger we have to choose our variables. For this reason we have decided to perform these tests with a constant noise level equal to 0.05. First look at figure 6.8 to examine the θ-values, where we have chosen $p = 1.6N$. Due to the fact

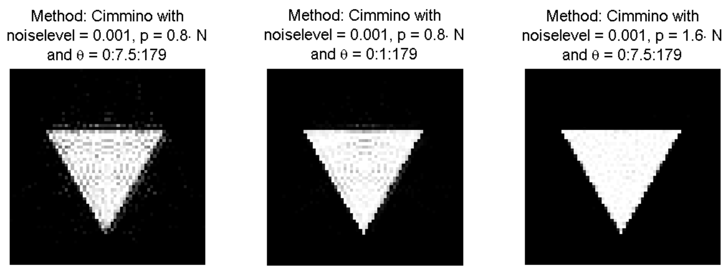


Figure 6.2: Examples of reconstructions for p and θ. As we notice, an increase in p results in a better reconstruction, whereas an increase in θ does not necessarily give the same result.

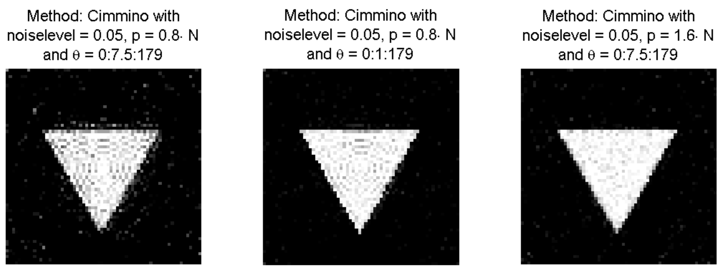


Figure 6.3: Examples of reconstructions for p and θ. As we notice, an increase in p results in a better reconstruction, whereas an increase in θ does not necessarily give the same result.

that from the last section, we saw that this value yielded a good reconstruction.

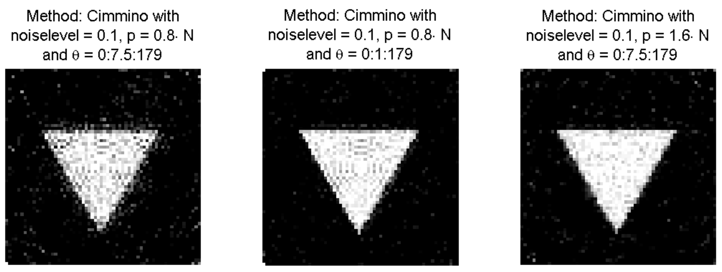


Figure 6.4: Examples of reconstructions for p and θ. As we notice, an increase in p results in a better reconstruction, whereas an increase in θ does not necessarily give the same result.

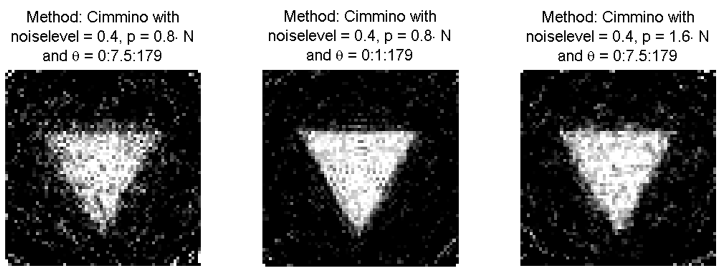


Figure 6.5: Examples of reconstructions for p and θ. As we notice, an increase in p results in a better reconstruction, whereas an increase in θ does not necessarily give the same result.

It is noticeable on figure 6.8 that even for a quite low θ-value we are still able visually to identify the grain. It is first at the value $\theta = 0 : 50 : 179$ that

Table 6.5: Method: Kaczmarz, with *noiselevel* = 0.4 and $N = 64$. Shows the different 3 types of measurement values obtained for different p and θ values. The table to the right is a helping table, indicating what measurement type the different measurement values corresponds to.

$\theta \backslash p$	0.8N	1N	1.2N	1.4N	1.6N
0:7.5:179	**4.74** *0.74* 0.61	**3.44** *0.53* 0.70	**2.51** *0.55* 0.63	**2.49** *0.76* 0.57	**2.20** *0.49* 0.63
0:5:179	**3.61** *0.58* 0.48	**2.69** *0.63* 0.62	**2.03** *0.49* 0.58	**1.73** *0.40* 0.66	**1.56** *0.42* 0.49
0:3.5:179	**2.91** *0.50* 0.65	**1.44** *0.53* 0.54	**1.68** *0.45* 0.58	**1.54** *0.40* 0.63	**1.44** *0.41* 0.64
0:2.5:179	**2.08** *0.48* 0.58	**1.71** *0.52* 0.58	**1.54** *0.65* 0.64	**1.25** *0.46* 0.63	**1.20** *0.51* 0.63
0:1:179	**1.54** *0.56* 0.62	**1.46** *0.62* 0.73	**1.12** *0.48* 0.60	**1.59** *0.54* 0.66	**1.51** *0.57* 0.74

Relative error, A
Number of miss-classified pixels, B
Measure of sharpness, C

the grain becomes blurry, which also only corresponds to use 4 different angles. But if we consider 6.9 something else is occurring. Here we see that when p is increased drastically, then we actually obtain two quite good reconstructions. But still for $\theta = 0 : 50 : 179$ the reconstruction is poor. In this way we have found a somewhat lower bound for θ. Let us see if the same applies for p.

We consider figure 6.10. As we can see for small values of p, the reconstructions are completely ruined. There are lot of artefact in the image, which is not good. If we consider figure 6.11, then we have changed θ by a factor 10, and we still obtain a poor reconstruction. The question is now at which value p could seem to have some kind of minimum, and for this purpose we observe figure 6.12. On this figure the lowest value for p starts to appear, and it is seen to be around $p = 0.8N$. Below $p = 0.8N$ the artefact is too dominating, but around this level the artefact tends to disappear.

Let us investigate if the same occurs for the 3D test problems. We have once again chosen to keep the noise level constant with *r1_max* as well. Recall that θ corresponds to *degree* for 3D and p corresponds to *u_max* for 3D. First let us consider figures 6.13 and 6.14. These figures illustrate when we are decreasing

Table 6.6: Method: CGLS, with *noiselevel* = 0.1 and N = 64. Shows the different 3 types of measurement values obtained for different p and θ values. The table to the right is a helping table, indicating what measurement type the different measurement values corresponds to.

$\theta \backslash p$	0.8N	1N	1.2N	1.4N	1.6N
0:7.5:179	**1.42** *0.41* 0.56	**0.68** *0.35* 0.51	**0.39** *0.27* 0.58	**0.17** *0.23* 0.56	**0.10** *0.21* 0.58
0:5:179	**0.81** *0.38* 0.54	**0.61** *0.35* 0.56	**0.29** *0.24* 0.58	**0.07** *0.20* 0.54	**0.05** *0.22* 0.56
0:3.5:179	**0.51** *0.35* 0.54	**0.22** *0.31* 0.54	**0.32** *0.23* 0.54	**0.10** *0.21* 0.54	**0.07** *0.18* 0.57
0:2.5:179	**0.37** *0.32* 0.55	**0.12** *0.28* 0.68	**0.07** *0.22* 0.70	**0.02** *0.22* 0.64	**0** *0.20* 0.70
0:1:179	**0.17** *0.30* 0.67	**0.02** *0.26* 0.70	**0.05** *0.20* 0.70	**0** *0.15* 0.64	**0** *0.16* 0.67

Relative error, A
Number of miss-classified pixels, B
Measure of sharpness, C

degree and as we can see, when *degree* = 6, we simply have to few direction vectors which is indicated by the fact that the grain consists of "curved" lines. If we increase *degree* = 14 we see that the grain is consisting of straight lines. This implies that the found grain is a tetrahedron, which we know it should be. From these figures it follows that a minimal value of *degree* = 14 should be chosen.

With this we turn our attention towards finding the lowest number of rays. To do this we consider figures 6.16 and 6.15. Figure 6.16 shows a reconstruction done with a large *degree* value and a low *u_max* value. As we see this reconstruction is still decent with not to much artefact within the reconstruction. This is even though $u\_max \approx 0.5r1\_max$. If we consider figure 6.15 we notice a difference. In this reconstruction a lot artefact tend to appear even though we have a large value of *degree*. This indicates that we have found a minimum value for *u_max* which is $u\_max \approx 0.5r1\_max$. If we go below this value, to much artefact will dominate the reconstruction.

By now we have looked at the different variables for the 2D case, and now we turn our point of interest towards the 3D case.

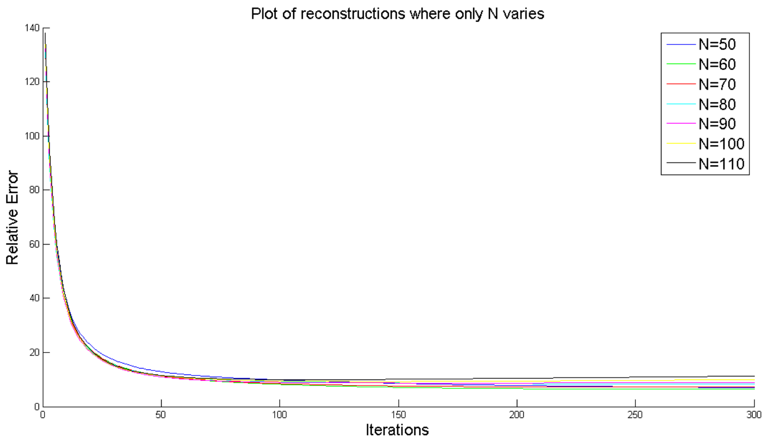


Figure 6.6: Example of reconstructions for Cimmino's method with standard w, p and θ in 2D. The variable we are varying is N. As one notice, the size of N does not interfere with the relative error of the reconstruction.

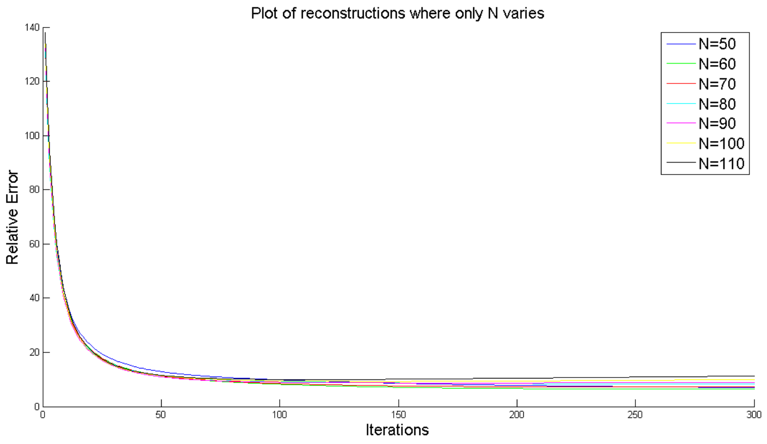


Figure 6.7: Example of reconstructions for Cimmino's method with standard w, p and θ in 3D. The variable we are varying is N. As one notice, the size of N does not interfere with the relative error of the reconstruction.

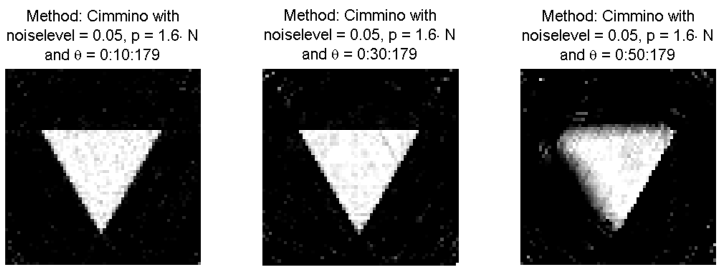

Method: Cimmino with noiselevel = 0.05, p = 1.6· N and θ = 0:10:179
Method: Cimmino with noiselevel = 0.05, p = 1.6· N and θ = 0:30:179
Method: Cimmino with noiselevel = 0.05, p = 1.6· N and θ = 0:50:179

Figure 6.8: Example of reconstructions for very small θ-values. As we can see we reach a limit of θ for when we cannot make a proper reconstruction.

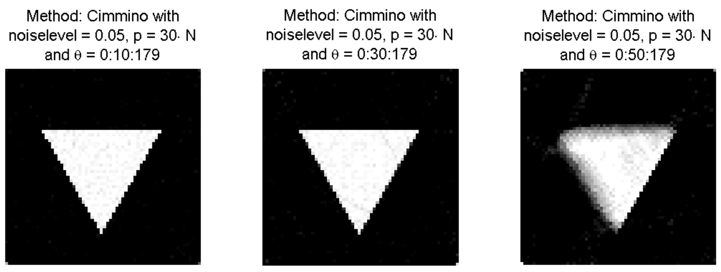

Method: Cimmino with noiselevel = 0.05, p = 30· N and θ = 0:10:179
Method: Cimmino with noiselevel = 0.05, p = 30· N and θ = 0:30:179
Method: Cimmino with noiselevel = 0.05, p = 30· N and θ = 0:50:179

Figure 6.9: Example of reconstructions for very small θ-values. As we can see we reach a limit of θ for when we cannot make a proper reconstruction.

6.3 "p" and "θ" in 3D

In Chapter 4 we saw that the method for constructing a 3D tomography problem is the same as in 2D. The fact that is important to notice is how the variables

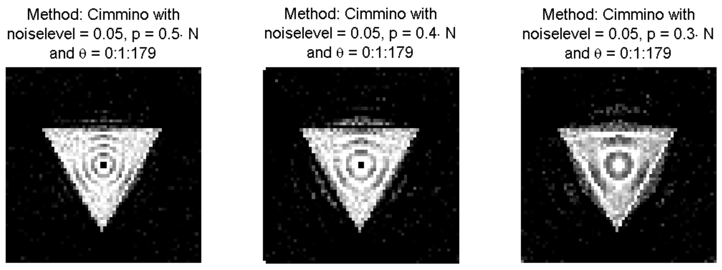

Method: Cimmino with noiselevel = 0.05, p = 0.5· N and θ = 0:1:179
Method: Cimmino with noiselevel = 0.05, p = 0.4· N and θ = 0:1:179
Method: Cimmino with noiselevel = 0.05, p = 0.3· N and θ = 0:1:179

Figure 6.10: Example of reconstructions for very small p-values. These reconstructions show that we need a larger p-value.

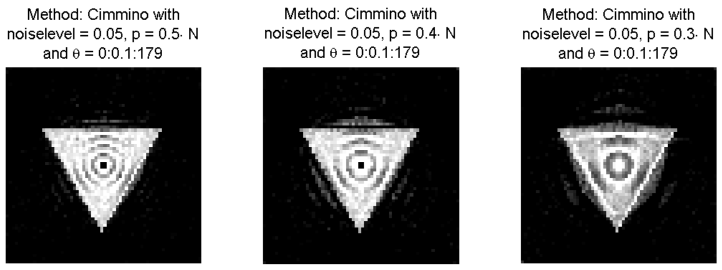

Method: Cimmino with noiselevel = 0.05, p = 0.5· N and θ = 0:0.1:179
Method: Cimmino with noiselevel = 0.05, p = 0.4· N and θ = 0:0.1:179
Method: Cimmino with noiselevel = 0.05, p = 0.3· N and θ = 0:0.1:179

Figure 6.11: Example of reconstructions for very small p-values. These reconstructions show that we need a larger p-value since it is not enough to increase θ.

in 2D correspond to the variables in 3D so similar tests can be performed. Recall that $r1\_max$ determines the size of N, since $N = 2 \cdot r1\_max + 1$. The function *getLebedevSphere* created a certain amount of direction vectors given a

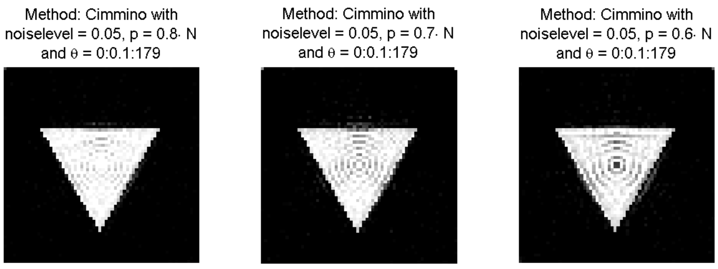


Figure 6.12: Example of reconstructions for very small p-values. These reconstructions show that we have found a minimum for the p-value.

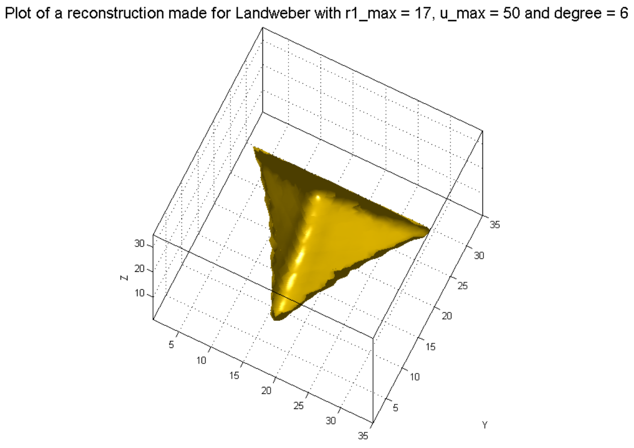

Plot of a reconstruction made for Landweber with r1_max = 17, u_max = 50 and degree = 6
Z
Y

Figure 6.13: Example of a reconstructions for a small *degree*-value. The reconstruction is made with Landweber, and standard $r1\_max$. $degree = 6$, $u\_max = 50$ while the noise added is 0.05. As we can see the grain is "curvy", meaning that the reconstruction does not consist of straight lines only.

parameter *degree*. The length of these vectors will correspond to the θ value from 2D, and last but not least, recall that all of our projections were discretized into

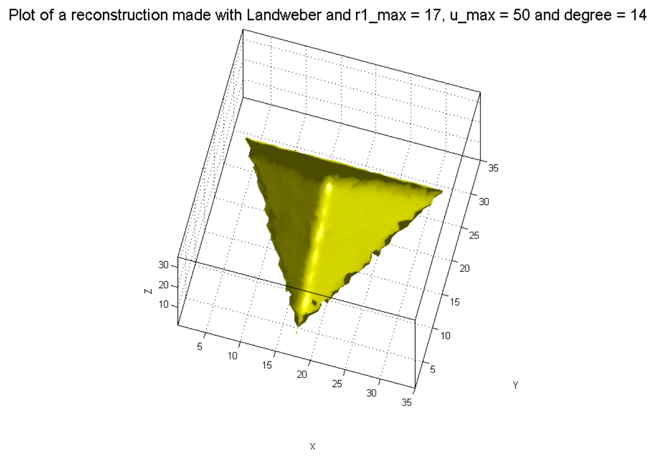

Plot of a reconstruction made with Landweber and r1_max = 17, u_max = 50 and degree = 14
Z
30
20
10
5
10
15
20
25
30
35
X
5
10
15
20
25
30
35
Y

Figure 6.14: Example of a reconstructions for a small *degree*-value. The reconstruction is made with Landweber, and standard $r1\_max$. $degree = 14$, $u\_max = 50$ while the noise added is 0.05. As we can see the grain consists of more straight lines now, indicating that we have reconstructed the edges of the grain properly.

$(2 \cdot u\_max + 1)^2$ pixels, which indicates that we have a total of $p = (2 \cdot u\_max + 1)^2$ rays. So to sum up we have the following result

$$\begin{aligned} N \text{ in 2D} &\sim r1\_max \text{ in 3D} \\ p \text{ in 2D} &\sim u\_max \text{ in 3D} \\ \theta \text{ in 2D} &\sim \text{length(v\_list) in 3D.} \end{aligned}$$

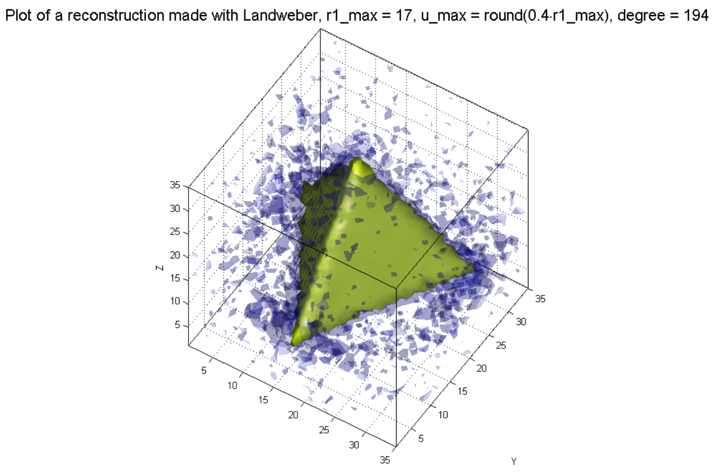
The size of A is determined by the size of the above variables, which is

$$\begin{aligned} row_A &= "p" \cdot length("\theta"), \\ column_A &= "N"^3, \end{aligned}$$

and when translated into 3D variables corresponds to

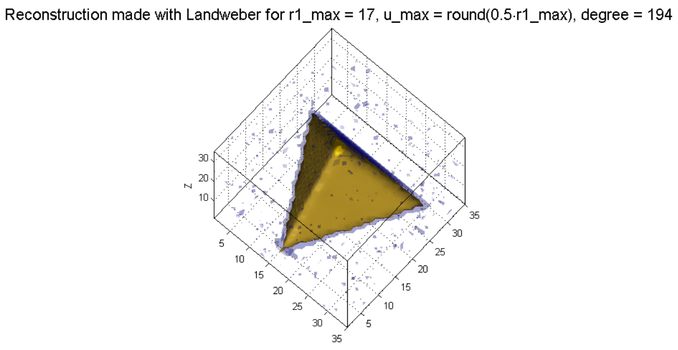
$$\begin{aligned} row_A &= (2 \cdot u\_max + 1)^2 \cdot length(v\_list), \\ column_A &= (2 \cdot r1\_max + 1)^3, \end{aligned}$$

Unfortunately everything is not perfect, and for special combination of values for the 3D test cases we can obtain reconstructions dominated by large voxel values. This will be considered in the next section.



Plot of a reconstruction made with Landweber, r1_max = 17, u_max = round(0.4·r1_max), degree = 194
Z
Y

Figure 6.15: Example of a reconstructions for a small u_max-value. The reconstruction is made with Landweber, and standard $r1\_max$. $u\_max = round(0.4 \cdot r1\_max)$, $u\_max = 50$ while the noise added is 0.05. As we can see the grain is "curvy", meaning that the reconstruction does not consist of straight lines only.



Reconstruction made with Landweber for r1_max = 17, u_max = round(0.5·r1_max), degree = 194
Z

Figure 6.16: Example of a reconstructions for a small u_max-value. The reconstruction is made with Landweber, and standard $r1\_max$. $degree = 14$, $u\_max = 50$ while the noise added is 0.05. As we can see the grain consists of more straight lines now, indicating that we have reconstructed the edges of the grain properly.

6.3.1 Extreme Reconstruction

As we shall see, given some special values of the variables $r1\_max, u\_max$ and $degree$, we will obtain some reconstructions where we cannot necessarily determine the best reconstruction by the 1-norm. The reason is that after very few iterations, voxel values have diverged towards a high number. We define all numbers above 3 as "a high number" when considering the test problems. For an illustration we refer to figure 6.17. On this figure every voxel that is above 3 has been plotted. The green circles indicate where to look for these values. Very few voxels have obtained this high value, but these high values will interfere with our measurement of quality, namely $\|x^k - x^{exact}\|_1$. Usually these values appear in the corners of our reconstructions but it does not necessarily have to be in the corners.

As we can see these values will affect our reconstructions, but the question is by "how much"? For this we observe figure 6.18. As we can see the two iterative methods Landweber and SART perform better reconstructions by approximately a factor 2 compared to the other 3 SIRT methods. The question is now what is going on since these large voxel values appear.

To explain this weird phenomenon we first recall that the large voxel values happened for the Cimmino, CAV and DROP method. It is not shown, but the same will happen for the Kaczmarz method. And the reason is found in the formulas for these iterative methods. Recall that their formulas are given in equation (3.8), (3.4), (3.5) and (3.6). All these methods have one important feature in common, which does not apply for the rest of the iterative methods. It is that at some point we divide by the term $\|a^i\|_2^2$. For the DROP method the term is $\|a^i\|_S^2$, but the issue remains the same. So why is this term of such importance? To answer this question we consider figure 6.19. We have a ray, indicated by the red line, which hits a corner of our object and nothing more. The corresponding row of such a ray in A will consists of only zeroes, except at that one point where we hit out object. In this example the ray has hit with a length of 10^{-4}. When we take the norm of such a row, the resulting value will be very close to zero, and when one divides with almost 0 we obtain a very high number. This is the reason these large valued voxels in our reconstructions appear.

It seems that for the iterative methods, where this phenomenon occurs, there is a certain system according to which we can expect the voxels to be of high value. As mentioned above, we only need a few iterations before these voxels will appear. For that reason a test has been made for different values of $r1\_max$ and u_max, where only 10 iterations has been made for each combination. The result can be seen in figure 6.20.

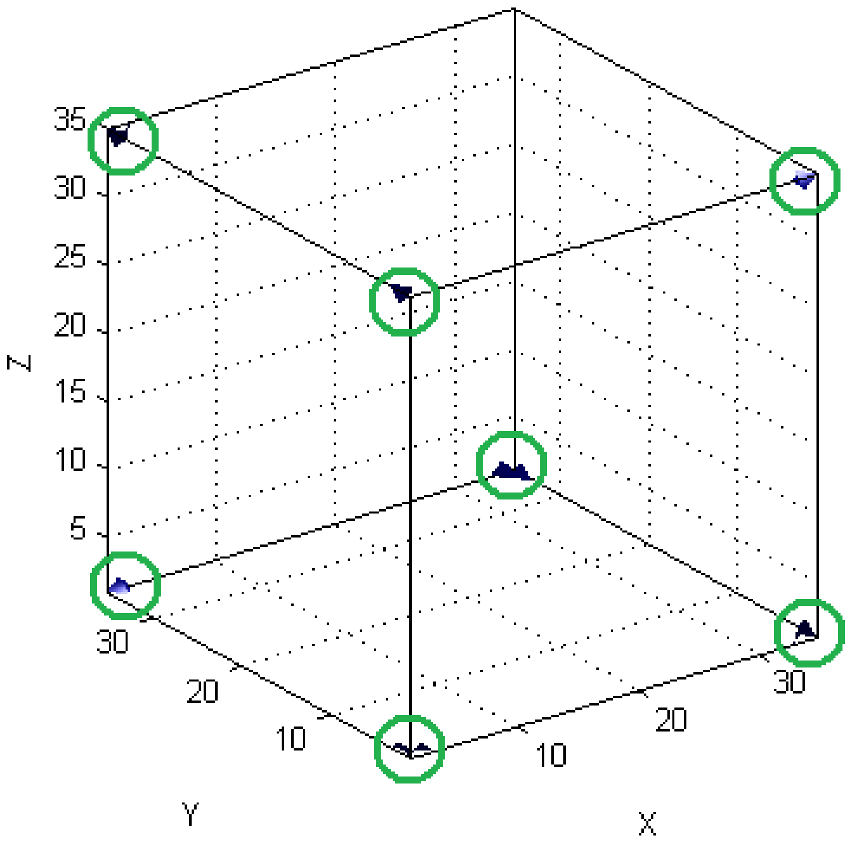


Figure 6.17: Plot of the artefact. The green circles indicate where the viewer should look to identify these small artefact. There are not many and they appear usually in the corners of the reconstructions.

So if we were to use one of these combinations for a reconstruction, we could expect the reconstruction to be dominated by these large voxel values, and therefore the optimal iteration found would not necessarily be the exact one. Therefore when doing reconstructions in 3D we should be very careful when choosing our parameters.

Unfortunately this also occur for Kaczmarz in 2D. When a reconstruction is made for Kaczmarz, the corners can appear "white" due to large pixel values. Therefore we need to ensure that a reconstruction made with high pixel values for Kaczmarz, is still appropriate. First we consider figure 6.21. As we can

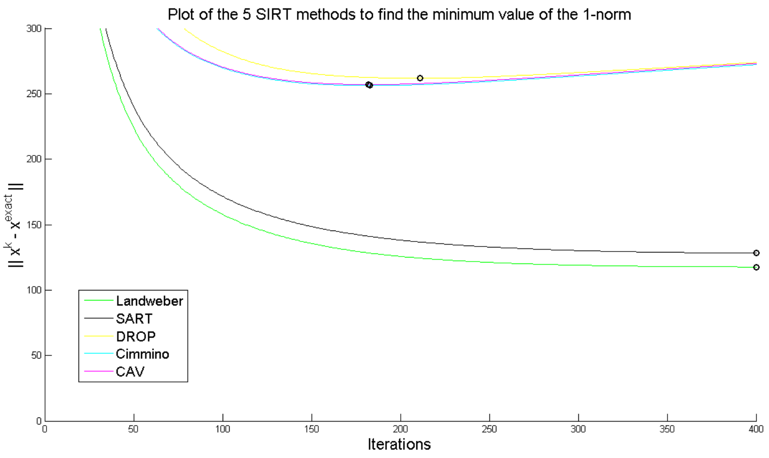


Figure 6.18: Plot of the 5 different SIRT methods for a case where some of the methods has high pixel values within their reconstructions. For this plot we have that $r1\_max = 15$, $u\_max = 23$ and $length(v\_list) = 38$. As noticed, Landweber and SART perform better reconstructions.

see there are few pixel values which are high. Hopefully they do not interfere to much with the solution found by the relative error. To investigate this we consider figure 6.22. As we can see from this, two plots are made. The blue line is the relative error found at each iteration for the reconstruction with high pixel values. The red line is the relative error per iteration for the reconstruction found without these high value pixels. As one can see these pixels does not interfere drastically with the results for 2D. Therefore we need to be extra careful when doing 3D reconstructions.

6.4 Summary

In this chapter we considered how the matrix A is set up. We know that the size of A is determined by a certain choice of variables, and the idea was to make A is small as possible, without ruining our reconstructions obtained. As we saw there were "extreme values" for both p and θ, and these variables were found for both 2D and 3D test problems. Also the effect of w was showed for the 2D test cases.

Unfortunately the iterative methods can for some special choices of variables

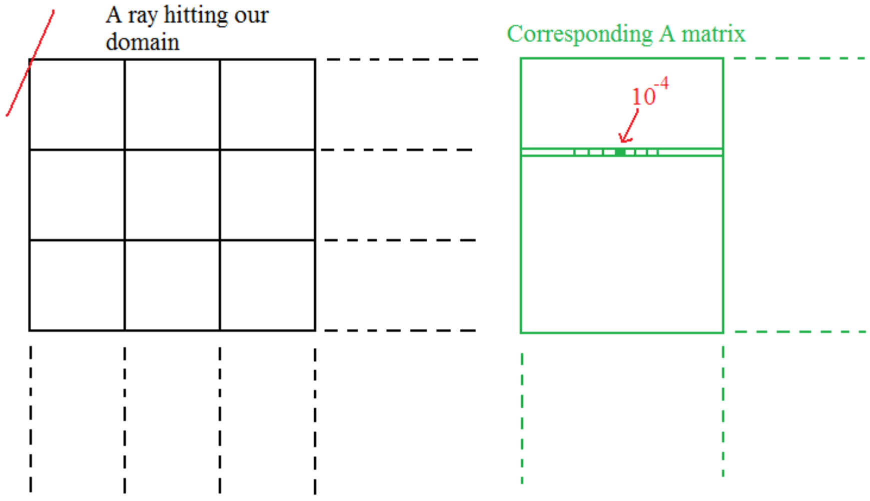


Figure 6.19: Illustrates why the term $\|a^i\|_2^2$ can be a problem. If a ray (the red line) is just hitting a corner, but not precisely, and does not hit anything else, the corresponding row in A (Green matrix) will consists of zeroes, except at that one point where the ray did hit. If this ray hits with a length very close to 0, the corresponding value in A will be close to 0, in this case it is set to 10^{-4}.

create reconstructions which are dominated by large pixel/voxel values. We saw examples of these, and for the 3D case, we plotted the combinations to see which choice of variables one should not use. In 2D, these large values did not interfere with the reconstruction in a significant way.

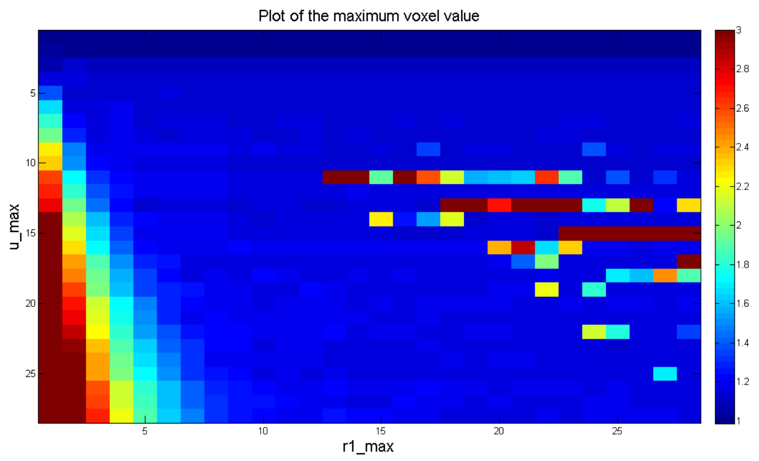


Figure 6.20: Illustration of where we can expect the 4 iterative methods to give artefact. On the horizontal axis we have $r1\_max$ and on the vertical u_max. For each combination of these values, 10 iterations has been made and the maximum voxel value has been entered. Everything above 3 has been set equal to 3. The method used for this plot was Cimmino, where $length(v\_list) = 38$.

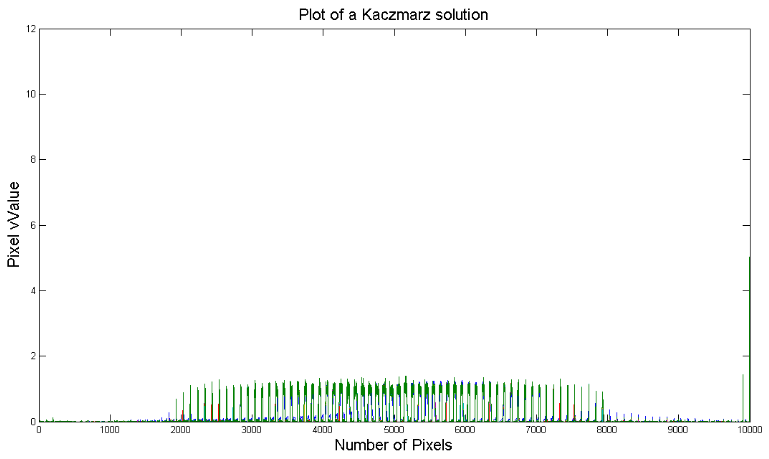


Figure 6.21: Plot of a reconstruction made with Kaczmarz. As we can see there are few high value pixels which interfere with our solution.

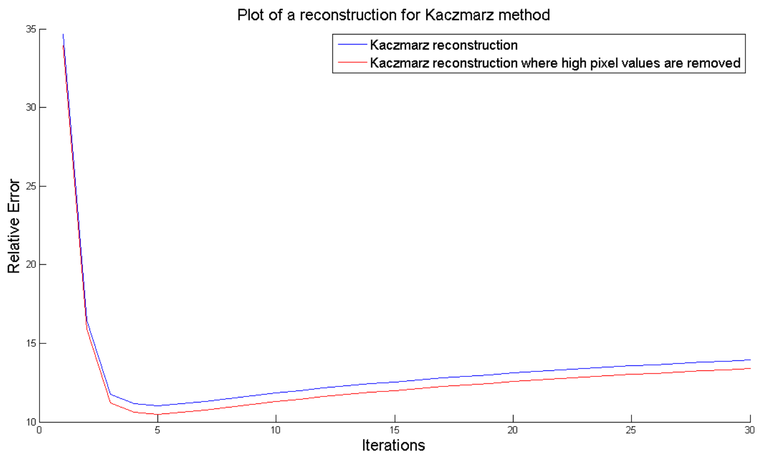


Figure 6.22: Plot of a reconstruction made with Kaczmarz. Two vectors are plotted against each other, one is where we have calculated the relative error while keeping the high pixel values. The other is where we have calculated the relative error and removed the high pixel values.

Chapter 7

The Influence of Geometric Error

Until now we have studied all the variables and discussed how they should be chosen. Therefore we are now able to solve a problem defined as (2.1) where the right hand side consists of noise. In this section we will investigate a different type of error - the geometrical error.

From (2.1) we know that our right hand side b consists of the exact b and some added noise e, see (2.10). We also know from the real world that every time we perform tests to measure observations or data, the data can be filled with noise. So what if the matrix A is filled with error, due to the fact that the rays being sent through are actually not being sent through in the way that we think? To illustrate this see figure 7.1. As we can se from figure 7.1, the black pixels shown by full lines, illustrate where we think our pixels are located, and the orange pixels shown by a dotted line, illustrate where the pixels are precisely located. The red arrows are the number of rays penetrating our object. It is clear that the new matrix A_{GEO} we obtain should be completely different from the matrix A_{Normal}. Here A_{GEO} is the matrix we obtain by simulating geometrical error in A, and A_{Normal} is the matrix we would obtain if no geometrical error is simulated. From the example we would expect ray_1 to hit pixel 4 only, whereas it actually hits pixel 2,3 and 4. For the 3D case, the principle is exactly the same.

For this reason there are two types of error we can measure, and that is if

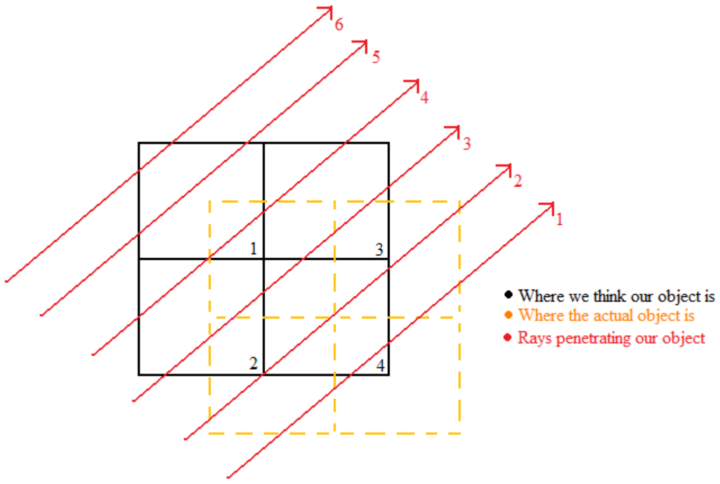

6
5
4
3
2
1
1
3
2
4
● Where we think our object is
● Where the actual object is
● Rays penetrating our object

Figure 7.1: Illustration of geometric error. The idea is that we think our domain is located within the black pixels. But the fact is that our pixels have been translated, indicated by the orange domain. Therefore the orange domain corresponds to where our pixels are located, and the black corresponds to where we think our pixels are located. As we can see, we would think ray_1 only hits pixel number 4, where in fact it hits pixel number 2,3 and 4.

there is geometric error is A or in b, indicating that we have two new problems which we need to make reconstructions from. They are

$$1. \qquad \overline{A}x = b \tag{7.1}$$

$$2. \qquad Ax = \overline{b}, \tag{7.2}$$

where $\overline{A}$ denotes a geometric error in A and $\overline{b}$ denotes a geometric error in b. If we would set up $\overline{A}x = \overline{b}$, it would correspond to the original problem from (2.1), which we have already tested.

7.1 Simulation of a Geometric Error

Recall from Chapter 4, that our rays were distinguished from each other by the different points P on each ray. We wish to simulate a geometric error in a systematic way, where first we consider a simple translation of the lines so we need to translate all the points P_i. To illustrate this see figure 7.2. We see for a given δ-value in the x- y-and z-coordinate system, that our point will move in the specified direction. Hereby we can determine what this translation should correspond to in pixels.

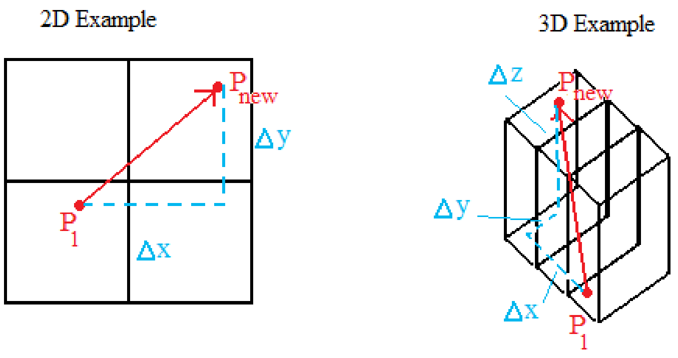


Figure 7.2: Illustration of how we translate a point along all the axes available. We give a Δ value in each coordinate, and by this value we translate all our points P_i.

This has been implemented in MATLAB, and to test if it is done correctly, meaning that we obtain the correct A matrix, we show a simple example. Consider figure 7.3. In this 2D example, we know that

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}.$$

If we imagine that $(\Delta x, \Delta y) = (1, 1)$, it will correspond to a translation of all the lines 1 pixel to the right and 1 pixel up. In this simple example it would mean that $ray_{1,new} = ray_2$ and $ray_{3,new} = ray_4$. The rays $ray_{2,new}$ and $ray_{4,new}$ would not hit anything and should consist of zeroes only. This is exactly the case as seen on figure 7.4, and therefore the simulation works as planned.

Exactly the same principle has been implemented for the 3D case, and for illustration see figure 7.5. As we can see some of the lines are "empty", corresponding to the fact that they are not hitting our object and therefore consist of zeroes only. Therefore we seem able to simulate translation and geometric error, which will be used to investigate if the iterative methods can produce decent reconstructions for this phenomenon.

Therefore let us see what will happen when this error is simulated. There will be a problem with the measure of the relative error since our x_{exact} does

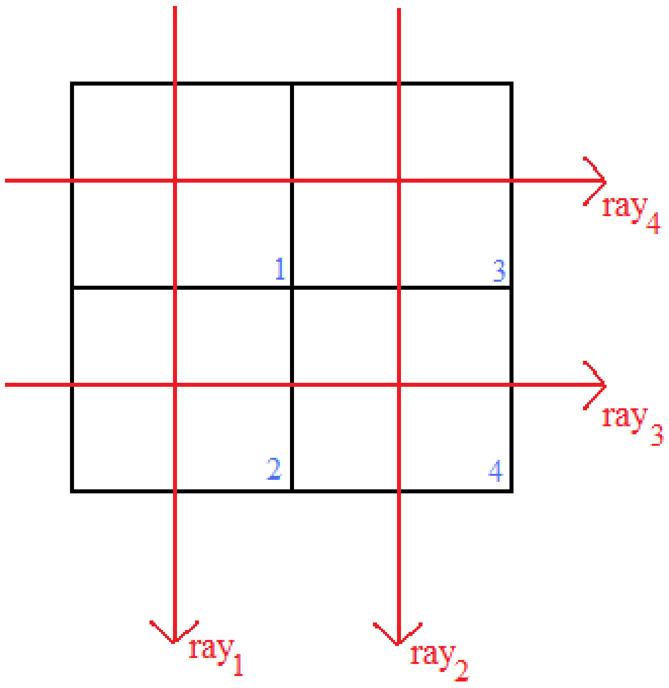


Figure 7.3: Illustration of how the A-matrix is being created. As we can see ray_1 hits pixel number 1 and 2, and therefore in the corresponding A matrix, ray_1 should have a zero for pixel 3 and 4. From figure 7.4 we see the A matrix created from this problem to the left.

not lie where our $x^{k^{opt}}$ lies. When we simulate this kind of geometrical error, we correct for the error by moving our x_{exact} by the same amount as we translate our rays. We consider figures 7.6, 7.7 and 7.8. The first figure, 7.6 shows a reconstruction where the translation is only present at the x-coordinate. The red cross indicates the middle of the image, and as we can see we have obtained two good reconstructions with a sharp edge. On the two other figures the same trend is seen. Even though the figures are moved, the iterative methods are still able to produce a good reconstruction with a sharp edge. Notice that the methods use almost the same amount of iterations. This does not surprise us though, since we translate x_{exact} the same amount as we do for the matrix A when measuring the relative error.

This was a geometrical error simulated as $\overline{A}x = b$. The next thing is to investigate what will happen if we do it for b instead. We consider figures 7.9,

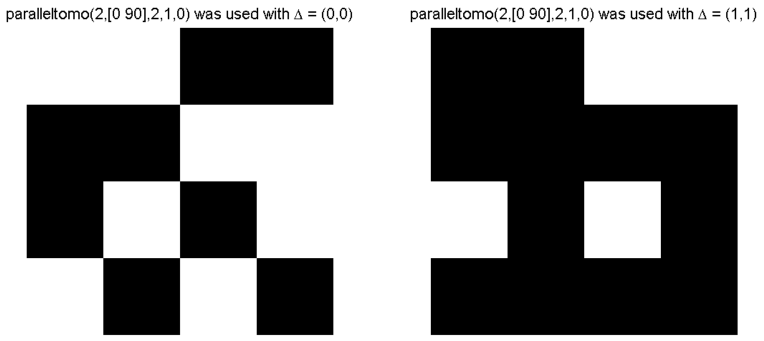


Figure 7.4: Shows the zeroes and ones of A where $N = 2$, $w = 1$, $p = 2$ and $\theta = [0\ 90]$. A white pixel indicates a value of 1 and a black pixel indicates a value of 0. Different values of Δ have been used for the two images. As we can see our new matrix A_{GEO} is different from A_{Normal}.

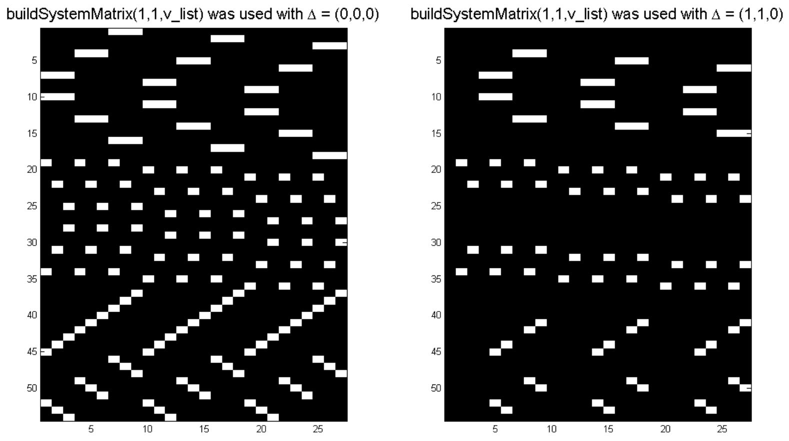


Figure 7.5: Shows the zeroes and ones of A where $N = 3$, $p = 9$ and $v\_list \sim getLebedevSphere(6)$. A white pixel indicates a value of 1 and a black pixel indicates a value of 0. Different values of Δ has been used for the two images. As we can see our new matrix A_{GEO} is different from A_{Normal}.

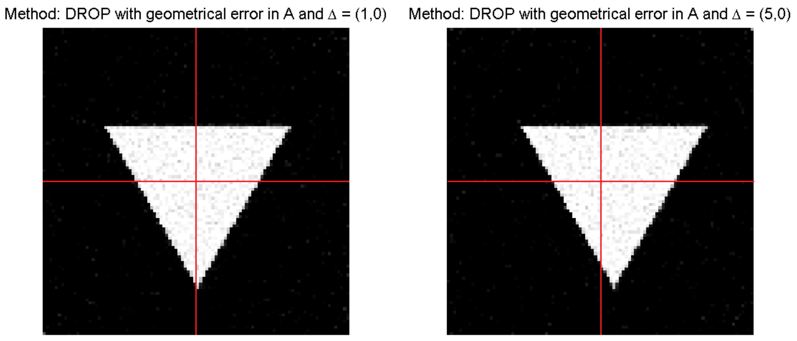


Figure 7.6: Shows two reconstructions made for standard variables, $noiselevel = 0.05$ with DROP as a method. Both reconstructions simulate a geometric error in A, and has only been translated in the x-direction. The figure to the left has been translated a small value, while the right figure is translated a larger value. The optimal iteration number for the right reconstructions is 218 and $max_{slope} = 0.8706$. For the left reconstruction we have optimal iteration = 220 and $max_{slope} = 0.8774$. The red cross indicates the middle of the image.

7.10 and 7.11. All these 3 figures illustrate a geometrical error in b. The trend in these figures corresponds to the trend found when we simulate a geometrical error in A. We see that the reconstructions are good, with a sharp edge, and the iterative methods are able to handle this kind of error. Also the optimal iterations are close to each other, which is what we would expect. The only noticeable thing here is that when we simulate a geometrical error in A, then the figure is moved in 2 directions. In this case it is moved in the positive x-direction and the positive y-direction. If we simulate a geometrical error in b, then the images are moved in the two opposite directions, negative x and negative y.

The reason for this phenomenon is best illustrated with a simple example. We consider the following

$$A_{true} = eye(4), \quad b_{true} = (1,2,3,4)^T,$$

where A_{true} is our geometrical matrix, in this case an identical matrix of size 4×4. b_{true} corresponds to our right hand side. The exact solution if found by

$$x_{true} = A_{true} \backslash y_{true} = (1,2,3,4)^T,$$

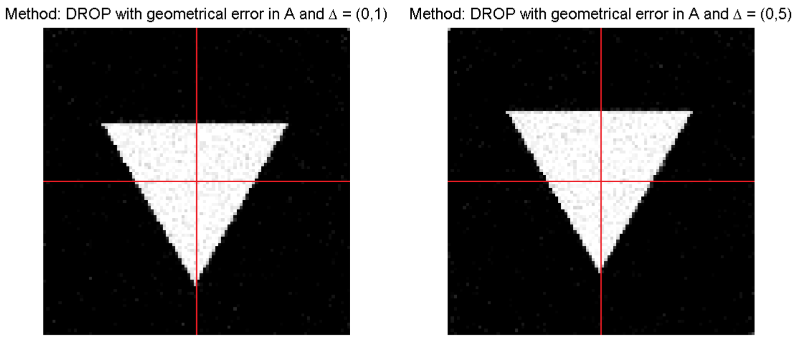


Figure 7.7: Shows two reconstructions made for standard variables, $noiselevel = 0.05$ with DROP as a method. Both reconstructions simulate a geometric error in A, and has only been translated in the y-direction. The figure to the left has been translated a small value, while the right figure is translated a larger value. The optimal iteration number for the right reconstructions is 210 and $max_{slope} = 0.8804$. For the left reconstruction we have optimal iteration = 231 and $max_{slope} = 0.8756$. The red cross indicates the middle of the image.

where "backslash" in Matlab solves our system of equations. We now simulate the geometrical error, indicating that we will get a new A matrix

$$A_{false} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad \Rightarrow \quad b_{false} = A_{false} \cdot x_{true} = (2,3,4,1)^T.$$

Now we can find the results with the geometrical error in both A and b resulting in

$$\begin{aligned} x_{GEO,A} &= A_{false} \backslash b_{true} = (4,1,2,3)^T \\ x_{GEO,b} &= A_{true} \backslash b_{false} = (2,3,4,1)^T. \end{aligned}$$

We notice from this example, that if we simulate a geometrical error in A, we will move in a certain direction, in our example all the values have been moved 1 arbitrary length downwards. If we simulate a geometrical error in b we see from the example that the values have moved 1 arbitrary length upwards. The result shows that when we simulate a geometrical error in either A or b, our reconstruction will move in a specified direction for A and in the opposite direction for b. This is exactly what the reconstructions showed us.

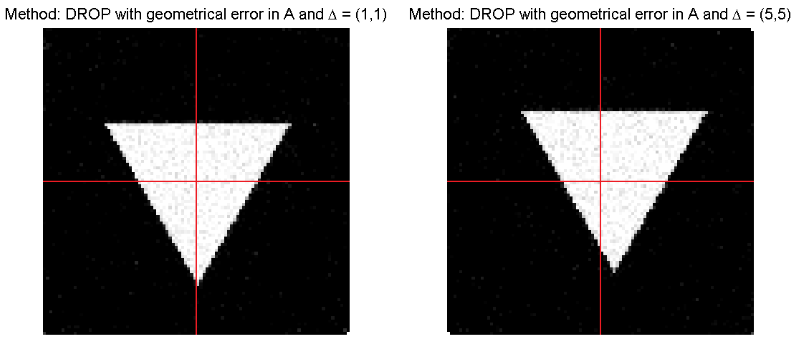

Method: DROP with geometrical error in A and Δ = (1,1)
Method: DROP with geometrical error in A and Δ = (5,5)

Figure 7.8: Shows two reconstructions made for standard variables, $noiselevel =$ 0.05 with DROP as a method. Both reconstructions simulate a geometric error in A, and has been translated in the x- and y-direction. The figure to the left has been translated a small value, while the right figure is translated a larger value. The optimal iteration number for the left reconstruction is 215 with corresponding $max_{slope} = 0.8721$. For the right reconstruction we have optimal iteration $= 226$ and $max_{slope} = 0.8695$. The red cross indicates the middle of the image.

On figures 7.12 and 7.13 we see a geometrical error simulated for the CGLS method and the Kaczmarz method. The reconstructions look like what we expected. The CGLS reconstructions are not that sharp, and the Kaczmarz reconstruction tend to have artefact in the image. The trend is exactly like above, when a reconstruction is translated due to geometrical error in A, the reconstruction obtained by geometrical error in b is translated the opposite direction.

As we shall see the case is exactly the same for 3D test problems. We consider figure 7.14. On this figure we have showed the original grain which corresponds to the middle grain. As we can see, geometrical error in A has moved the reconstruction in one direction and geometrical error in b has moved the reconstruction in the opposite direction. This is exactly what we saw in the 2D case.

Therefore it seems that all our iterative methods are able to do good reconstructions even if this phenomenon is present. In the next section we will look upon a different way of simulating geometrical error.

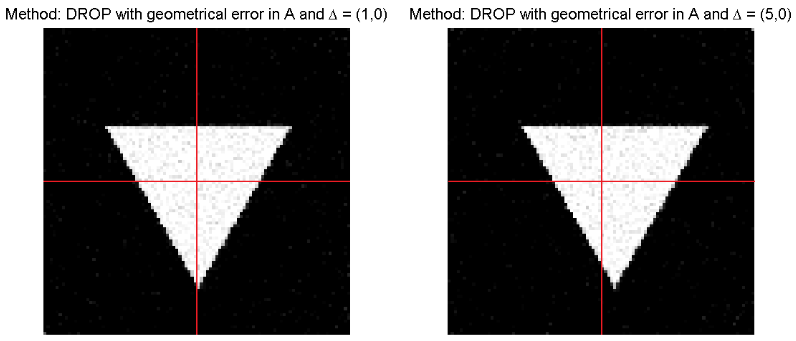

Method: DROP with geometrical error in A and Δ = (1,0)
Method: DROP with geometrical error in A and Δ = (5,0)

Figure 7.9: Shows two reconstructions made for standard variables, $noiselevel = 0.05$ with DROP as a method. Both reconstructions simulate a geometric error in b, and has only been translated in the x-direction. The figure to the left has been translated a small value, while the right figure is translated a larger value. The optimal iteration number for the right reconstructions is 232 and $max_{slope} = 0.8725$. For the left reconstruction we have optimal iteration $= 237$ and $max_{slope} = 0.8687$. The red cross indicates the middle of the image.

7.2 Translation in each Angle

Earlier we have considered a geometrical error where we translated all the points by a specific Δ. In this section, we will consider the final type of a geometrical error. This type is similar to the type from the last section, but instead of translating every point the same distance Δ, we assume that our grain is ”moving”. This means that given an angle θ_i for $i = 1, \ldots, length(\theta)$, our figure should be translated differently for each angle. Let us illustrate this with an example. We choose our parameters

$$\theta = [0, 45, 90, 135], \quad \Delta = (4, 4).$$

This means that we have 4 different angles for which we let our rays penetrate our domain, and from the earlier examples we have learned that we would then translate all our points 4 pixels in the x-direction and the y-direction. The idea is now that we translate the point in such a way that

$$\Delta_i = \left(\frac{\Delta_x}{length(\theta)} \cdot i, \frac{\Delta_y}{length(\theta)} \cdot i \right), \quad \text{for } i = 1, \ldots, length(\theta).$$

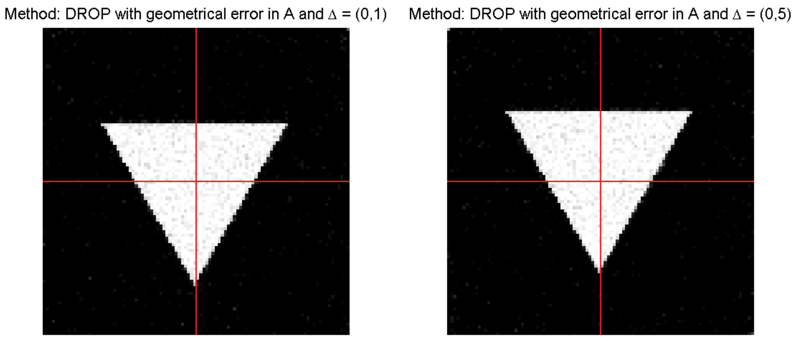


Figure 7.10: Shows two reconstructions made for standard variables, $noiselevel = 0.05$ with DROP as a method. Both reconstructions simulate a geometric error in A, and has only been translated in the y-direction. The figure to the left has been translated a small value, while the right figure is translated a larger value. The optimal iteration number for the right reconstructions is 241 and $max_{slope} = 0.8901$. For the left reconstruction we have optimal iteration = 228 and $max_{slope} = 0.8723$. The red cross indicates the middle of the image.

Here Δ_x corresponds to the x-coordinate of Δ and Δ_y corresponds to the y-coordinate of Δ. From this example we would get that when $\theta = 0$ we would translate all our points 1 pixel. If $\theta = 45$ we would translate our points 2 pixels and so on. By doing this we end up translating all our points differently for each angle, and with this we introduce the concept "drifting".

The new feature is that we do not know how much the total figure will be translated, and therefore we cannot translate the exact grain onto the reconstruction. Therefore we cannot make use of the relative error, since this value would be meaningless. In the next tests of this simulation, plots of the reconstructions will be performed in a systematic way to observe if the iterative methods can handle this phenomenon. This means that the iteration we are showing has been chosen manually. Let us consider some reconstructions made with drift. First let us investigate a geometrical error in b. We consider figures 7.15 and 7.16. A geometrical error in b has been simulated on these reconstructions, and as we can see for these reconstructions, systematic artefact around the edges of the grain tends to appear. The more we increase Δ, the more artefact we get. It can be hard to see, but also the more we increase Δ, the smaller the reconstruction will appear.

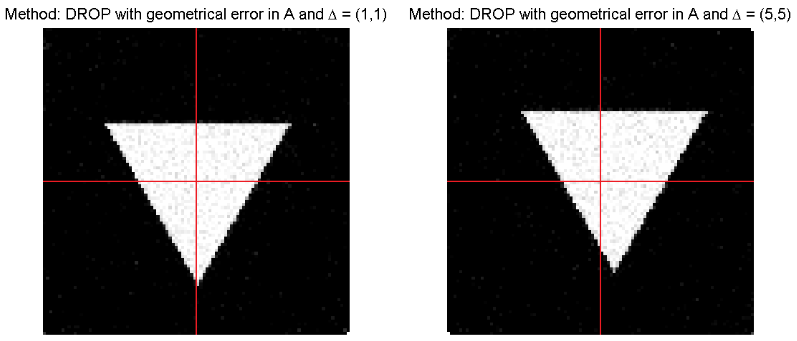

Method: DROP with geometrical error in A and Δ = (1,1)
Method: DROP with geometrical error in A and Δ = (5,5)

Figure 7.11: Shows two reconstructions made for standard variables, $noiselevel = 0.05$ with DROP as a method. Both reconstructions simulate a geometric error in A, and has been translated in the x- and y-direction. The figure to the left has been translated a small value, while the right figure is translated a larger value. The optimal iteration number for the left reconstruction is 227 with corresponding $max_{slope} = 0.8693$. For the right reconstruction we have optimal iteration = 236 and $max_{slope} = 0.8773$. The red cross indicates the middle of the image.

Let os see if the same applies for A. We consider figures 7.17 and 7.18. As we notice, an increase in Δ does create artefact within the reconstructions, but they tend to appear in the corners of the grain instead of on the edges. Also by increasing Δ, the size of the grain increases as well.

For the Kaczmarz method the artefact tend to appear in a different way, and for this we consider figure 7.19. Here the geometrical error has been simulated in A, and as we can see it seems like the half left side of the grain has a lower pixel value than the right half grain. The picture to the left is after 5 iterations and the picture to the right is after 10. The rest of the reconstructions are omitted, but the Kaczmarz reconstructions shows the same trend as the DROP method. A geometric error in A would increase the size of the grain whereas a geometric error in b decreases the grain. The images of CGLS have been omitted, since the same trend appeared as for the SIRT methods.

Let us consider the 3D examples to investigate if the same behavior occurs, and first we consider figures 7.20 and 7.21. In these test problems, all artefact

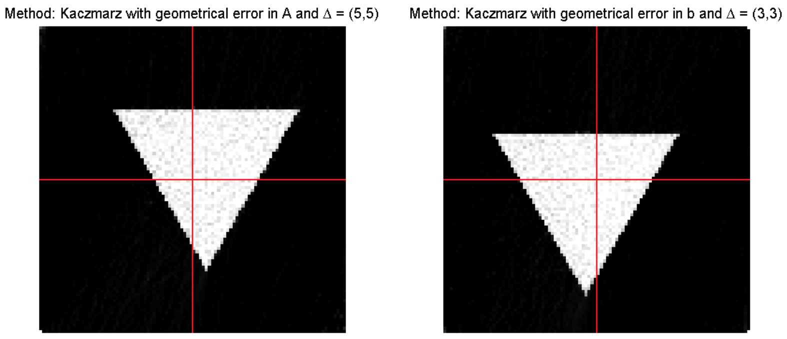


Figure 7.12: Illustrates the geometrical error when reconstructions are made with Kaczmarz's method. The figure to the left is with a geometrical error in A and $\Delta = (5, 5)$. The figure to the right is with geometrical error in b and $\Delta = (3, 3)$. Both figures have been reconstructed where the noise added was 0.05. As one will notice, when the geometrical error is simulated in A and when the geometrical error is simulated in b, then the figure moves in opposite directions.

shown on reconstructions are shown with "the same voxel value". With this we mean that when we plot our reconstructions, we plot every voxel value around 0.1. This will indicate the artefact created within the reconstructions. These two figures illustrate the simulation of drift for the SIRT methods. As we can see, even though we have a low noise level, artefact still appear. Also the reconstruction is not good, and there seems to be some sort of deformation of the grain. This could indicate that the SIRT methods will have difficulties when reconstructing such problems.

We now consider figure 7.22. On this figure, drift has been simulated in both A and b. The figure to the left is the reconstruction with drift in A, whereas the figure to the right is for b. Again we notice artefact appearing around our grain, and also some deformation of our grain. This indicates that Kaczmarz's method has difficulties reconstructing these type of problems.

The final figures to consider are for the CGLS method, and these are shown in figure 7.23. Also here artefact tend to appear, the only big difference in this method compared with the SIRT and Kaczmarz's method, is that CGLS tend to create a lot more artefact than the other methods. More or less this is what

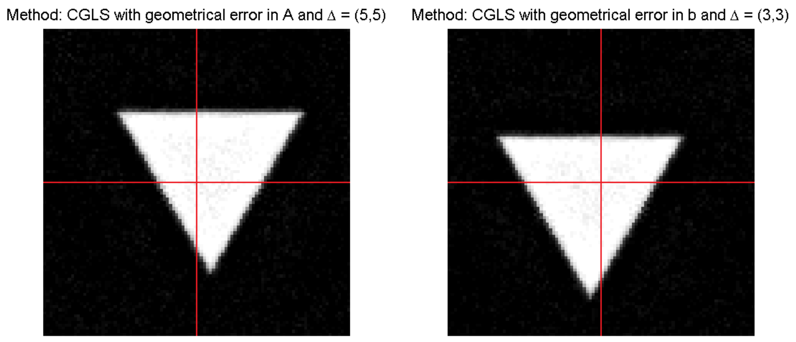

Method: CGLS with geometrical error in A and Δ = (5,5)
Method: CGLS with geometrical error in b and Δ = (3,3)

Figure 7.13: Illustrates the geometrical error when reconstructions are made with the CGLS method. The figure to the left is with a geometrical error in b and $\Delta = (5,5)$. The figure to the right is with a geometrical error in b and $\Delta = (3,3)$. Both figures have been reconstructed where the noise added was 0.05. Notice, when the geometrical error is simulated in b the figure is translated in one direction and when the geometrical error is simulated in A then the figure is translated in the opposite direction.

we could expect since CGLS have proven to be worse generally compared with the other methods.

7.3 Summary

In this chapter we have seen reconstructions made, where a geometrical error in either A or b was present. It was shown that for all the iterative methods, we were able to find a good reconstruction when the geometrical error was translated identical for every angle. Hereafter the concept drifting was simulated, and as we saw all the iterative methods had difficulties with reconstructing these types of problems. The size of the grain both decreased and got increased, depending on if the geometrical error was located in A or b. Also artefact tend to appear for these kind of problems.

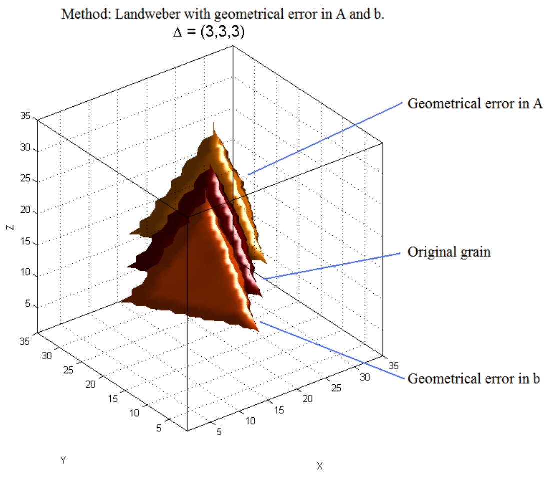


Figure 7.14: Illustrates the geometrical error when reconstructions are made with Landweber's method. The noise level is 0.05 with standard variables used, with $scalar = 0.25$. $\Delta = (3, 3, 3)$. As we can see from this figure, geometrical error in A has moved the grain in 1 direction, whereas geometrical error in b has moved it in the opposite direction.

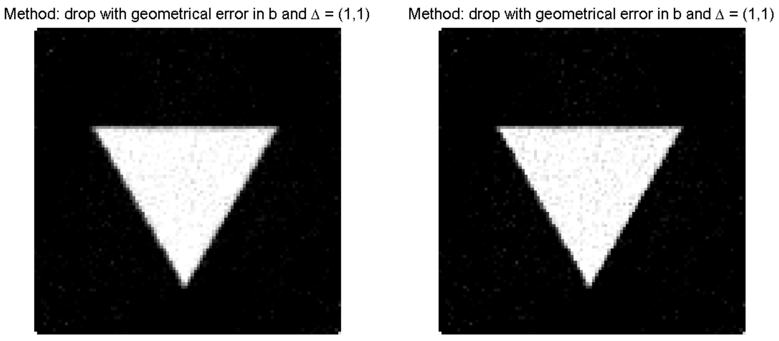


Figure 7.15: Illustrates a geometrical error in b made with the concept drift. The method used is DROP, with $noiselevel = 0.05$ and standard variables. The reconstructions are made with the same $\Delta = (1, 1)$. The left is made with $iteration = 100$ and the image to the right is with $iteration = 200$.

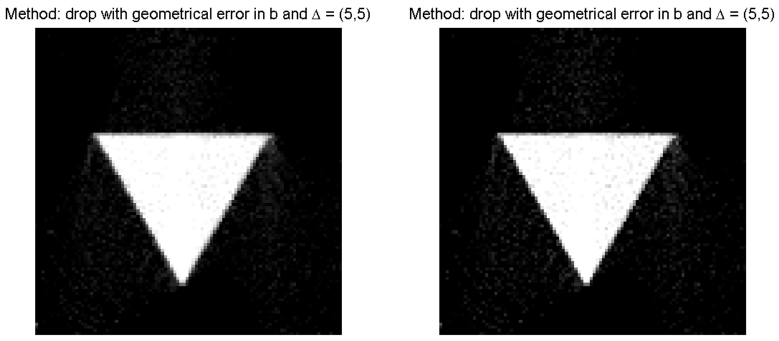

Method: drop with geometrical error in b and Δ = (5,5)
Method: drop with geometrical error in b and Δ = (5,5)

Figure 7.16: Illustrates a geometrical error in b made with the concept drift. The method used is DROP, with $noiselevel = 0.05$ and standard variables. The reconstructions are made with the same $\Delta = (5, 5)$. The image to the left is made with $iteration = 100$ and the image to the right is with $iteration = 200$.

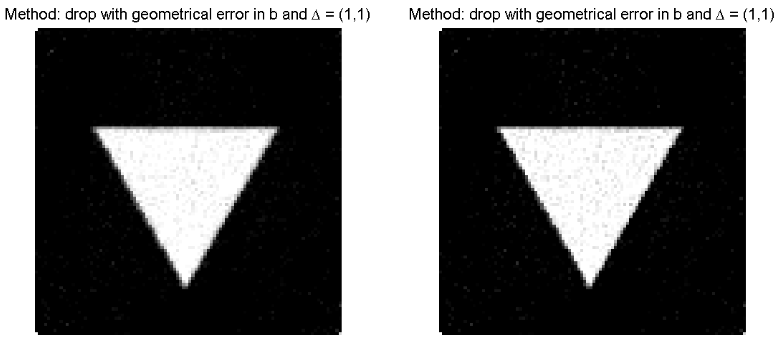

Method: drop with geometrical error in b and Δ = (1,1)
Method: drop with geometrical error in b and Δ = (1,1)

Figure 7.17: Illustrates a geometrical error in A made with the concept drift. The method used is DROP, with $noiselevel = 0.05$ and standard variables. The reconstructions are made with the same $\Delta = (1, 1)$. The image to the left is made with $iteration = 100$ and the image to the right is with $iteration = 200$.

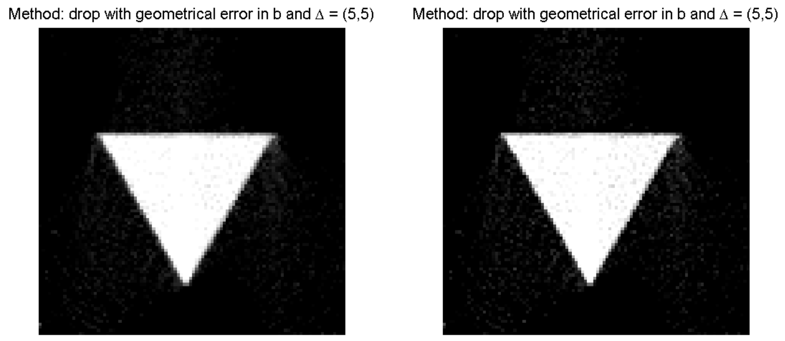

Method: drop with geometrical error in b and Δ = (5,5)
Method: drop with geometrical error in b and Δ = (5,5)

Figure 7.18: Illustrates a geometrical error in A made with the concept drift. The method used is DROP, with *noiselevel* $= 0.05$ and standard variables. The reconstructions are made with the same $\Delta = (5, 5)$. The image to the left is made with *iteration* $= 100$ and the image to the right is with *iteration* $= 200$.

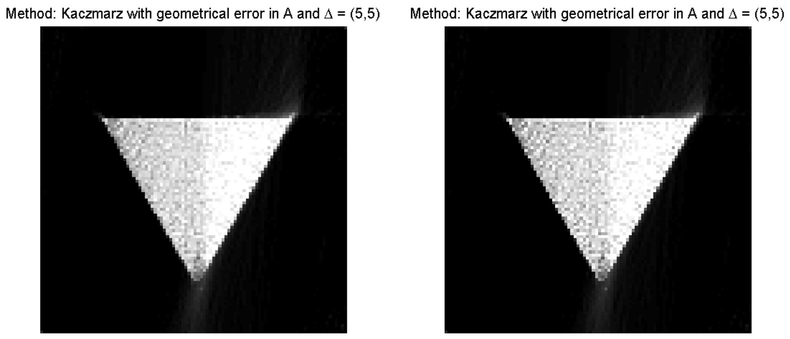

Method: Kaczmarz with geometrical error in A and Δ = (5,5)
Method: Kaczmarz with geometrical error in A and Δ = (5,5)

Figure 7.19: Illustrates a geometrical error in A made with the concept drift. The method used is Kaczmarz, with *noiselevel* $= 0.05$ and standard variables. The reconstructions are made with the same $\Delta = (5, 5)$. The image to the left is made with *iteration* $= 5$ and the image to the right is with *iteration* $= 10$.

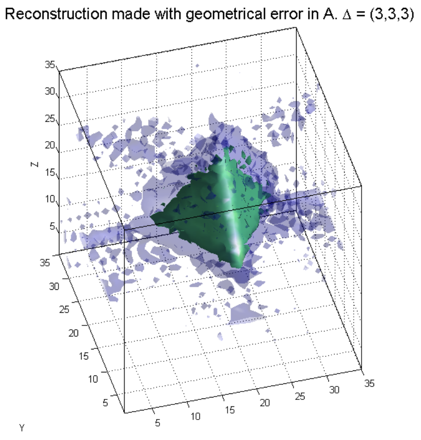

Reconstruction made with geometrical error in A. Δ = (3,3,3)
Z
Y
5
10
15
20
25
30
35

Figure 7.20: Illustrates a reconstruction made with Landweber where we have simulated drift in A with $\Delta = (3, 3, 3)$. Standard variables have been chosen with $noiselevel = 0.05$. The iteration shown is number 200.

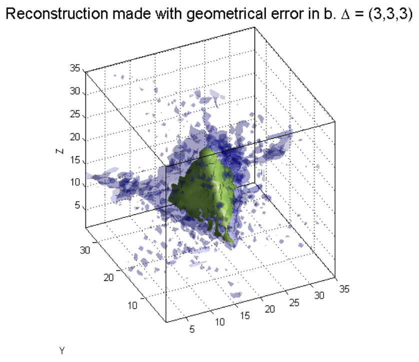

Reconstruction made with geometrical error in b. Δ = (3,3,3)
Z
Y
5
10
15
20
25
30
35

Figure 7.21: Illustrates a reconstruction made with Landweber where we have simulated drift in b with $\Delta = (3, 3, 3)$. Standard variables have been chosen with $noiselevel = 0.05$. The iteration shown is number 200.

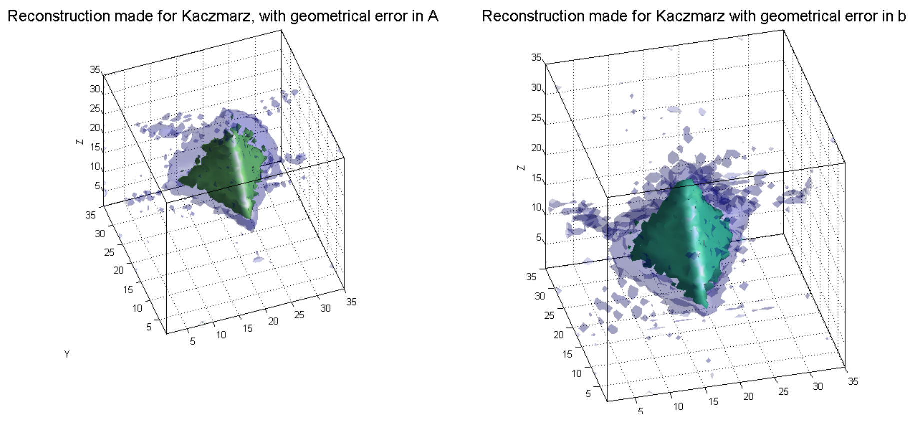

Reconstruction made for Kaczmarz, with geometrical error in A
Reconstruction made for Kaczmarz with geometrical error in b
Z
Y

Figure 7.22: Illustrates two reconstructions made with Kaczmarz where we have simulated drift in A for the left image and simulated drift in b for the right image. $\Delta = (3, 3, 3)$ and standard variables have been chosen with $noiselevel = 0.05$. The iteration shown is number 10.

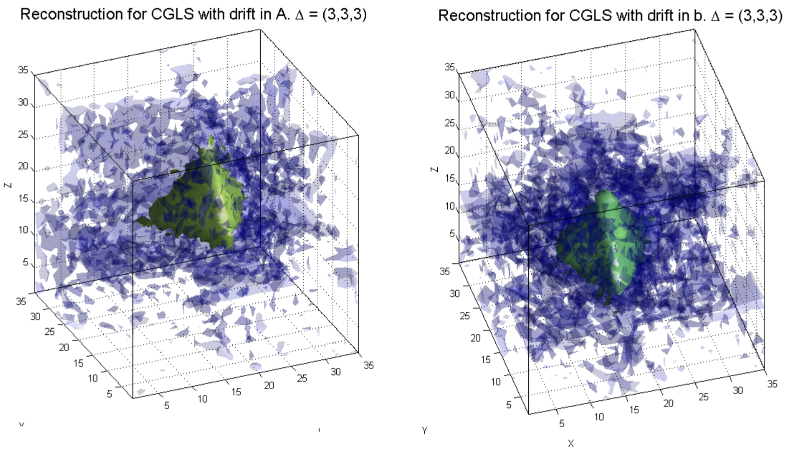

Reconstruction for CGLS with drift in A. Δ = (3,3,3)
Reconstruction for CGLS with drift in b. Δ = (3,3,3)
Z
Y
X

Figure 7.23: Illustrates two reconstructions made with CGLS where we have simulated drift in A for the left image and simulated drift in b for the right image. $\Delta = (3, 3, 3)$ and standard variables have been chosen with $noiselevel = 0.05$. The iteration shown is number 10.

Chapter 8

Stopping Criteria - For now

Until now we have only looked at reconstructions where the solution is known. In the real world the solution is unknown, and therefore we need new methods to check if our given solution x^k is good enough. Therefore we turn our interest towards the stopping criteria. The idea is that given certain stopping criteria and rules, our numerical iterative methods should be able to (hopefully) stop iterating when the solution is acceptable. This is what we will look upon in this chapter. There are two special cases of stopping rules that will be examined, namely the Discrepancy Principle (DP) and the Normalized Cumulative Periodogram (NCP).

8.1 The Discrepancy Principle

Let us take a look at the first criterion, namely the Discrepancy Principle from [2]. This is one of the most common stopping criteria for theoretical studies of regularization methods, due to the simplicity of the method. The idea is very simple, see 8.1

$$\text{Choose } k = k_{DP} \text{ such that } \|Ax_k - b\|_2 \geq \tau\|e\|_2 > \|Ax_{k+1} - b\|_2, \tag{8.1}$$

where τ is a "safety factor", and $\|e\|_2$ is the expected value of the perturbation norm. So the idea is that when the residual vector, $(Ax - b)$, equals the discrepancy in the data, $(\tau \|e\|_2)$ at a certain iteration k, we should stop iterating since this value should indicate the optimal reconstruction according to the stopping criterion.

Therefore we need to choose the parameter τ and determine $\|e\|_2$. The problem is how to determine $\|e\|_2$ though. In the real world we do only obtain the value b, where b corresponds to (2.10). The error term is unknown and it can be very difficult to determine its value or an estimate of it. For an example we refer to [7]. The idea behind the example was very simple. One should look at regions of pixels where the pixel value was more or less equal throughout the whole region. From here one found the standard deviation within the region, and this standard deviation was then the estimate of the error. It turned out to be problematic, since different values of the regions would result in different estimation values, even though it was the same error added to the test problem.

Therefore when we test this stopping criterion, we will instead assume that we know the norm of the error, and from here find the error. The norm of the error is found rather easily since this is what we add to our test problem, given from (2.12). This is done by `noise*norm(b_exact)*e` in MATLAB, and this value will be inserted as the estimate for $\|e\|_2$.

The last thing we need to do is to determine τ which is done by *training strategies.* We refer the reader to [6] for further information, but the function we will use for this is *trainDPME.m.* The call for this function is given as

$$tau = traindpme(A, b, x\_exact, method, type, delta, s).$$

Here A is our coefficient matrix, b is the noise-free right hand side, x_exact is the exact image, *method* specifies which method the user wants to use, *type* determines the stopping rule, s is the number of samples from which we find a reconstruction, and *delta* is the noise level we add to each sample s. The resulting value corresponds to τ, and then by multiplying with the above found value, we will end up with an estimate of $\tau\|e\|_2$ which will be used for the tests. The number of samples should be chosen such that s is around 15-20.

Before looking at reconstructions, let us first consider what we can expect to happen for this stopping criterion by watching figure 8.1. We know that at some point our residual vector (the blue line) will converge towards the norm of the error (the red dotted line). By multiplying our "safety factor" τ onto the estimation of the norm of the error, the stopping rule from (8.1) should stop iterating before reaching the flattening of the residual vector. Therefore a value of $\tau < 1$ would be bad to choose, since then we would never stop iterating. Why this is of importance will be shown in the following.

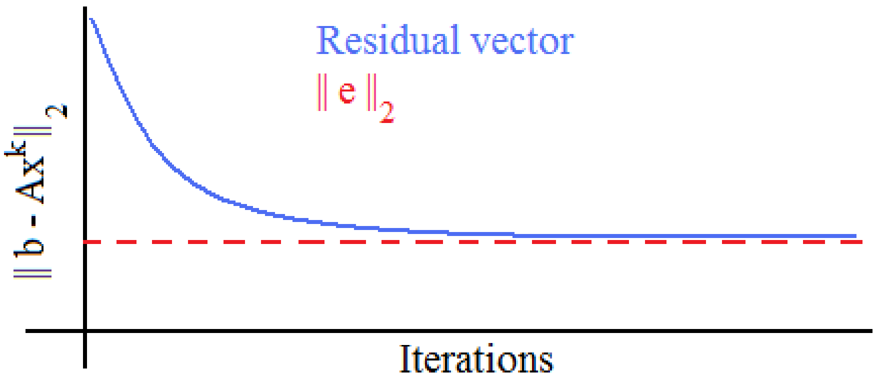


Figure 8.1: Illustrates how the DP stopping rule should works. The red dotted line illustrates the norm of the error. This in the real world will be hard to estimate. The blue line illustrates our residual vector which we know at some point converges to the norm of the error.

In MATLAB we have used the training algorithm mentioned above to determine all our τ values. The training algorithm was not implemented for the CGLS algorithm, therefore we have chosen this value manually, meaning that the value chosen resembles the values found for the other iterative methods. The values found can be seen in table 8.1. All these values of τ has been used to do reconstructions with, and the result was that MATLAB showed that 0 iterations were used for all methods. We know for a fact that it is very difficult to determine the error within an image, and on forehand this stopping criterion could therefore be bad to use. A further notice is that 4 of the methods use a value of $\tau < 1$ which indicates that when we use this value, we would never stop iterating, resulting in the stopping criterion k_{max}.

Table 8.1: Shows the τ value obtained by the training algorithm for every iterative method in 2D. All the problems were generated with $noiselevel = 0.05$, $N = 100$, $\theta = 0 : 1 : 179$, $p = round(\sqrt{2}N)$ and $s = 20$.

Method used	τ found by training
Kaczmarz	5.6826
Landweber	36.2676
SART	0.3369
CAV	0.0354
DROP	0.0383
Cimmino	0.0343

If we enter the value of τ manually, we are able for the Landweber and Kaczmarz method to stop iteration at some point. As an example if $\tau_{Kaczmarz} = 2$ we stopped iterating after 2 iterations, and if we chose $\tau_{landweber} = 1.2$ we stopped after 31 iterations. The two reconstructions can be found in figure 8.2. So how come that the last 4 iterative methods obtain a different τ value?

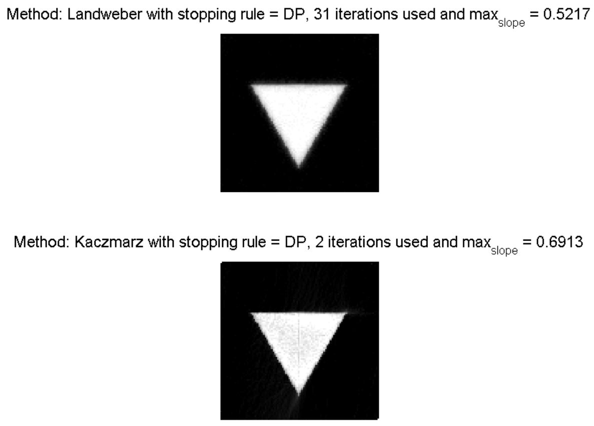

Method: Landweber with stopping rule = DP, 31 iterations used and max$_{slope}$ = 0.5217
Method: Kaczmarz with stopping rule = DP, 2 iterations used and max$_{slope}$ = 0.6913

Figure 8.2: Illustration of reconstructions done with the DP stopping criterion. τ has been entered manually for both pictures. Standard values of the variables have been used and $noiselevel = 0.05$. The top picture is done with Landweber and the bottom is done with Kaczmarz. As one notices, the reconstructions do not obtain a good measure of the sharpness even though the noise level is low.

To explain this we first recall that the DP stopping rule is defined as (8.1). Therefore the explanation must lie in the way we choose our residual vector. For the SIRT methods, the stopping rule has been implemented so that instead of having $\tau\delta$ as the stopping parameter, we now have $\tau\delta\|M^{\frac{1}{2}}\|_2$, where M is defined in Chapter 3. This implies that the residual vector for all the SIRT methods is defined as $\|M^{\frac{1}{2}}(b - Ax^k\|_2$, and apparently this M matrix yields low values for the residual vector. Therefore it is difficult to do reconstructions with this stopping criterion and also to choose an appropriate value of τ. The only reason that we can still obtain decent reconstructions with this stopping rule for Landweber's method, is because that $M = I$ for Landweber's method. For the CGLS method we have entered a value of $\tau = 2$ and from this the DP stopping criterion stopped iterating after 3 iterations.

The same happens for these methods in the 3D case and therefore the images are omitted, since this stopping criterion worked for 3 methods only. And that was by manually choosing the parameter.

8.2 Normalized Cumulative Periodogram

The next stopping criterion we will look upon is the so called Normalized Cumulative Periodogram (NCP) from [2] and [6]. The idea originated from Chapter 2 and 3, where we used the SVD and Picard plot to determine the truncation parameter k for the TSVD solution. The challenge is to see if there exists a possibility of choosing this parameter without computing the SVD or looking at the Picard plot.

The approach which we take is to view the residual vector ($r^k = Ax^k - b$) as a time series, and consider the exact right hand side b_{exact} as a signal that appears completely different from our noise vector e. Since b_{exact} represents a smooth signal we are able to do this, and now the idea is to find the value of the regularization parameter for which it applies that the residual vector changes from being signal-like to noise-like. With signal-like it is meant that the residual vector is dominated by components from b_{exact}, whereas noise-like means that the residual vector is dominated by the error components.

In the Discrepancy Principle we chose the parameter k when the norm of the residuals was equal to the norm of the noise. The principle here is almost the same, instead of using the norm of the noise as a stopping criterion, the stopping criterion depend on a statistical quantity that determines when the residual vector can be considered as noise. To do this we will use the discrete Fourier Transformation (DFT). Let $\hat{r}^k$ denote the DFT of the residual vector of an iterative method, then

$$\hat{r}^k = dft(r^k) = ((\hat{r})_1, (\hat{r})_2, \ldots, (\hat{r})_m)^T \in \mathbb{C}^m. \tag{8.2}$$

Hereby the power spectrum of r^k is defined as a real vector where

$$p^k = \left(|\hat{r})_1|^2, |\hat{r})_w|^2, \ldots, |\hat{r})_{q+1}|^2\right)^T, \quad q = [m/2]. \tag{8.3}$$

Here q denotes the largest integer such that $q \leq m/2$. Now we define the normalized cumulative periodogram for the residual vector r^k as the vector $c(r^k) \in \mathbb{R}^q$,

$$c(r^k)_i = \frac{(p^k)_2 + \ldots + (p^k)_{i+1}}{(p^k)_2 + \ldots + (p^k)_{q+1}}, \quad i = 1, \ldots, q. \tag{8.4}$$

If our residual vector consists of only white noise, then by definition the expected power spectrum is flat, implying that $E((p^k)_2) = E((p^k)_3) = \ldots = E((p^k)_{q+1})$, and all the points of our expected NCP curve will lie on a straight line from $(0, 0)$ to $(q, 1)$.

The last problem we must solve is how to determine that the NCP is within a straight line? There is a statistical method, which can determine with a significant level of 5% if the NCP lies within a straight line. It is if the NCP curve lies within the Kolmogorov-Smirnoff limits, which are equal to $\pm 1.35q^{-1/2}$. It turns out unfortunately that this cannot be achieved, because the real data do not necessarily have white noise to begin with, and because we actually extract some SVD components from the noise, indicating that the noise components that are left, are not totally white noise. Therefore we will choose our regularization parameter when the NCP is closest to a straight line, meaning that the residual vector resembles white noise the most. This is done by taking the 2-norm between the NCP and the vector $c_{white} = (1/q, 2/q, \ldots, 1)^T$, indicating that our new k_{NCP} will be chosen by minimizing 8.5

$$d(k) = \|c(r_k) - c_{white}\|_2. \tag{8.5}$$

Now that the theory for this stopping criterion is formulated we are ready to create our reconstructions. To do this let us consider tables 8.2, 8.3, 8.4 and 8.5. All these tables illustrate our 7 methods used for a 2D or 3D test problem with different noise added. The optimal solution has been found by our stopping criterion, and is compared to the optimal solution found by the relative error.

We begin with the first two tables, 8.2 and 8.3. These two tables are both made for a 2D test problem, but with different noise levels. We notice that the stopping criterion, functions very well and stops the iterations close to the exact solution. For $noiselevel = 0.05$ we find that all the SIRT methods work with a relative error approximately 1% larger than the best possible relative error. For the CGLS method it is 2% above the optimal solution found by the relative error, and for Kaczmarz it has found the best possible solution. So if our test problem is in 2D and the noise in the problem is low, NCP would be a very good stopping criterion. If we add noise with $noiselevel = 0.4$ in 2D we find a slight change in the trend. For the SIRT methods, and even for the CGLS method we are extremely close to the optimal solution, but for Kaczmarz we are slightly off the optimal solution. We can conclude that, the NCP stopping criterion works really well for 2D problems, whether noise is present or not.

With this in mind we consider the last two tables, 8.4 and 8.5. These tables are made for a 3D test problem where the noise added to the problem differs. Let us start with the table with $noiselevel = 0.05$. There is a slight change in the trend compared to the 2D example. The SIRT methods now find an optimal solution approximately 2% above the relative error. The same happens for Kaczmarz's method, but for CGLS we have almost found the best solution. If we increase the noise level to 0.4, all of the 7 methods find their optimal solution incredibly close to the solution found by the relative error. For the CGLS method we have found the optimal solution. It is seen from the tables that the SIRT methods used with the new stopping criterion, always stops after almost the same number of iterations. So also for the 3D case, we can conclude that the NCP stopping criterion works really well.

Table 8.2: Shows the optimal iteration found by applying the NCP stopping criterion for the different methods. The reconstructions are made in 2D. All the problems were generated with $noiselevel = 0.05$, $N = 100$, $\theta = 0 : 1 : 179$ and $p = round(\sqrt{2}N)$.

Method used	Iteration stopped	$\frac{\|x^{k^{stop}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$	$\frac{\|x^{k^{opt}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$
Kaczmarz	5	10.51	10.51
Landweber	86	8.26	7.50
SART	98	8.30	7.53
CAV	99	8.41	7.74
DROP	97	8.92	7.86
Cimmino	97	8.46	7.66
CGLS	5	16.15	14.62

Table 8.3: Shows the optimal iteration found by applying the NCP stopping criterion for the different methods. The reconstructions are made in 2D. All the problems were generated with $noiselevel = 0.4$, $N = 100$, $\theta = 0 : 1 : 179$ and $p = round(\sqrt{2}N)$.

Method used	Iteration stopped	$\frac{\|x^{k^{stop}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$	$\frac{\|x^{k^{opt}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$
Kaczmarz	3	63.12	58.68
Landweber	35	29.23	27.24
SART	39	28.32	27.64
CAV	39	29.43	28.63
DROP	39	29.99	29.07
Cimmino	39	29.73	27.15
CGLS	3	38.01	37.59

In the earlier sections we saw two different methods that could be used as stopping rules for our iterative methods. In this and in the following sections we will discuss these other stopping rules and if it would be beneficial to include

Table 8.4: Shows the optimal iteration found by applying the NCP stopping criterion for the different methods. The reconstructions are made in 3D. All the problems were generated with *noiselevel* $= 0.05$, *r1_max* $= 17$, *degree* $= 38$ and *u_max* $= 23$.

Method used	Iteration stopped	$\frac{\|x^{k^{stop}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$	$\frac{\|x^{k^{opt}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$
Kaczmarz	8	10.47	8.69
Landweber	94	8.06	6.36
SART	86	8.97	6.91
CAV	84	9.02	7.22
DROP	81	10.22	7.30
Cimmino	85	9.18	7.12
CGLS	7	29.42	29.19

them.

8.3 New Stopping Rules - Some Ideas

Before we formulate new ideas for new stopping rules, we need to see what we can learn from the regular grain. We know from our test case that our true image consists of only zeroes and ones, which indicates that a possible solution should also only consist of zeroes and ones. The principle in the idea is then

$$x^k = x^{k,bin} \quad \text{for all } k = 1, 2, \ldots, k_{max}.$$

By doing this all our iterations would only consist of zeroes and ones, and hopefully we would then find a good solution. This is called "Integer Programming" and if it should be done in MATLAB, we would need new algorithms. It is not a part of this project, and therefore it will not be considered. It should just be mentioned that for this idea to be successful, methods do exist that can compute such a solution. On the downside it is known that these types of problems take a lot of time to compute, and since this is not appropriate, we therefore need to look at other ideas for stopping rules.

The next idea we will try is the idea that has been implemented, but we return to that later. The idea originated from the fact that we wanted to exploit certain properties in our test problem. Therefore a list of what we know for certain in the reconstruction was made:

Table 8.5: Shows the optimal iteration found by applying the NCP stopping criterion for the different methods. The reconstructions are made in 3D. All the problems were generated with $noiselevel = 0.4$, $r1\_max = 17$, $degree = 38$ and $u\_max = 23$.

Method used	Iteration stopped	$\frac{\|x^{k^{stop}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$	$\frac{\|x^{k^{opt}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$
Kaczmarz	4	49.47	48.73
Landweber	33	31.79	31.71
SART	35	36.09	36.04
CAV	35	37.31	37.28
DROP	36	38.12	38.12
Cimmino	35	37.22	37.19
CGLS	3	59.22	59.22

- 1. In our solution we will have a figure in 2D with equal side length or a polyeder in 3D located somewhere. We do not know how big this figure is or where it is placed.
- 2. The image as a whole will consist of two domains, a figure in 2D or 3D and a background.
- 3. The pixel/voxel value of the figure is not necessarily known but we know that this value is constant inside the whole figure. For the test problem, we know that the value of the figures corresponds to 1.
- 4. The pixel/voxel value for the background on the other hand is known, and we know that in a perfect world the value should be equal to 0.

It is exactly item number 4 from 8.3 that we wish to exploit for this type of stopping rule. The idea is now that if we can somehow locate the background in our solution for each iteration, then in this way we can also locate the position of the grain. This is what we intend to do, with special interest in locating the background, and the help we need comes from Digital Image Processing.

8.4 New Stopping Rules - Implementation

We now turn our attention to the new idea for a stopping rule, namely to locate the background within our solutions. From [1] we have a method of doing this, and in MATLAB it is done by use of 3 functions, namely *bwlabel.m, bwmorph.m* and *regiongrow.m.* We refer the reader to [1] for more information about the algorithms, and turn our attention towards the last function, *regiongrow*, since this is the function we will use. The function groups pixels into regions, given certain criteria, and this is exactly what we can exploit.

The call for the function is

$$[\text{g, NR, SI, TI}] = \textit{regiongrow}(\text{f, S, T}),$$

where the three output parameters NR, SI and TI are not interesting. The input variable f corresponds to the image, of which we wish to find the regions. In this case this corresponds to the given iteration x^k. S is a matrix consisting of seed points. We will not go into detail with how the function works, but the idea is that these seed points correspond to starting values, and from these points, the function uses the neighboring pixels to check wether these pixels should be connected in a region with the seed point or not. S is a matrix of $N \times N$, and is binary. Where there is a 1 in S, the corresponding pixel in f is a seed point, indicating that if $S_{ij} = 1$ then $f_{ij} =$ a seed point , where $i, j = 1, 2, \ldots, N$. An option to S is to enter a scalar value instead of a matrix, e.g. 0.1, and then all pixels in f which are equal to this value will be set as a seed point. The last input variable T is a threshold number. If T is a matrix, then the value at T_{ij} determines the threshold for the pixel at f_{ij}. This threshold value is used to determine if a point should be connected to the same region as a seed point via 8-connectivity or not. For more on 8-connectivity we refer to [1]. T can also be a scalar, indicating that it will become a global threshold. In our case we will stick to a single value of T, namely the global threshold for our pixels. For the next tests we will use a threshold value equal to $T = 0.5$ and all seed points will be defined as $S(x^k < 0.05) = 1$. This means that every pixel that has a value lower than 0.05 in our solution x^k will be set to a seed point.

Now we are ready to test the idea in praxis. The idea is that for every iteration k, we will locate the background. We know that the background should be equal to 0, so when we have located the background, we take the sum of these pixels. The closer we are to 0 the better. One might ask why we cannot locate the whole grain instead of locating the background? The reason is very simple. We do not know how big the size of our grain is, and therefore we do not know what "the sum" of our grain is. But the background should always have a sum equal to 0. So, now that we have found the sum of our background, which is defined as a vector called $sum_{background}$, the criterion for when we should stop iterate is

$$sum_{background}(x^k) < sum_{background}(x^{k+1}). \tag{8.6}$$

Therefore when we have found $sum_{background}$ for all iterations k, we take the difference of this vector, by the function *diff* in MATLAB. If this difference is positive, we know that (8.6) is obtained and the iteration stops.

For the 3D case there is a problem regarding this. The function *bwmorph* does not support 3D images and it seems that there does not exist a program yet that can do the same in 3D as *bwmorph* does in 2D. Therefore when we test this stopping criterion for 3D we have done the following. We will keep the z-axis constant, and for every z-value, we find the corresponding xy-plane. This plane is in principle a 2D figure, and from this plane we calculate the sum of the background, just like we did for the 2D case. So instead of calculating the sum of the background for 1 iteration in 2D, we need to do it N times per iteration for 3D. The code can be found in Appendix D.

This turns out to be problematic, and let us illustrate this with an example. We observe figure 8.3. As we can see no minimum has been located, resulting in the fact that our method would use k_{max} as a stopping criterion instead of our newly found stopping rule. The reason is that we have iterated too much, and it seems that with the stopping rule, we might converge too slowly towards our solution. Another problem is that the iteration numbers 1 and 2 seem to have a low value of $sum_{background}$, indicating that according to our newly found stopping rule, we should stop at these iterations. For now we will disregard the first two iterations, and assume that these are used to get our iterative methods starting. The optimal solution was found to be equal to $k^{opt} = 195$ according to the relative error, which was made for a Landweber test problem with standard variables. This indicates that the minimum of the relative error was around $7 - 8\%$.

Therefore we need to add another restriction for our stopping rule, in addition to skipping the first few iterations. We get the idea from an observation of the top figure in 8.3. As we can see from the graph, it drops fast around the first 100 iterations, and hereafter it wears off. Therefore the idea will be to look at a set of iterations, and if the difference between two consecutive iterations is below a certain percentage, then the iterations stop. We define a new term $rel_{e,b}(i)$, where $i = 1, \ldots, k_{max}$, as the absolute relative error change in the background per iteration, indicating

$$rel_{e,b}(i) = \left| \frac{sum(x^{k+1}_{background}) - sum(x^{k}_{background})}{sum(x^{k}_{background})} \right| \cdot 100, \quad i = 1, \ldots, k-1.$$

This means that we take the difference between the k'th and the k+1'th iteration, and divide by the sum of the background at the k'th iteration. For an example see below. We assume that we have found our $sum_{background}$ and wish to determine $rel_{e,b}$. We have

$$\begin{aligned} sum_{background} &= (8, 5, 2, 1), \quad \text{indicating that} \\ difference &= (-3, -3, -1). \end{aligned}$$

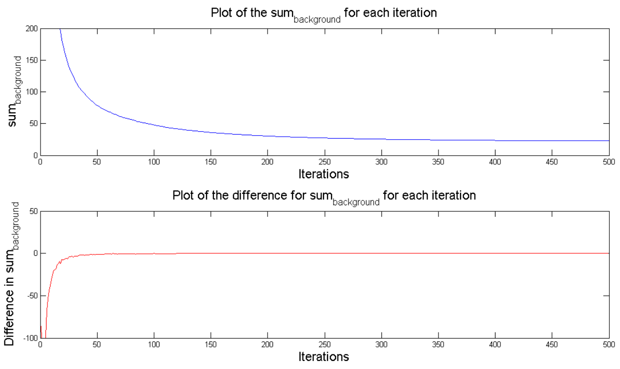


Figure 8.3: Shows the sum of the background region, and the difference in this vector. The noise added equals $noiselevel = 0.05$, with standard values for the 2D test problem. The method used is Cimmino. The top figure shows that no minimum is found, since the background gets closer and closer to 0. The bottom figure shows the difference, and as we can see none values are above 0.

From this the relative error in the background per iteration would be defined as

$$rel_{e,b} = \left(\left| \frac{-3}{8} \right|, \left| \frac{-3}{5} \right|, \left| \frac{-1}{2} \right| \right).$$

From this example, we see that from iteration 3 to iteration 4, $rel_{e,b}$ drops with 50%. By implementing the new criterion we end up with figure 8.4 for the test problem above, where $k^{opt} = 195$. From this criterion we discover another problem. If we were to choose a value of $rel_{e,b}$, it seems appropriate to choose a value between 0.05% and 1%, since between these values the graph has reached its plateau and is flattened out. If we were to choose one of these values we would end up with a solution corresponding to $k^{opt} \simeq 60$, due to the high spikes on the graph.

Therefore we introduce the final addition to the stopping rule, which shall ensure that these spikes do not influence the stopping criterion. The idea is that instead of finding $rel_{e,b}$ for one iteration only, we find the mean of this value spread over multiple iterations, as a running mean. The new value is defined in the following way:

$$rel_{e,b}^{new}(i) = \frac{rel_{e,b}^{new}(i-2) + \ldots + rel_{e,b}^{new}(i+2)}{5}, \quad i = 3, 4, \ldots, k-2.$$

By doing this we obtain figure 8.5. In this picture the large spikes seem to have disappeared, indicating that we could find an appropriate value of $rel_{e,b}$. On

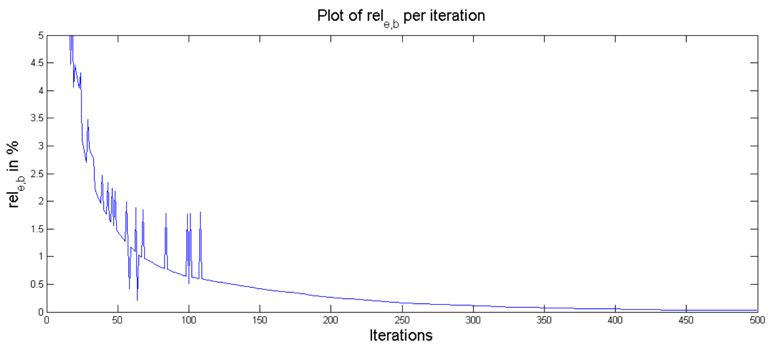


Figure 8.4: Plot that shows the relative error change in the background for one iteration. $noiselevel = 0.05$, with standard values for the 2D test problem and the method used is Cimmino. As we can see there are some random spikes on the graph, which we need to remove or flatten out.

figure 8.6 all the reconstructions has been made. We notice that the stopping criterion for the top right figure is still a bit blurry, which makes perfect sense, since we stopped at 60 iterations. Earlier we saw that by stopping to soon blurry reconstructions will result. The last 3 reconstructions seems reasonable, the only comment is that the lower left figure has used a lot of iterations, indicating that the figure took much more time to compute.

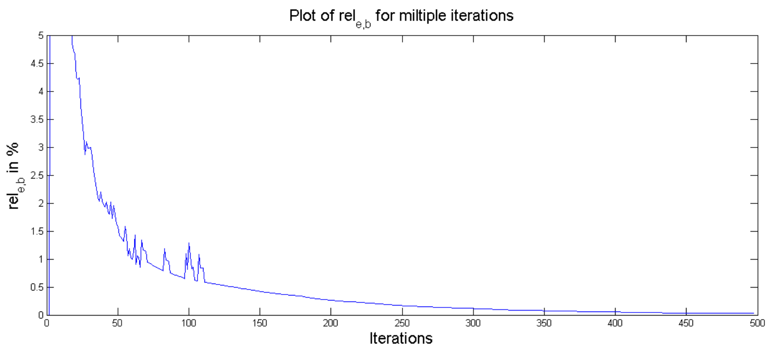


Figure 8.5: Plot that shows the relative error change in the background for multiple iterations. $noiselevel = 0.05$, with standard values for the 2D test problem and the method used is Cimmino. The spikes have more or less disappeared which indicates that it is easier to choose an exact value of $rel_{e,b}$.

With this we turn the interest towards the reconstructions obtained by the new stopping criterion. We therefore consider the tables 8.6, 8.7, 8.8 and 8.9. We

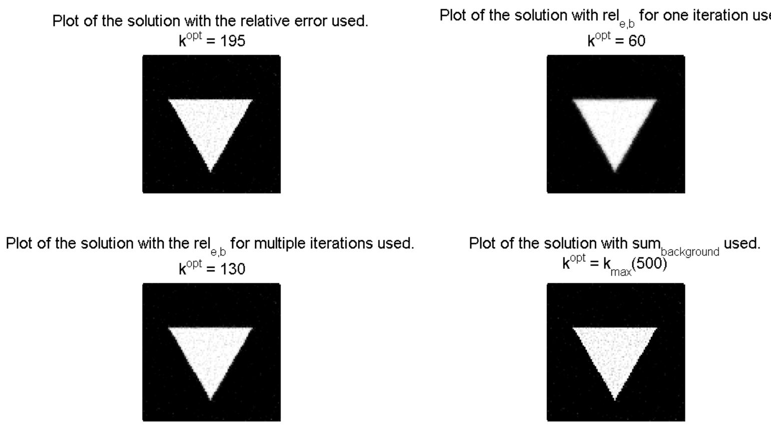

Plot of the solution with the relative error used.
$k^{opt} = 195$
Plot of the solution with $rel_{e,b}$ for one iteration used.
$k^{opt} = 60$
Plot of the solution with the $rel_{e,b}$ for multiple iterations used.
$k^{opt} = 130$
Plot of the solution with $sum_{background}$ used.
$k^{opt} = k_{max}(500)$

Figure 8.6: Figure that shows the 4 reconstructions obtained. The top left is found by the relative error of the 1-norm. The top right figure is found by using $rel_{e,b} = 0.5$ for one iteration only as a stopping rule. The lower left figure is found by using $rel_{e,b} = 0.5$ for multiple iterations as a stopping rule, and the lower right figure is found by using the $sum_{background}$ as a stopping rule.

start with the table 8.6. This compares the results of applying our new stopping criterion to the 7 different methods. The noise added to the problem is 0.05, and the test problem is 2D. We see that 6 of the methods, all the SIRT and Kaczmarz yield solutions extremely close to the optimal solution found by the relative error. The only method that fell out of category was the CGLS method, and the reconstruction found for this method with the new stopping criterion is very poor. To investigate the sensitivity towards noise, we consider table 8.7, which is the same test problem, with the exception that $noiselevel = 0.4$. We see from this table that the SIRT methods find a decent solution, not many percent away from the exact. On the other hand the Kaczmarz solution obtained is of poor quality, and the same goes for the CGLS method. So it seems that this new stopping criterion works well for the SIRT methods in 2D, independent of the noise added. Also it works well for Kaczmarz, if the noise present is low. For the CGLS method we do not seem to be able to stop with an acceptable reconstruction for the 2D case.

Let us consider the 3D case, and first we consider table 8.8. The table illustrates the 7 different methods and the solutions obtained by our new stopping criterion. The noise added is 0.05, and the test problem is in 3D. The trend is similar to the 2D example, the SIRT methods find a decent reconstruction by $1 - 2\%$ off the solution found by the relative error criterion. The same applies for Kaczmarz. Again the solution found for the CGLS method is poor. In table 8.9 the noise level is increased to 0.4. From this we see that the SIRT methods all have found a very good solution, and in some of the cases they have found the optimal solution. Kaczmarz method has found a solution only off by 3% compared to the relative error and the reconstruction found for CGLS is still poor.

We may conclude that, the new stopping criterion works well for all the SIRT methods. For both 2D and 3D, and even if noise is present, we obtain a very good reconstruction. For the Kaczmarz method we get mixed results. If much noise is present in 2D, we get a poor reconstruction. For the other 3 cases, we obtain a decent reconstruction for Kaczmarz's method. For the CGLS method the stopping criterion should not be used, since the reconstructions were of too poor quality. As a note, for this stopping criterion, the SIRT methods uses the same amount of iterations compared to each other. This was also the case when the NCP stopping criterion was used.

Table 8.6: Shows the optimal iteration found by applying the new stopping criterion for the different methods. The reconstructions are made in 2D. All the problems were generated with $noiselevel = 0.05$, $N = 100$, $\theta = 0 : 1 : 179$ and $p = round(\sqrt{2}N)$.

Method used	Iteration stopped	$\frac{\|x^{k^{stop}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$	$\frac{\|x^{k^{opt}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$
Kaczmarz	9	10.94	10.51
Landweber	131	7.66	7.50
SART	132	7.82	7.53
CAV	134	8.02	7.74
DROP	134	8.27	7.86
Cimmino	134	7.94	7.66
CGLS	57	52.15	14.62

The question is now why the CGLS method gave such poor results. From the tables one could think that due to the stopping criterion, the CGLS method stops iterating to late. And the reason for this might be the following. Recall figure 3.3 from Chapter 3. The figure illustrates the semi-convergence plot for the CGLS method. What we see on this graph is that the method very quickly, independent of the noise, find the optimal reconstruction and then diverges. Therefore it might happen, that when finding the optimal solution, the error within the background never reaches the criterion for where the method should stop iterating. In this case it was when $rel_{e,b}(i) < 0.5$. This is exactly the case if we consider figure 8.7. As we can see from this example, the constant c which

Table 8.7: Shows the optimal iteration found by applying the new stopping criterion for the different methods. The reconstructions are made in 2D. All the problems were generated with *noiselevel* $= 0.4$, $N = 100$, $\theta = 0:1:179$ and $p = round(\sqrt{2}N)$.

Method used	Iteration stopped	$\frac{\|x^{k^{stop}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$	$\frac{\|x^{k^{opt}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$
Kaczmarz	15	80.71	58.68
Landweber	35	27.58	27.24
SART	42	28.89	27.64
CAV	53	29.54	28.63
DROP	57	32.46	29.07
Cimmino	53	31.71	27.15
CGLS	40	> 100	37.59

will be explained in the next section, which we wanted to stop at, is first reached after a lot of iterations, even though the optimal solution for CGLS is found fast. In fact we hit it at iteration 1 or 2, but as mentioned earlier we do not stop at the two first iterations. Therefore for the CGLS method, we need to do something different for the new stopping criterion to work.

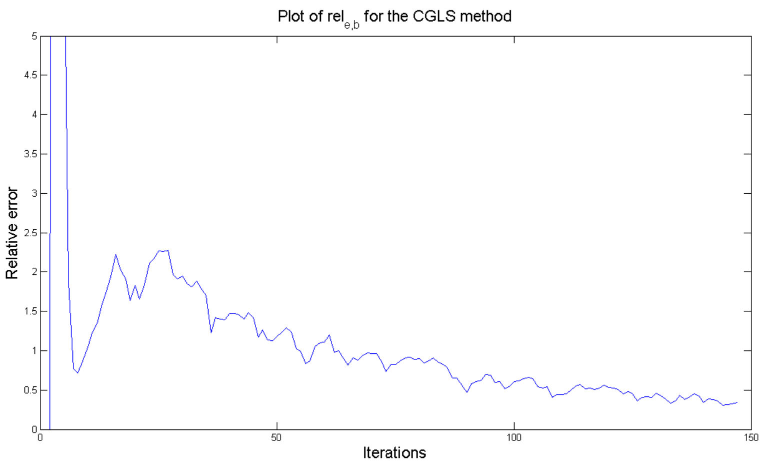


Figure 8.7: Plot of $rel_{e,b}$ for the CGLS method with *noiselevel* $= 0.05$. The test problem is in 3D with standard variables. As one notice, we first hit the value 0.5 around 90 iterations, and we know at this point that the reconstruction will be ruined.

Table 8.8: Shows the optimal iteration found by applying the new stopping criterion for the different methods. The reconstructions are made in 3D. All the problems were generated with *noiselevel* = 0.05, *r1_max* = 17, *degree* = 38 and *u_max* = 23.

Method used	Iteration stopped	$\frac{\|x^{k^{stop}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$	$\frac{\|x^{k^{opt}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$
Kaczmarz	23	8.74	8.69
Landweber	95	8.03	6.36
SART	89	8.87	6.91
CAV	88	9.31	7.22
DROP	90	9.80	7.30
Cimmino	88	9.07	7.12
CGLS	103	81.54	29.19

Recall for this stopping criterion that the $sum_{background}$ was not enough as a stopping criterion for the SIRT methods. What we will show here is that this simple constraint is enough to obtain a good measure for the CGLS method. We consider figure 8.8. As we can see from this constraint, we have found a new minimum. The minimum is obtained at $k = 7$, and the relative error at this value is 29.44%. Recall that the optimal value was found at 29.19% and therefore we are very close to the optimal solution. So in conclusion, the new stopping criterion works good for the SIRT methods and the Kaczmarz method, but for the CGLS method we have to use the simplified version to obtain a good reconstruction. This simplification was not enough for the SIRT methods or the Kaczmarz method.

In these examples a constant value of $rel_{e,b}$ was chosen, along with a constant value for S and T. In the next sections we will discuss the proper choice of these parameters.

8.4.1 Determining Values for our Stopping Rule

The idea is to investigate if there is a value for which we can choose $rel_{e,b} = c$, where c indicates a scalar. If we can choose a standard value for when we should stop iterating, independent of the noise added to the problem, it would be perfect. Therefore we observe figure 8.9. On this figure different noise levels has been compared with each other, and it seems that if we should choose a value that should be obtained for every noise level, a value of $c = 0.5$ seems appropriate. This is due to the fact that when the curves on the plot are starting

Table 8.9: Shows the optimal iteration found by applying the new stopping criterion for the different methods. The reconstructions are made in 3D. All the problems were generated with $noiselevel = 0.4$, $r1\_max = 17$, $degree = 38$ and $u\_max = 23$.

Method used	Iteration stopped	$\frac{\|x^{k^{stop}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$	$\frac{\|x^{k^{opt}} - x^{exact}\|_1}{\|x^{exact}\|_1} \cdot 100$
Kaczmarz	6	51.45	48.73
Landweber	29	31.71	31.71
SART	34	36.07	36.04
CAV	32	37.28	37.28
DROP	34	38.14	38.12
Cimmino	34	37.20	37.19
CGLS	20	> 100	59.22

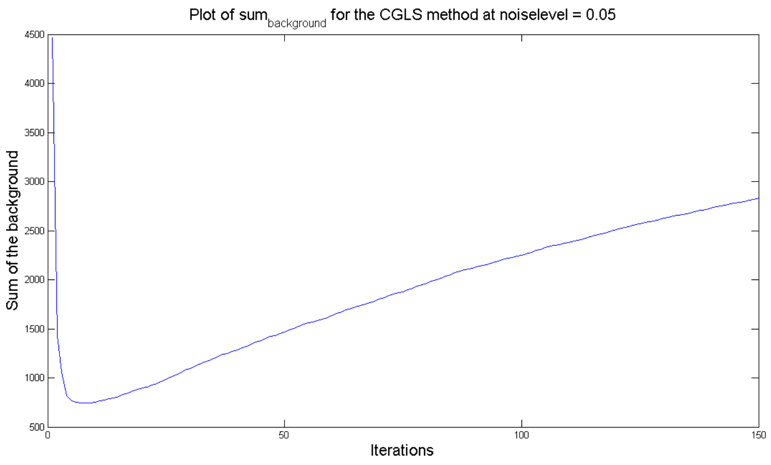


Figure 8.8: Plot of $sum_{background}$ for the CGLS method with $noiselevel = 0.05$. The test problem is in 3D with standard variables. As we see we have a minimum, and this is found at $k = 7$.

to flatten out, it corresponds approximately to the value 0.5. One could argue that in the real world, we almost never experience a noise level equal to 40% so why bother setting the criterion that low. It is just a precaution made to be on the safe side, since for a real world problem we do not know how big the errors in the problem are.

As for the other variables we have decided that values below 0.05 in each iter-

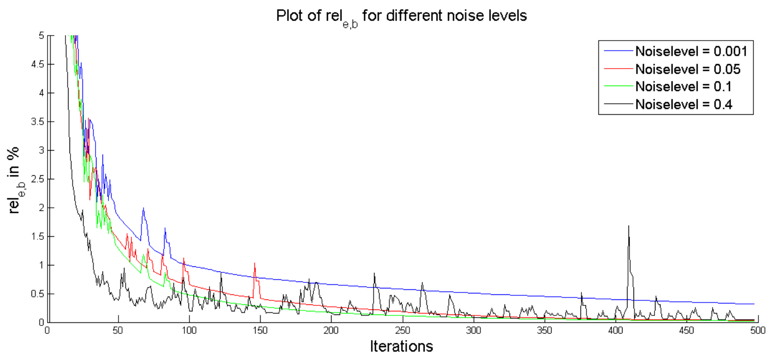


Figure 8.9: Figure that shows the 4 reconstructions obtained. The top left is found by the relative error of the 1-norm. The top right figure is found by using $rel_{e,b} = 0.5$ for one iteration only as stopping rule. The lower left figure is found by using $rel_{e,b} = 0.5$ for multiple iterations as stopping rule, and the lower right figure is found by using the $sum_{background}$ as a stopping rule.

ation should be set as background. This means that all the voxels which are below 0.05 in an iteration will be set to a seed point. For the threshold we have decided to cut at $T = 0.5$. Even if much noise is present, a value above 0.5 should not be set as a background. And also, values above 0.5 would probably be a piece of the grain, even when the reconstruction is dominated by noise. These variables are also what we used for the above test problems.

8.5 Summary

In this chapter we have considered which stopping criteria we have available, and seen how these works on a given test problem. It turned out that the discrepancy principle had problems, both for finding τ, and when this was known, it was difficult to obtain a decent reconstruction. The NCP showed to be very good, both for 2D problems as well as 3D problems, and also for problems which had a lot of noise added.

We wanted to exploit our knowledge about the grain to come up with a new stopping criterion, and therefore we saw some ideas for how this could be achieved. One special idea turned out to be of interest which has been implemented and tested. This idea was regarding Digital Image Processing. The criterion turned out to work for all the SIRT methods, and for Kaczmarz in some extend, but the same did not apply for the CGLS method. At one point the new stopping criterion was to ”simple” and therefore kept on iterating for the SIRT methods. This simplicity turned out to work perfectly for the CGLS method, since we were able to obtain a good reconstruction by this.

Chapter 9

Summary and Conclusion

The objective of this thesis is to test different methods on the basis of specific chosen tests and compare them to each other. Furthermore an analysis of old stopping criteria should be made, and compare them to an analysis of a new stopping criterion and explain the ideas behind this criterion. Also different measurement methods should be developed and tested so we could distinguish a good reconstruction from a poor. This we believe is achieved.

One of the first tests we made, was to observe if the iterative methods all fulfilled the semi-convergence criterion. We found that this was the case, which would ensure that our iterative methods would converge towards a solution.

It was explained how a grain is created and simple tests of the grain were set up and investigated. We saw that it did not matter which value of N we would choose. If the other variables were the same, we could expect to obtain the same relative error. The choice of w was important on the other hand. If we did not choose this carefully our reconstructions would appear blurry. Therefore we chose the standard value of $w = \sqrt{2}N$ which gave good reconstructions.

In the thesis we saw that the non-negativity constraint should always be applied if possible. In this way we would obtain the best reconstructions. There was no big difference between using the 1-norm or the 2-norm to calculate the relative error, therefore we decided to make use of the 1-norm. A different guess of the starting vector could result in a slightly sharper image measured by the max_{slope} value, but this was at the expense of a higher relative error. Therefore we decided to make use of the 0 vector as a starting vector for all our iterative methods. A new measurement method was introduced to measure the sharpness within a reconstruction. For 2D this turned out to be a decent measure because when a reconstruction appeared blurred visually, the value of our slope was lower. Furthermore we showed that the value was not chosen completely at random. For the 3D problems the value differed from the 2D case. It was shown that for a test problem with much noise, the edge would still be sharp, and therefore our max_{slope} value would appear sharp for every 3D test problem. So it seems that when doing reconstructions in 3D, the iterative methods can keep the edges sharp.

Next we investigated how A was created. This matrix takes a long time to compute, and we wanted to find the lowest possible values of p and θ without ruining our reconstruction. Depending on the desired precision of the reconstructions, one could find in the tables which combination of variables that would result in an acceptable relative error value. Also extreme values were found for both p and θ. As we saw in 2D, our values should be chosen so that the minimum value of p was $p = 0.8N$, and the minimum value of θ was $\theta = 0 : 30 : 179$. For the 3D case these values differed. It was seen that for the *degree* value, we needed at least $degree = 14$ different direction vectors. Since we obtained these direction vectors with help of *getLebedevSphere*, it could happen that $degree = 14$ was too large. But this function could only simulate for $degree = 6$ and $degree = 14$, which resulted in the minimum 14. For *u_max* which corresponds to p in 2D, we saw that at least $u\_max = 0.5 \cdot r1\_max$ was needed in order to obtain a decent reconstruction.

An unfortunate feature is that the test problems are given in a specific way. Therefore we could for some reconstructions obtain large values of the pixels and/or voxels, which especially for the 3D case could dominate the relative error. We saw for the SIRT methods that Landweber could find a relative error approximately by a factor 2 better than Cimmino. In 2D the dominance by the large values was insignificant. In the real world, if a material physicist had this problem, some precautions would be made. For instance, one would define a certain threshold, and if $\|a^i\|_2^2 < threshold$, this row would be removed. In this way we would never experience these large pixel/voxel values. Instead we had to choose a proper combination of *r1_max* and *u_max* to avoid these cases.

Then the geometrical error was simulated and explained for both A and b. There were two types of geometrical errors, one with constant translation per angle and the other with varying translation per angle. We saw in the case for constant translation per angle, that a simulation of a geometric error in A would move our reconstruction in a specific direction. If we simulated a geometric error in b our reconstruction would move in the opposite direction. As we saw, the iterative methods could account for this type of phenomenon, and by translating the exact image the same amount for which we simulated the geometric error, we could still make use of the relative error as a measure.

A problem started when we simulated the other type of geometric error, drifting. We saw that when drifting was simulated in A, the more we added of Δ, the larger the reconstruction would appear. Also artefact tend to happen around the corners of our reconstruction. If we simulated the same for b, the reconstruction would get smaller the more we increased Δ. Also artefact popped up here, but they were located around the edges instead of around the corners. All our iterative methods had difficulties doing reconstructions when this phenomenon occurs. Furthermore we could not use the relative error, since we did not know the exact location of our reconstruction.

We considered two stopping criteria, namely the discrepancy principle (DP) and the normalized cumulative periodogram (NCP). For the DP stopping criterion we encountered problems when we wanted to determine the value τ. Also the residual vectors for some of the iterative methods were found in a different way, indicating that we could not make use of this stopping criterion. This stopping criterion only worked for some methods, and did this even poorly. Therefore we would not recommend to use it. The NCP on the other hand worked flawlessly. Both for 2D and 3D test problem, the NCP stopping criterion found solutions close to the optimal solution determined by the relative error. Even when much noise was present in our problems, the NCP still worked.

This lead us to the implementation of a new stopping criterion. We wanted to exploit certain features about our grain for this criterion, and the first idea was to look at the sum of the background. The background would be recognized by Digital Image Processing. This "simple" idea seemed to be insufficient, since for the SIRT methods and the Kaczmarz method, we would not stop iterating before we had reached k_{max}. A different way of stopping with application of the background was introduced. For this criterion the relative error in the change of the background was used. The idea was to consider two iterations, x^k and x^{k+1}. Their difference in the sum of the background would be divided by the total sum of the background for x^k. This turned out to be problematic, since "spikes" appeared in our plots, indicating that we would stop too soon. Therefore we took a mean of 5 iterations instead of considering 1 iteration only. This turned out to be perfect for the SIRT methods and almost perfect for the Kaczmarz method. Even if much noise was present we would still find a good solution for the SIRT method. Only for the 2D case with much noise, did we not find a good solution for Kaczmarz.

We saw for the ART methods that Randomized Kaczmarz performed poorly compared to Symmetric Kaczmarz and the usual Kaczmarz. Symmetric Kaczmarz performed as well as the usual Kaczmarz, but it was shown that Symmetric Kaczmarz used twice the computation time per iteration as the normal Kaczmarz. Therefore the normal Kaczmarz was representative for the ART methods. A unfortunate feature about the Kaczmarz method was that artefact was generated, and the more noise we had present, the more artefact appeared. This of course resulted in poor values of the relative error. The same trend happened for the CGLS method, the method was very sensitive towards noise, and when much noise was present, the reconstruction was also noisy. The SIRT methods were better at handling noise in the objects.

We saw that Kaczmarz and CGLS used fewer iterations than the SIRT methods in order to find its optimal solution regardless of the error. In 2D the relative error found was larger than the relative error found for the SIRT methods. On the other hand, Kaczmarz seemed to obtain the same relative error as the SIRT methods for the 3D test cases, when little noise was present. The CGLS method generated solutions where the relative error was much larger than for the Kaczmarz method and the SIRT methods, regardless of the noise and the dimension of the problems.

The phenomenon "drifting" was reconstructed poorly by all methods.

The stopping criterion NCP, was shown to be very good for all our iterative methods, regardless of the noise and the dimension of our problem. This stopping criterion gave a decent result every time. The new stopping criterion found in this thesis was shown to be good for all the SIRT methods regardless of the noise and the dimension of the problem. For Kaczmarz's method in 2D with much noise added, we had difficulties with the determination of a good solution. For the CGLS method the simplified version of this stopping criterion was shown to work well, whereas the other methods needed additional constraints from this stopping criterion in order to stop at a reasonable iteration number.

Therefore on the basis of the results, my recommendation will be: Generally I would use the NCP criterion, due to the fact that this worked well for every test problem. Furthermore in general, if our test problems are given in a specific way, where e.g. a material physicist have not made use of $\|a^i\|_2^2 < threshold$, then I would choose to use Landweber's method to ensure that no high pixel or voxel values can occur.

If I was in a hurry and needed a fast solution, I would always prefer the Kaczmarz method since this method uses less iterations than the SIRT methods and provides a solution with smaller relative error than the CGLS method.

9.1 Future Work

In this section we will discuss possible future work emerging from the results of this thesis.

When the new stopping criterion was implemented for the 3D test problems, we needed to make a special version, since there does not exist a 3D version of *bwmorph*. As mentioned earlier, when we had a test problem in 3D we needed to run the code N times for a 3D case. Since we can locate the grain and the background, it could be implemented in such a way that we only do the calculations for those N values where we have located our grain. This implies that e.g. if $N = 30$ and our stopping criterion finds out the grain is located between $N = 10$ and $N = 20$, we would only need to run through $N \in [8\ ;\ 22]$. This would save some computation time.

With this new stopping criterion we also have the possibility of extending this stopping criterion to multiple grains. When multiple grains are present, we will have an indicator variable, that shows which grain we work on. The special feature of this indicator variable is that when it is 1 for a certain grain, it is 0 for all other grains. Therefore we would be able to do reconstructions for multiple grains, since only 1 grain would be considered at a time, while the rest would be set as a background.

Even though the DP stopping criterion seemed bad, one might implement it in such a way that it meets the requirements for use of the residual vector $r^k = \|M^{\frac{1}{2}}(b - Ax^k\|_2$. It is possible that the new DP stopping criterion can produce good reconstructions, but it is doubtful.

Appendix A

Maple sheet

Restarting the sytem with the correct packages.
> restart; with(linalg); with(geom3d);
> help(”greatdodecahedron”);
Using the correct command to create a regular polyhedron.
> RegularPolyhedron(i, [3, 5], point(o, 0, 0, 0), 1);
> form(i);
Creating the points that the above command gave. Below is an example

$$vec1 := \left[0, \frac{1}{5} \cdot 5^{\frac{3}{4}} \cdot \frac{-\frac{1}{2} - \frac{1}{2} \cdot \sqrt{5}}{\sqrt{\frac{1}{2} + \frac{1}{2} \cdot \sqrt{5}}}, -\frac{\frac{1}{5} \cdot 5^{\frac{3}{4}}}{\sqrt{\frac{1}{2} + \frac{1}{2} \cdot \sqrt{5}}}\right];$$

$$vec2 := \left[-\frac{\frac{1}{5} \cdot 5^{\frac{3}{4}}}{\sqrt{\frac{1}{2} + \frac{1}{2} \cdot \sqrt{5}}}, 0, \frac{1}{5} \cdot 5^{\frac{3}{4}} \cdot \frac{-\frac{1}{2} - \frac{1}{2} \cdot \sqrt{5}}{\sqrt{\frac{1}{2} + \frac{1}{2} \cdot \sqrt{5}}}\right]$$

$$vec3 := \left[\frac{\frac{1}{5} \cdot 5^{\frac{3}{4}}}{\sqrt{\frac{1}{2} + \frac{1}{2} \cdot \sqrt{5}}}, 0, \frac{1}{5} \cdot 5^{\frac{3}{4}} \cdot \frac{-\frac{1}{2} - \frac{1}{2} \cdot \sqrt{5}}{\sqrt{\frac{1}{2} + \frac{1}{2} \cdot \sqrt{5}}}\right]$$

Creating vectors given from the points found from the command.
```
> yx := vec2-vec1;
> zx := vec3-vec1;
```
Taking the cross product to find the normal which we seek.
```
> simplify(crossprod(yx, zx));
> faces(i);
```

Appendix B

Test figures for 3D

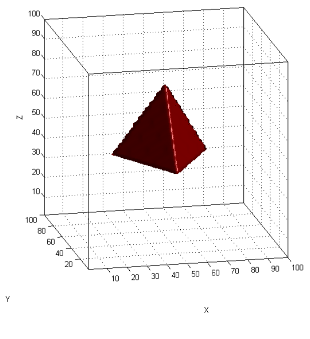


Figure B.1: Example of a regular polyeder made with 4 planes. The figure is centered in the middle, and $N = 100$ has been used since the figure only had to be visualized, not reconstructed.

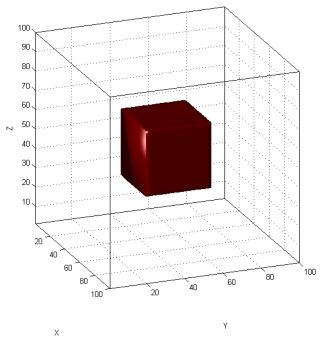


Figure B.2: Example of a regular polyeder made with 6 planes. The figure is centered in the middle, and $N = 100$ has been used since the figure only had to be visualized, not reconstructed.

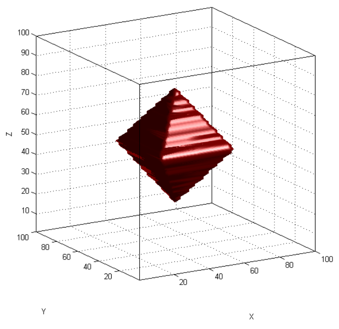


Figure B.3: Example of a regular polyeder made with 8 planes. The figure is centered in the middle, and $N = 100$ has been used since the figure only had to be visualized, not reconstructed.

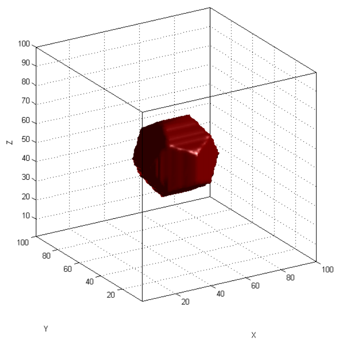


Figure B.4: Example of a regular polyeder made with 12 planes. The figure is centered in the middle, and $N = 100$ has been used since the figure only had to be visualized, not reconstructed.

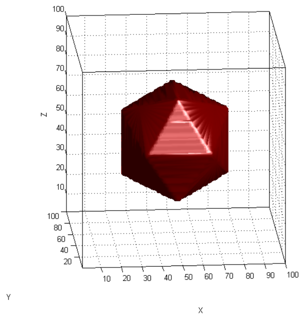


Figure B.5: Example of a regular polyeder made with 20 planes. The figure is centered in the middle, and $N = 100$ has been used since the figure only had to be visualized, not reconstructed.

Appendix C

Scripts for the Grains

```
%2D case
function X = grain2d(N,x0_x,x0_y,n,scalar)

%Initializing
X = zeros(N,N);
theta = 360/(2*n);
a_new = zeros(2,n);
x0_new = [round(N*x0_y) round(N*x0_x)]';

%Computing all the vectors within the unit circle
for i=1:n
    a_new(1,i) = cosd((2*i-1)*theta);
    a_new(2,i) = sind((2*i-1)*theta);
end

%Further initializing
[~, index2] = size(a_new);
h_new = zeros(index2,1);
eqn = zeros(index2,3);
d_new = (N/2)*ones(index2,1)*scalar;

%Determining the parameters needed to write up the equation for a line.
```

```
for t=1:index2
    h_new(t,1) = -a_new(:,t)'*x0_new-d_new(t,1);
    eqn(t,1) = a_new(1,t);
    eqn(t,2) = a_new(2,t);
    eqn(t,3) = h_new(t,1);
end

%Setting the parameters together to create every equation for a line. All
%of the pixel values below this line are set to 1.
for i=1:N
    for j=1:N
        if eqn(1:end,1)*i+eqn(1:end,2)*j+eqn(1:end,3) <= 0
            X(i,j) = 1;
        end
    end
end

X = X(:);

%3D case

%The code took up to much space, so instead of showing the whole thing we
%have only shown the important features.

%First the call for the function is given as
function X = grain3d(N,x0_x,x0_y,x0_z,n,scalar)

%Here we set up our parameters to form the planes in 3D. Everything below
%this planes will be set to 1, meaning that these planes corresponds to
%an edge for our 3D figure

[~, index2] = size(a_new);
h_new = zeros(index2,1);
eqn = zeros(index2,4);
d_new = (N/2)*ones(index2,1)*scalar;

for t=1:index2
    h_new(t,1) = -a_new(:,t)'*x0_new-d_new(t,1).*norm(a_new(:,t));
    eqn(t,1) = a_new(1,t);
    eqn(t,2) = a_new(2,t);
    eqn(t,3) = a_new(3,t);
    eqn(t,4) = h_new(t,1);
end
```

```
for i=1:N
    for j=1:N
        for k=1:N
            if eqn(1:end,1)*i+eqn(1:end,2)*j+eqn(1:end,3)*k+eqn(1:end,4) <= 0
                X(i,j,k) = 1;
            end
        end
    end
end

X = X(:);
```

Appendix D

Test figures for 3D

```
%An example of how we will make use of the new stopping criterion for 3D.
%Initializing
T = 0.5;
tau_background = 0.05;

%Looping over all our iterations.
for i=1:k
    %Taking out the current iteration.
    X_new = reshape(X_cim(:,i),N,N,N);
    %Looping over N. We keep z steady and find the sum of all our xy-planes.
    for l=1:N
        X_use = X_new(:,:,l);
        %Determining points in the background
        S = zeros(N,N);
        S(X_use(:)<tau_background) = 1;
        S = reshape(S,N,N);
        %In case no seed points has been found we
        %set everything to a seed point.
        if isempty(S)
            S = ones(N,N);
        end
        %using the function region grow on the current
        %plane for the current iteration.
        g = regiongrow(reshape(X_use(:),N,N),S,T);
        g = g(:);
        temp_rec = X_use(:);
        temp_rec = temp_rec(:);
        %Finding the sum of the background in our current plane
        sum_background_one_plane(l,i) = sum(temp_rec(g==1));
    end
end
%Summing over all the planes found for one iteration.
for p=1:k
    sum_background(p) = sum(sum_background_one_plane(:,p));
end
%Finding the difference between each iteration.
difference = diff(sum_background);
%Calculating the relative error in the change
%of the background per iteration.
for m=1:length(difference)
    rel_error(m) = abs(difference(m)) / sum_background(m) * 100;
end
%Calculating the relative error in the change
%of the background per 5 iterations.
for j=3:length(difference)-2
    rel_error_mean_cim(j) = ( ( (abs(difference(j-2)) / sum_background(j-2))...
        +(abs(difference(j-1)) / sum_background(j-1))+ ...
        (abs(difference(j)) / sum_background(j))+ ...
        (abs(difference(j+1)) / sum_background(j+1))+ ...
        (abs(difference(j+1)) / sum_background(j+1)) ) ...
        / 5 ) * 100;
end
```

Bibliography

[1] Gonzalez, R. C. and Woods, R. E. and Eddins, S. L., Digital Image Processing using MATLAB, Pearson Prentice Hall, 2004.

[2] Hansen, P. C., Discrete Inverse Problems: Insight and Algorithms, Society for Industrial & Applied, 2010.

[3] Jørgensen, J. H., Knowledge-Based Tomography Algorithms, Department of Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2009.

[4] Kreyszig, E., Kreyszig (1998) Advanced engineering mathematics, John Wiley (New York), 1998.

[5] Lebedev, V. I. and Laikov, D. N., A quadrature formula for the sphere of the 131st algebraic order of accuracy, Doklady. Mathematics, vol. 59, no. 3, p. 477-481, MAIK Nauka/Interperiodica, 1999.

[6] Saxild-Hansen, M., AIR Tools - A MATLAB package for Algebraic Iterative Reconstruction Techniques, Department of Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2010.

[7] True, C. and Jørgensen, D., Mathematical Models for Imagereconstruction, Department of Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2008.

[8] Wang, Z. and Bovik, A. C., Modern image quality assessment, Synthesis Lectures on Image, video, and Multimedia Processing, vol. 2, no. 1, p. 1-156, Morgan & Claypool Publishers, 2006.

[9] ”`http://en.wikipedia.org/wiki/Condition_number#Matrices`”

[10] ”http://www.mathworks.com/matlabcentral/fileexchange/?term=tomobox”

[11] ”http://www.mathworks.com/matlabcentral/fileexchange/27097-getlebedevsphere”

[12] ”http://www.wordiq.com/definition/Inverse_problem”

[13] ”http\protect\kern+.2222em\relax//www2.imm.dtu.dk/~pch/”

[14] ”www.kdassem.dk/undervis/matematik/polyedre/eulersaetn.html”