university of groningen

faculty of mathematics and natural sciences

Iterative image reconstruction algorithms for diagnostic medicine

Bachelor thesis Mathematics and Computing Science

July 2014

Student: J. Snoeijer

Supervisor (Mathematics): R.W.C.P. Verstappen

Supervisor (Computing Science): J.B.T.M. Roerdink

Abstract

In this thesis we consider some iterative methods for image reconstruction in the context of diagnostic medicine. From a CT-scanner we get projection data of a patient. This data is used to reconstruct an image of the patient such that the physician can see into the body of the patient without cutting the patient. This can be seen as a linear programming problem and can be converted to a system of linear equations. This gives rise to mathematical analysis of optimization and row-action methods for solving linear systems. To get good image reconstructions we analyze and compare iterative image reconstruction algorithms such as the Kaczmarz algorithm and block-iterative versions such as Cimmino's algorithm and Block-Kaczmarz. We consider the stability of the block-iterative methods and show the possibility and advantage of parallel computing for the Block-Kaczmarz algorithm.

Contents

1 Introduction

In many situations people want to see something of the interior of an object without damaging the object. For example a cheese factory wants to know how much bubbles there are in the produced cheese. For this example there are possibly alternatives to overcome the problem of cutting all cheeses. If we look at another field such as diagnostic medicine then a physician wants also to see the inside of a human body. Due to the discovery of X-rays by Röntgen there exists physical phenomena to derive information from the inside of a human body without damaging it. Also other properties can be used to derive information from a human body, like radioisotopes, ultrasound or magnetic resonance [14].

This leads us to a general class of problems which we will discuss in this thesis. The problem described above can be seen as a inversion problem [6]. We have an object $\mathbf{x}$ from which we want to know something about the interior. We can relate that object $\mathbf{x}$ to data $\mathbf{y}$, which we know through the following relation:

$$\mathbf{y} = \mathcal{O}\mathbf{x}$$

where $\mathcal{O}$ is a general operator on the object $\mathbf{x}$ such as "send X-rays through the object $\mathbf{x}$". A standard X-ray image shows mainly bones and some organs within the body because they absorb a large part of the radiation [18]. Thus sending X-rays to an object results in projections like 'shadow images'. In this thesis we will focus on *Image Reconstruction* based on this sort of 'projection data'. This will result in the implementation of a number of iterative algorithms to reconstruct image $\mathbf{x}$ based on the data $\mathbf{y}$.

Therefore we want to begin with some analysis on constraint optimization and the Lagrange multiplier in Section 2. In Section 3 we discuss linear programming to minimize linear cost functions subject to linear equality and inequality constraints. This will result in a discussion of the Simplex Method. In Section 4 we consider a number of general methods for solving linear systems.

After these sections we come back to the already sketched context of diagnostic medicine and formulate a discretized model for transmission tomography in Section 5. In Section 6 we consider iterative methods to reconstruct the image from the projection data. The iterative methods we discuss are the row-action methods Kaczmarz (ART), Cimmino (SIRT) and Block-Kaczmarz (SART).

The final part of this thesis brings the discussion about image reconstruction to a basic implementation of the three algorithms, the implementation is discussed in Section 7. When the implementation is clear we can do some practical analysis of the different implementations in Section 8 to compare the performance of the different algorithms. This gives us also a basic understanding of how we can influence the quality of the resulting images. This thesis will end with a conclusion in Section 9 where we summarize the main results we found.

1.1 Some linear algebra notation

In this thesis we will use some linear algebra. In almost all cases it is clear from the context what we mean with our notation. In some cases it is not immediately clear and therefore we will introduce some conventions in our notation:

- Vectors are written in lowercase and bold-face as in: $\mathbf{x} \in \mathbb{R}^n$.
- Matrices are written in uppercase and bold-face as in: $\mathbf{A} \in \mathbb{R}^{m \times n}$. This matrix $\mathbf{A}$ has m rows and n columns.
- Column i of matrix $\mathbf{A}$ is written as $\mathbf{a}_i$ (or if in the context also rows of $\mathbf{A}$ are used then column i can also be written as $\mathbf{A}_{(\bullet,i)}$)
- Row i of matrix $\mathbf{A}$ is written as $\mathbf{A}_{(i,\bullet)}$
- The product between vectors $(\mathbf{u} \cdot \mathbf{v})$ is always the inner product, so we neglect the notion of row or column vectors.
- The i^{th} element of vector $\mathbf{x}$ is written as x_i.

2 Constraint optimization and the Lagrange multiplier

In this section we will consider the field of mathematical optimization. We will consider the minimum of a function of multiple variables and introduce constraints for the minimum. This will end with the Lagrange multiplier.

2.1 Minimum and critical points of functions $\mathbb{R}^n \to \mathbb{R}$

In many applications it is important to find an optimal solution for a quantity that changes over some variables. A natural question in these cases is when a quantity reaches its largest or smallest value. For example in the context of economy people wants to minimize cost or maximize profit. In the case that a quantity only depends on one variable we can use the *derivative* of that function to find extrema of that function. If a quantity depends on more than one variable then we need some extra theory to find the extrema of a function [7]. Therefore we introduce the following definition:

Definition 2.1. Let $f : X \subseteq \mathbb{R}^n \to \mathbb{R},\ X$ open.
The function f has a **local minimum** at the point $\mathbf{a} \in X$ if $\exists U$ as neighborhood of $\mathbf{a}$ such that $f(\mathbf{x}) \geq f(\mathbf{a})\ \forall \mathbf{x} \in U$.

In the same way we could also define a **local maximum** but because the maximum of function f is the minimum of function $g := -f$ we skip that definition.

Note that the definition defines a *local* minimum in stead of a *global* or *absolute*. This is because in a neighborhood of point $\mathbf{a} \in X$ the function f did not reach a lower value than the value at $\mathbf{a}$ but the definition did not exclude the possibility that the function f reaches a lower minimum at a point elsewhere in X. It is also possible that f did not have a minimum in the region X because X is open.

From functions of one variable we know that if we seek a minimum (or maximum) of a function $g : \mathbb{R} \to \mathbb{R}$ we could consider the derivative of the function g. At a minimum the tangent line is horizontal thus the derivative at a minimum is zero. This notion can also be extended to the multi-variable case, which is shown in the following theorem:

Theorem 2.2. Let $f : X \subseteq \mathbb{R}^n \to \mathbb{R}$ differentiable, X open.
If f has a local minimum (or maximum) at $\mathbf{a} \in X$, then $Df(\mathbf{a}) = \mathbf{0}$

Proof. Define a new function $F : \mathbb{R} \to \mathbb{R}$ as $F(t) := f(\mathbf{a} + t\mathbf{h})\ \forall \mathbf{h}$
Then $F(t)$ must have a local minimum at $t = 0$ for all $\mathbf{h}$. Note that $F(t)$ is a function from $\mathbb{R} \to \mathbb{R}$ thus we can use the one-variable calculus, and thus $\frac{\mathrm{d}F}{\mathrm{d}t}(0) = 0$. Using the chain rule, we have also:

$$\frac{\mathrm{d}F}{\mathrm{d}t}(t) = \frac{\mathrm{d}f(\mathbf{a} + t\mathbf{h})}{\mathrm{d}t} = Df(\mathbf{a} + t\mathbf{h}) \cdot \mathbf{h} = \nabla f(\mathbf{a} + t\mathbf{h}) \cdot \mathbf{h}.$$

And thus:

$$0 = \frac{\mathrm{d}F}{\mathrm{d}t}(0) = Df(\mathbf{a}) \cdot \mathbf{h} = f_{x_1}(\mathbf{a})h_1 + f_{x_2}(\mathbf{a})h_2 + \ldots + f_{x_n}(\mathbf{a})h_n \quad \forall \mathbf{h}$$

By choosing $\mathbf{h} = \mathbf{e}_i$ for $i = 1 \ldots n$ (where $\mathbf{e_i} \in \mathbb{R}^n$ is the i^{th} basis vector of $\mathbb{R}^n$) we get:

$$f_{x_i}(\mathbf{a}) = 0 \quad \forall i = 1 \ldots n$$

And thus $Df(\mathbf{a}) = \mathbf{0}$. □

The previous theorem shows that if function f has a local minimum at $\mathbf{a} \in X$ then $Df(\mathbf{a}) = \mathbf{0}$. This is only a *necessary* condition for a local minimum (or maximum) and not a *sufficient* condition.

For example the function $f(x, y) = x^2 - y^2$ has a derivative $Df(x, y) = (2x, -2y)$. At $\mathbf{a} = (0, 0)$ we see $Df(\mathbf{a}) = \mathbf{0}$, but $f(x, y)$ has not a minimum (or maximum) at $(0, 0)$ as seen in Figure 1.

To formalize the difference in meaning of a minimum (or maximum) at a point $\mathbf{a}$ of a function f and the points that satisfy the condition of Theorem 2.2 we introduce the notion of critical points:

Definition 2.3. Let $f : X \subseteq \mathbb{R}^n \to \mathbb{R}$ differentiable, X open.
A point $\mathbf{a} \in X$ is called a **critical point** if $Df(\mathbf{a}) = \mathbf{0}$.

From Theorem 2.2 we see that the set of minimum (or maximum) points is a subset of the set of critical points of a function f.

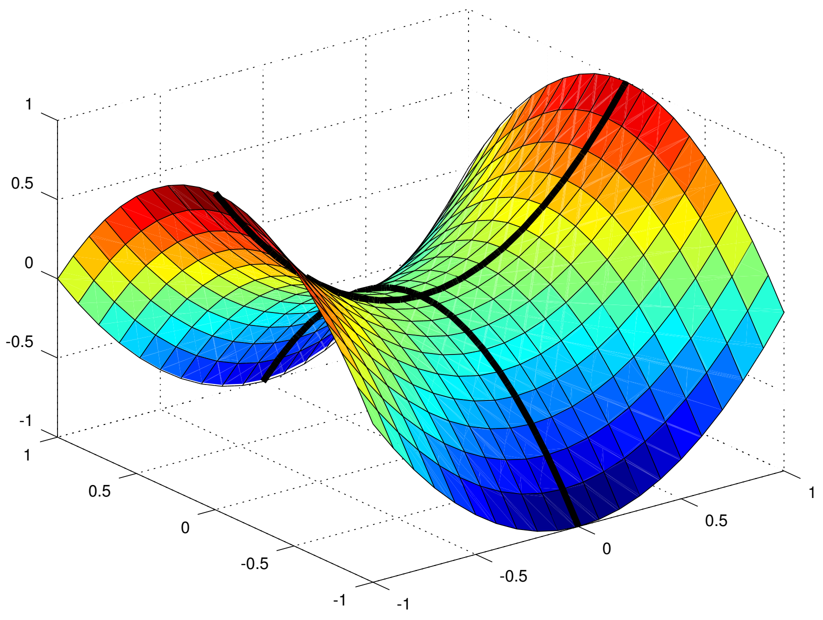


Figure 1: Plot of the function $f(x, y) = x^2 - y^2$ which does not have a minimum (or maximum) at $(0, 0)$. The point $(0, 0)$ is called a saddle-point.

2.2 Constraint minimization

In many applications it is the case that we need not only to minimize (or maximize) a function, but there are also some additional constraints for a solution. First we will look at an example to see the context of this minimization:

Example 2.4. An open rectangular box (a box without a top-side) needs to have a specified volume: 4 dm^3. The manufacturer wants to minimize the amount of materials used to make this box. The sizes of the box can be described by three variables x, y, z as the length of the box in the appropriate dimension. Thus we want to minimize:

$$A(x, y, z) = 2xy + 2yz + xz$$

We have also the constraint that describes the volume of the box, namely:

$$V(x, y, z) = xyz = 4$$

In this example we have two equations but three variables. In general this system is not solvable, but in this case we are lucky that we can solve the variable z in terms of x and y using the constraint equation and we get the new equation to minimize:

$$a(x, y) = 2xy + \frac{8}{x} + \frac{4}{y}.$$

Thus we have one equation and two variables. We can find the critical points of $a(x, y)$ by setting the derivative equal to zero: $Da(x, y) = \mathbf{0}$. This gives us two equations and two variables, from which we can conclude that $x = 2$ and $y = 1$ yields a local minimum for the new area function. After computing $z = \frac{4}{xy} = 2$ we get the solution for the minimization problem as shown in Figure 2.

From the previous example we see that we want to minimize a function $f : \mathbb{R}^n \to \mathbb{R}$ (in the example $A(x, y, z)$) subject to a constraint $g(\mathbf{x}) = c$ (in the example $V(x, y, z) = 4$). In the example it was possible to solve one variable in terms of the other variables using the constraint equation, but that is not always easy or possible. So we want another method to solve this constraint optimization problems. This leads us to the method of Lagrange multiplier.

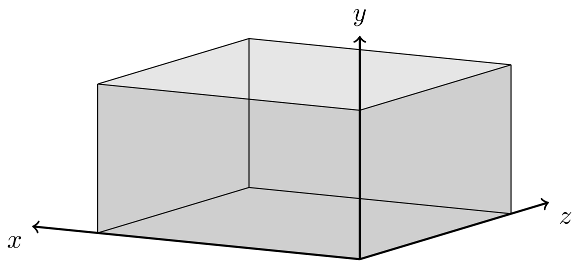


Figure 2: The box for the manufacturer wo wants to minimize material costs for a volume of 4 dm^3

2.3 Geometric properties

Instead of using some algebra to reduce the number of variables in the optimization problem we will consider some geometric properties of optimization problems to see another method to solve the optimization problems [3].

From vector calculus we know the following theorem that states that the directional derivative of a multivariate function is the dot product of the gradient and the (unit) direction[7]:

Theorem 2.5. Let $f : X \subseteq \mathbb{R}^n \to \mathbb{R}$ differentiable at $\mathbf{a} \in X,\ X$ open.
Then $\forall \mathbf{v} \in \mathbb{R}^n$ with $\|\mathbf{v}\| = 1$, the directional derivative $D_{\mathbf{v}}f(\mathbf{a})$ exists and we have

$$D_{\mathbf{v}}f(\mathbf{a}) = \nabla f(\mathbf{a}) \cdot \mathbf{v}.$$

If we see a function $f : \mathbb{R}^n \to \mathbb{R}$ as a pressure in $\mathbb{R}^n$ and the pressure function is differentiable then we can seek for the direction in which the pressure increases the most. Suppose we look at point $\mathbf{x} \in \mathbb{R}^n$ in the direction of $\mathbf{u} \in \mathbb{R}^n$ with $\|\mathbf{u}\| = 1$. Then from the previous theorem we know:

$$D_{\mathbf{u}}f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{u}.$$

Let θ be the angle between $\mathbf{u}$ and the gradient vector $\nabla f(\mathbf{x})$. Then the equation becomes:

$$D_{\mathbf{u}}f(\mathbf{x}) = \|\nabla f(\mathbf{x})\|\|\mathbf{u}\| \cos\theta = \|\nabla f(\mathbf{x})\| \cos\theta$$

This leads us to the following theorem:

Theorem 2.6. Let $f : X \subseteq \mathbb{R}^n \to \mathbb{R}$ differentiable, X open.
The directional derivative $D_{\mathbf{u}}f(\mathbf{x})$ is maximized with respect to the direction when $\mathbf{u}$ points in the *same* direction as $\nabla f(\mathbf{x})$ and is minimized when $\mathbf{u}$ points in the *opposite* direction. Furthermore the minimum and maximum values of $D_{\mathbf{u}}f(\mathbf{x})$ are $\mp\|\nabla f(\mathbf{x})\|$.

Proof. From the previous discussion we get:

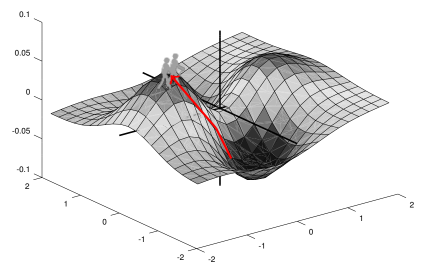
$$D_{\mathbf{u}}f(\mathbf{x}) = \|\nabla f(\mathbf{x})\|\|\mathbf{u}\| \cos\theta = \|\nabla f(\mathbf{x})\| \cos\theta$$

Where θ is the angle between $\mathbf{u}$ and $\nabla f(\mathbf{x})$. To maximize this expression we need $\cos\theta = 1$, thus $\theta = 0$. This means that $\mathbf{u}$ and $\nabla f(\mathbf{x})$ points in the same direction. To minimize the expression we need $\cos\theta = -1$, thus $\theta = \pi$. This means that $\mathbf{u}$ and $\nabla f(\mathbf{x})$ points in the opposite direction. □

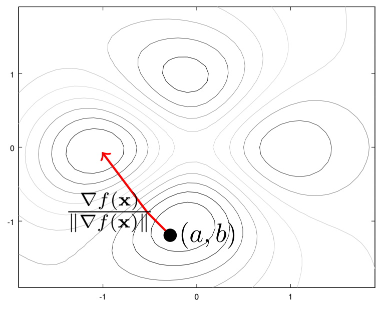
Example 2.7. If we restrict Theorem 2.6 to $\mathbb{R}^2$ then we can get some feeling with the geometric interpretation of this theorem. The function $z = f(x, y)$ describes the surface of a mountain. If we want to increase the height (or explicit the z-value) as fast as possible then we need to climb the mountain in the direction of the gradient of our current location, see Figure 3. The vector $\nabla f(a, b) \in \mathbb{R}^2$ describes the direction on the map of the mountain.

From the previous example we could also suggest that the gradient vector points perpendicular to the height lines of the map. This is indeed the case, as the following theorem states:

Theorem 2.8. Let $f : X \subseteq \mathbb{R}^n \to \mathbb{R}, f \in C^1,\ X$ open.
Let S be the **level set** defined by $S = \{\mathbf{x} \in X \mid f(\mathbf{x}) = c\}$. If $\mathbf{a} \in S$ then $\nabla f(\mathbf{a})$ is perpendicular to S.



(a) Plot of a landscape of a mountain.



(b) Map of level set of the mountain.

Figure 3: To increase the height as fast as possible climb in the direction $\frac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|}$.

Proof. Let C be a curve in S parameterized by $\mathbf{x}(t) = (x_1(t), x_2(t), \ldots, x_n(t))$ with $a < t < b$ and $\exists t_0 \in (a, b)$ such that $\mathbf{x}(t_0) = \mathbf{a}$. $C \subset S$, thus:

$$f(\mathbf{x}(t)) = f(x_1(t), x_2(t), \ldots, x_n(t)) = c$$

Thus if we derive f we get:

$$\frac{\mathrm{d}f(\mathbf{x}(t))}{\mathrm{d}t} = \frac{\mathrm{d}c}{\mathrm{d}t} \equiv 0.$$

And we have also the function $f \circ \mathbf{x} : (a, b) \to \mathbb{R}$ that gives with the chain rule:

$$\frac{\mathrm{d}f(\mathbf{x}(t))}{\mathrm{d}t} = \nabla f(\mathbf{x}(t)) \cdot \mathbf{x}'(t)$$

At t_0 this gives:

$$\nabla f(\mathbf{x}(t_0)) \cdot \mathbf{x}'(t_0) = 0$$

Note that the vector $\mathbf{v} = \mathbf{x}'(t_0)$ is the velocity vector at $x(t_0)$ and thus tangent to S at $x(t_0)$. Combined with $\mathbf{x}(t_0) = \mathbf{a}$ becomes the equation:

$$\nabla f(\mathbf{a}) \cdot \mathbf{v} = 0$$

And thus $\nabla f(\mathbf{a})$ is perpendicular to the level set S. $\square$

If we want to find a minimum of a function $f : \mathbb{R}^n \to \mathbb{R}$ we can also look at the level sets of function f. If the minimum of f exists then there will be a lowest level set that contains the minimum of f.

If we want to solve the optimization problem subject to a constraint $g(\mathbf{x}) = c$ then at a minimum the value of f that satisfies the constraint $g(\mathbf{x}) = c$ cannot decrease along the curve $g = c$. Otherwise the point was not a minimum. On the curve $g = c$ we seek points $\mathbf{x}$ where $f(\mathbf{x})$ did not change, that points are the critical points that could be a minimum for this optimization problem.

If we found a critical point then it could be that the curve $g = c$ is parallel to a level set $f(\mathbf{x}) = h$. The other possibility is that $g = c$ crosses a plateau where $f(\mathbf{x}) = h$ and $f(\mathbf{x})$ did not change in any direction. Thus $\exists P$ open $\in \mathbb{R}^n$, such that $\forall \mathbf{a} \in P : f(\mathbf{a}) = h$.

The first possibility (the curve $g = c$ is parallel to a level set $f(\mathbf{x}) = h$ at $\mathbf{a} \in \mathbb{R}^n$) is by Theorem 2.8 equivalent to the case where $\nabla g(\mathbf{a})$ is parallel to $\nabla f(\mathbf{a})$. This can be expressed as

$$\nabla f(\mathbf{a}) = \lambda \nabla g(\mathbf{a}) \tag{1}$$

The second possibility ($f(\mathbf{a})$ did not change in any direction) can be expressed as $\nabla f(\mathbf{a}) = 0$. Thus by setting $\lambda = 0$ this can also be expressed by Equation 1.

Formally this method can be described by the following theorem:

Theorem 2.9. Let X open and $X \subseteq \mathbb{R}^n$ and $f, g : X \to \mathbb{R} \in C^1$.
Let $S = \{\mathbf{x} \in X \mid g(\mathbf{x}) = c\}$ denote the level set of g at height c. Then if $f|_S$ has an extremum at $\mathbf{a} \in S$ such that $\nabla g(\mathbf{a}) = \mathbf{0}$, there must be a $\lambda \in \mathbb{R}$ such that:

$$\nabla f(\mathbf{a}) = \lambda \nabla g(\mathbf{a})$$

This give the following recipe to find critical points of f subject to the constraint $g(\mathbf{x}) = c$:

Recipe 2.10. Find critical points of a function f subject to a constraint $g(\mathbf{x}) = c$:

1. Form the vector equation $\nabla f(\mathbf{a}) = \lambda \nabla g(\mathbf{a})$
2. Solve the system of $n+1$ variables and $n+1$ equations: $\begin{cases} \nabla f(\mathbf{x}) = \lambda \nabla g(\mathbf{x}) \\ g(\mathbf{x}) = c \end{cases}$
3. Determine the nature of f at the found critical point.

Definition 2.11. The variable $\lambda \in \mathbb{R}$ in the previous recipe is called **Lagrange multiplier**.

3 Linear Programming

In the previous section we saw how we can solve a general optimization problem in minimizing a cost function subject to an equality constraint. This leads to a recipe to solve an optimization problem using the Lagrange Multiplier. However, equality constraints are not the only type of constraints that can be used in optimization problems. Therefore we introduce linear programming: the problem of minimizing a linear cost function subject to linear equality and inequality constraints [2, 8].

3.1 General and standard linear programming problems

Supporting the description of a general linear programming problem we will give an example of a linear programming problem that contains the most elements we will use in this section.

Example 3.1. An example of a linear programming problem can be stated as follows:

$$\begin{aligned} \text{minimize} \quad & f(x_1,x_2,x_3,x_4) = 2x_1 - x_2 + 4x_3 \\ \text{subject to} \quad & \begin{cases} x_1 + x_2 + x_4 \leq 2 \\ 3x_2 - x_3 = 5 \\ x_3 + x_4 \geq 3 \\ x_1 \geq 0 \\ x_3 \leq 0 \end{cases} \end{aligned}$$

Definition 3.2. A **cost vector** $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ is a vector that defines the costs per variable in $\mathbf{x} \in \mathbb{R}^n$ and in the linear programming problem the **cost function** $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$ will be minimized.

The constraints on the cost function can be divided into three groups of constraints, namely the constraints with a $\geq, \leq$ and $=$ relation. The sets M_1, M_2 and M_3 contains the indices of the constraints of the original problem related to the relations $\geq, \leq$ and $=$, respectively. The sets N_1 and N_2 contains the indices of the variables x_i that are non-negative or non-positive, respectively. In abstract form becomes the problem:

$$\begin{aligned} \text{minimize} \quad & f_{\mathbf{c}}(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x} \\ \text{subject to} \quad & \begin{cases} \mathbf{a}_i \cdot \mathbf{x} \geq b_i, & i \in M_1 \\ \mathbf{a}_i \cdot \mathbf{x} \leq b_i, & i \in M_2 \\ \mathbf{a}_i \cdot \mathbf{x} = b_i, & i \in M_3 \\ x_j \geq 0, & j \in N_1 \\ x_j \leq 0, & j \in N_2 \end{cases} \end{aligned}$$

Definition 3.3. The following terminology is used in relation to the linear programming problem:

- The variables $x_1, x_2, \ldots, x_n$ are called **decision variables**.
- If $j \notin N_1 \wedge j \notin N_2$ then x_j is called a **free variable**
- The vector $\mathbf{x} \in \mathbb{R}^n$ that satisfies all the constraints is called a **feasible solution**
- The set of all feasible solutions is called **feasible set**
- A feasible solution $\mathbf{x}^*$ that minimizes the cost function is called the **optimal solution** and the value $\mathbf{c} \cdot \mathbf{x}^*$ is the **optimal cost**
- If $\forall K \in \mathbb{R}\ \exists$ *feasible solution* with cost $< K$, then the cost is $-\infty$ or **unbounded (below)**

If we look in more detail to the constraints we can see some equivalent relations between them to convert all type of constraints to one type.

Recipe 3.4. Convert all linear programming systems to general form:

1. The constraint $\mathbf{a}_i \cdot \mathbf{x} = b_i$ is equivalent to the two constraints $\mathbf{a}_i \cdot \mathbf{x} \leq b_i$ and $\mathbf{a}_i \cdot \mathbf{x} \geq b_i$.
2. In the same way we can also rewrite the constraint $\mathbf{a}_i \cdot \mathbf{x} \leq b_i$ to $(-\mathbf{a}_i) \cdot \mathbf{x} \geq -b_i$.
3. Constrains in the form $x_j\ @\ 0$ can be converted to $\mathbf{a}_i \cdot \mathbf{x}\ @\ b_i$ where $\mathbf{a}_i$ is the j^{th} unit-vector and $b_i = 0$. The @ denoted the $\leq$ and $\geq$ relation.

Thus the feasible set in a general linear programming problem can be expressed in terms of inequality constraints of the form $\mathbf{a}_i \cdot \mathbf{x} \geq b_i$. If there are m constraints then this can be expressed in a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ as described by the following definition:

Definition 3.5. A linear programming problem of the form

$$\begin{aligned} \text{minimize} \quad & f_{\mathbf{c}}(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \geq \mathbf{b} \end{aligned}$$

is said to be in **general form**.

Next to the definition of a *general* form we will state also the definition of the *standard* form of a linear programming problem:

Definition 3.6. A linear programming problem of the form

$$\begin{aligned} \text{minimize} \quad & f_{\mathbf{c}}(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x} \\ \text{subject to} \quad & \begin{matrix} \mathbf{A}\mathbf{x} = \mathbf{b} \\ \mathbf{x} \geq \mathbf{0} \end{matrix} \end{aligned}$$

is said to be in **standard form**.

The standard form is a form in which explicitly is expressed that the value of b_i must be built up from a non negative usage of recourses $\mathbf{A}_{(i,\bullet)}$ while we also want to minimize the cost $\mathbf{c} \cdot \mathbf{x}$.

Example 3.7. An example of a problem in standard form is the diet problem. In that problem are n different foods and m different nutrients. A food i has an own amount of nutrient j stored in food-matrix $\mathbf{A}$: a_{ji}. The vector $\mathbf{b}$ describes the amount of the different nutrients for the food.

The standard form is a special case of the general form of a linear programming problem as we saw in Recipe 3.4, but the statement is also true the other way around as the following theorem states:

Theorem 3.8. A linear programming problem in general form can also be converted in an equivalent linear programming problem in standard form. I.e. from a feasible solution of the first problem a feasible solution of the other problem can be constructed with the same cost.

Proof. The proof is descriptive following Recipe 3.9. □

Recipe 3.9. Convert a general form to a standard form

1. An unrestricted variable x_j can be replaced by x_j^+ and x_j^- where $x_j^+ \geq 0$ and $x_j^- \geq 0$. This is true because $\forall a \in \mathbb{R}\ \exists x, y \in \mathbb{R}$ such that $a = x - y$.
2. The inequality constraint $\mathbf{A}_{(i,\bullet)} \cdot \mathbf{x} \geq b_i$ can be replaced by the constraints $\mathbf{A}_{(i,\bullet)} \cdot \mathbf{x} - s_i = b_i$ and $s_i \geq 0$.

Thus all linear programming problems can be converted in general and standard form.

Example 3.10. Now we will introduce an example of a linear programming problem that we will use also in the next sections to illustrate the topics we discuss there.

$$\begin{aligned} \text{minimize} \quad & f_{\mathbf{c}}(\mathbf{x}) = -10x_1 - 12x_2 - 12x_3 \\ \text{subject to} \quad & \begin{cases} x_1 + 2x_2 + 2x_3 \leq 20 \\ 2x_1 + x_2 + 2x_3 \leq 20 \\ 2x_1 + 2x_2 + x_3 \leq 20 \\ x_1 \geq 0 \\ x_2 \geq 0 \\ x_3 \geq 0 \end{cases} \end{aligned}$$

We can convert this linear programming problem to standard form by introducing slack variables x_4, x_5 and x_6:

$$\begin{aligned} \text{minimize} \quad & f_{\mathbf{c}}(\mathbf{x}) = -10x_1 - 12x_2 - 12x_3 \\ \text{subject to} \quad & \begin{cases} x_1 + 2x_2 + 2x_3 + x_4 = 20 \\ 2x_1 + x_2 + 2x_3 + x_5 = 20 \\ 2x_1 + 2x_2 + x_3 + x_6 = 20 \\ x_1, x_2, \ldots, x_6 \geq 0 \end{cases} \end{aligned}$$

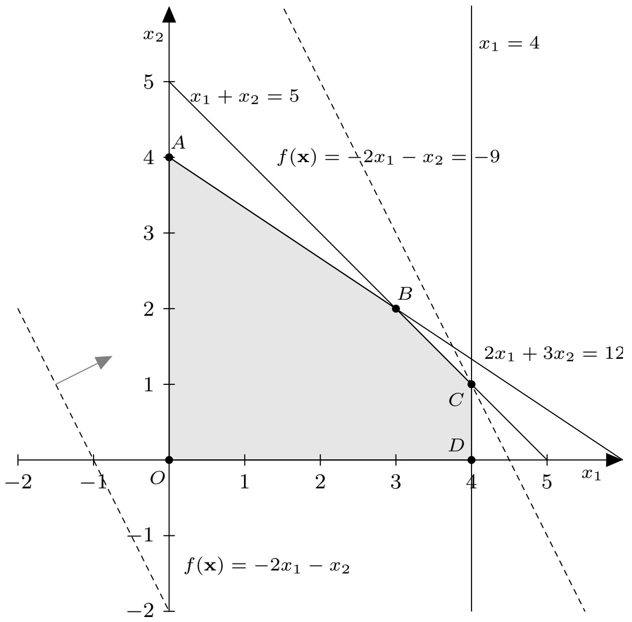


Figure 4: Graphical solution of the two variable linear programming problem.

3.2 Graphically solve linear programming problems

In this section we will discuss some graphical ideas about the feasible set and methods for solving linear programming problems. Therefore we will introduce a simple two variable problem[9]:

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) = -2x_1 - x_2 \\ \text{subject to} \quad & \begin{cases} x_1 + x_2 \leq 5 \\ 2x_1 + 3x_2 \leq 12 \\ x_1 \leq 4 \end{cases} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

The non-negativity constraints $\mathbf{x} \geq \mathbf{0}$ describe that the solution for this problem will be in the first quadrant of the $(x_1, x_2)-$plane. We can plot the lines from the constraints when the equality holds. For example the constraint $x_1 + x_2 \leq 5$ results in a line $x_1 + x_2 = 5$. This lines separates the plane into two sections. Now we need to find out on which section the constraint holds. If we take for example the origin then x_1 and x_2 are both zero and thus we have $x_1 + x_2 = 0 \leq 5$. Therefore the section containing the origin is the section that holds the feasible set. If we draw lines for the three constraints we get a picture like Figure 4. In this case we see a marked region in which the problem has a feasible solution.

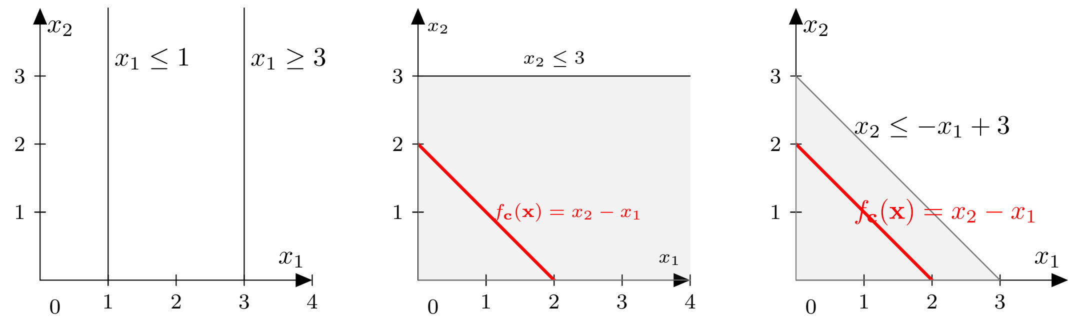
We want to minimize the linear function $f(\mathbf{x}) = -2x_1 - x_2$. In the $(x_1, x_2)-$plane solutions for this function are lines with slope $\frac{-2}{1} = -2$. The value on the line decreases when the line moves to the north-east corner of the first quadrant. Thus, to find the feasible solution $\mathbf{x}$ that minimizes $f(\mathbf{x})$ we seek the last point(s) $\mathbf{x}$ that exists in the intersection between $f(\mathbf{x})$ and the feasible set, when $f(\mathbf{x})$ decreases.

If we move the line $f(\mathbf{x}) = -2x_1 - x_2$ to the north-east corner we see that the value of $f(\mathbf{x})$ decreases. From the picture we see that this line hits first point O (at $f(\mathbf{x}) = 0$) then point A (at $f(\mathbf{x}) = -4$), then D and B (both at $f(\mathbf{x}) = -8$) and as last point C (at $f(\mathbf{x}) = -9$)

From the description and the picture it looks that the optimal solution (if a unique solution exists) is on the boundary of the feasible region. This is indeed the case, as we will see in the next section.

In the case of Figure 4 there exists a unique feasible solution, the point C. However, this is not always the case. Also other cases can arise for a minimization problem. For example the cases shown in Figure 5a, 5b and 5c.

It is harder to draw and interpret pictures for solving linear programming problems with more than two variables / dimensions so the ideas are presented in $2D$. The notion of a line that divides



(a) Sketch of situation where no feasible set exists, then there are contradicting constraints on some variable such as $x_1 \leq 1$ and $x_1 \geq 3$

(b) Sketch of situation in case where a feasible set is unbounded and $\forall z \to -\infty\ \exists (x_1, x_2)$ in feasible set. For example: $(-z, 0)$

(c) Sketch of situation in case where a boundary of the feasible region is parallel to the level sets of the function we want to minimize.

Figure 5: Sketches of possible types of constraints in which no feasible set exists (Figure 5a), a unbounded feasible set exists (Figure 5b) or not a unique minimum exists (Figure 5c).

the plane into two sections is also expendable to $\mathbb{R}^3$ and $\mathbb{R}^n$. The lines of the $2D$ picture will then be planes (for $\mathbb{R}^3$) or hyperplanes (for $\mathbb{R}^n$).

3.3 Convex sets

At the beginning of the first discussion of the *minimum* and *critical point* in Section 2.1, we saw that there was not an equivalence relation between a minimum of a function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$ and a critical point of that function. Or more precisely, if we found a critical point $\mathbf{a} \in \mathbb{R}^n$ of the function $f(\mathbf{x})$ in general it is not true that the critical point $\mathbf{a}$ is also the point on witch the minimum of a function $f(\mathbf{x})$ is attained. In this section we will construct additional constraints such that there is an equivalence for the minimum point and the point $\mathbf{a}$ [2].

3.3.1 Polyhedron

To get this equivalence we introduce the following definition:

Definition 3.11. A **polyhedron** P is a set that can be described in the form $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \geq \mathbf{b}\}$, where $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a matrix and $\mathbf{b} \in \mathbb{R}^m$ a vector.

Example 3.12. Following Definition 3.5, the general form of a linear programming problem can be described by the inequality constraints of the form $\mathbf{A}\mathbf{x} \geq \mathbf{b}$. Thus the feasible set of any linear programming problem is a polyhedron.

Definition 3.13. A set $S \subset \mathbb{R}^n$ is **convex** if $\forall x, y \in S$ and $\lambda \in [0, 1]$ the following statement holds:

$$z_\lambda = \lambda x + (1 - \lambda) y \in S.$$

Note that the points $z_\lambda \in S$ are points on the line segment between the two points $x, y \in S$. Thus a set S is convex if the line segments between two points $x, y \in S$ are also in S.

For our discussion about about polyhedrons and convexity some facts are important for the further discussion:

Theorem 3.14. The following statements holds:

- The intersection of convex sets is also convex.
- An halfspace $H_{\mathbf{a}}^b = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a} \cdot \mathbf{x} \geq b\}$ with $\mathbf{0} \neq \mathbf{a} \in \mathbb{R}^n$ and $b \in \mathbb{R}$ is a convex set.
- Every polyhedron is a convex set.

Proof. We prove the statements one by one:

- The proof of the first part is by induction over n where n is the number of intersections. Let S_i for $i \in \mathbb{N}$ be convex sets. Clearly, the (intersection)set $I_1 = S_1$ is convex. For $1 \leq n \leq k$ let $I_n = \bigcap_{i=1}^{n} S_i$ be convex.

  First we look at the (intersection)set I_{k+1}:

$$\begin{aligned} I_{k+1} &= \bigcap_{i=1}^{k+1} S_i \\ &= \left(\bigcap_{i=1}^{k} S_i \right) \cap S_{k+1} \\ &= I_k \cap S_{k+1} \end{aligned}$$

  Suppose $x, y \in I_{k+1}$, because I_{k+1} is an intersection we have $x, y \in I_k$ and $x, y \in S_{k+1}$. Because I_k and S_{k+1} are convex the points $z_\lambda = \lambda x + (1-\lambda)y$ are in I_k and S_{k+1}. Thus the points are also in the intersection I_{k+1}. Therefore the intersection I_{k+1} is also convex. And thus is the intersection of convex sets also convex.

- Let $H_{\mathbf{a}}^{b} = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{a} \cdot \mathbf{x} \geq b\}$ with $\mathbf{0} \neq \mathbf{a} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. Let $x, y \in H_{\mathbf{a}}^{b}$ and $\lambda \in [0,1]$. For the points $\lambda\mathbf{x} + (1-\lambda)\mathbf{y}$ on the line segment between $\mathbf{x}$ and $\mathbf{y}$ we have:

$$\mathbf{a} \cdot (\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \geq \lambda b + (1-\lambda)b = b$$

  Thus the points $z_\lambda = \lambda\mathbf{x} + (1-\lambda)\mathbf{y} \in H_{\mathbf{a}}^{b}$. Therefore the halfspace $H_{\mathbf{a}}^{b}$ is convex.

- A polyhedron is defined as $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} \geq \mathbf{b}, \mathbf{A} \in \mathbb{R}^{m \times n}\}$ and is thus the intersection of m halfspaces $H_{\mathbf{A}_{(i,\bullet)}}^{b_i}$ with $i = 1, 2, \ldots m$ where $\mathbf{A}_{(i,\bullet)}$ is the i^{th} row of matrix $\mathbf{A}$. The halfspaces are convex and the intersection of convex sets is convex thus a polyhedron is convex.

$\square$

3.3.2 Extreme points and basic feasible solution

From Figure 4 we noted that an optimal solution to a given linear programming problem is in a corner of the feasible set. In the picture the meaning of a corner was clear, but if we want to use this property in computations we need to define what it means to be at a 'corner' of the feasible set. We do that in terms of a polyhedron:

Definition 3.15. Let P be a polyhedron. A vector $\mathbf{x} \in P$ is a **extreme point** of P if we cannot find two other vectors $\mathbf{y}, \mathbf{z} \in P$ with $\mathbf{x} \neq \mathbf{y}$ and $\mathbf{x} \neq \mathbf{z}$ and $\lambda \in [0,1]$ such that:

$$\mathbf{x} = \lambda\mathbf{y} + (1-\lambda)\mathbf{z}$$

With this definition we require that an extreme point is a point that is not on a line segment between two points in P. If $\mathbf{x}$ is between two other points in P then in both directions $\lambda \to 0$ and $\lambda \to 1$ there exists points in P and thus is x not an extreme on that line in P. Both cases are illustrated in Figure 6.

To move to an algebraic interpretation of a corner we introduce the concept of a basic feasible solution.

Definition 3.16. If a vector $\mathbf{x}^*$ satisfies constraint i: $\mathbf{A}_{(i,\bullet)} \cdot \mathbf{x}^* = b_i$ of the general linear programming problem then the corresponding constraint is called **active**.

Definition 3.17. Given the polyhedron $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ with $\mathbf{A} \in \mathbb{R}^{m \times n}$ and let $\mathbf{x}^* \in \mathbb{R}^n$. Then

- The vector $\mathbf{x}^*$ is a **basic solution** if the following two statements holds:
  1. All m constraints of $\mathbf{A}\mathbf{x} = \mathbf{b}$ are active.
  2. Out of all the corresponding vectors of the constraints that are active at $\mathbf{x}^*$ from $\mathbf{A}_{(i,\bullet)}$ or $\mathbf{e}_i$ for all constraints $x_i = 0$ there are n linearly independent.

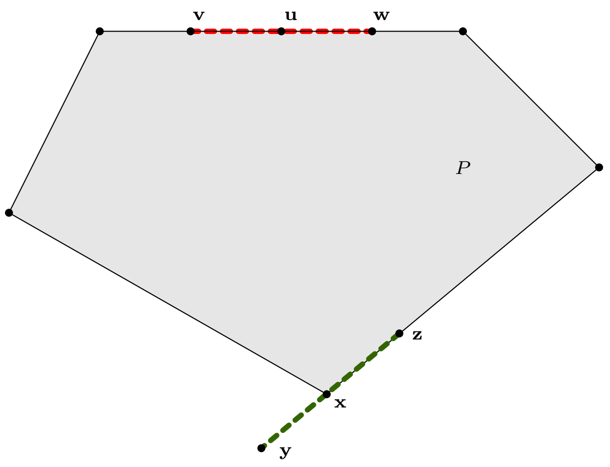


Figure 6: The point $\mathbf{x}$ is an extreme point of P because if $\mathbf{x} = \lambda\mathbf{y} + (1-\lambda)\mathbf{z}$ with $\mathbf{x} \neq \mathbf{y}$ and $\mathbf{x} \neq \mathbf{z}$ then $\mathbf{y} \notin P$ or $\mathbf{z} \notin P$. The point $\mathbf{u}$ is not an extreme point because $\mathbf{v}, \mathbf{w} \in P$ and $\exists\lambda \in [0,1]$ such that $\mathbf{u} = \lambda\mathbf{v} + (1-\lambda)\mathbf{w}$

- If $\mathbf{x}^*$ is a basic solution that satisfies all the original constraints then $\mathbf{x}^*$ is a **basic feasible solution**.

Example 3.18. In this example we consider a polyhedron $P = \left\{(x_1, x_2, x_3) \in \mathbb{R}^3 \mid x_1 + x_2 + x_3 = 1, \mathbf{x} \geq \mathbf{0}\right\}$. This polyhedron defines a plane in $\mathbb{R}^3$ as shown in Figure 7. All points on the marked plane satisfy the 4 constrains: $x_1 + x_2 + x_3 = 1$ and $\mathbf{x} \in \mathbb{R}^3 \geq \mathbf{0}$. Thus all points on the plane are feasible solutions, according to Definition 3.3.

If we consider the basic solution we look at the constraints when the equality-relation must be satisfied (also for the $\geq$ and $\leq$ relations), according to Definition 3.17. Thus we search points $\mathbf{x} \in \mathbb{R}^3$ that satisfies:

$$\mathbf{Ax} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{b}$$

If we make the first three rows of this system active then we get an invertible matrix $\mathbf{A}^*$ from the original matrix $\mathbf{A}$. This matrix has one row (the first one) related to the constraint $x_1 + x_2 + x_3 = 1$. The other two rows are rows of the type $\mathbf{e}_i$ that corresponds to $x_i = 0$. We have chosen three linearly independent rows, and thus $\mathbf{A}^*$ is invertible. Thus we can compute $\mathbf{x} = \mathbf{A}^{*-1}\mathbf{b} = (0,0,1)$ as a basic solution. If we check if this solution is also feasible, which is indeed the case, then we have a basic feasible solution.

In Figure 7 we see the basic feasible solutions A, B and C. These points are basic solutions because they satisfy the constraint $x_1 + x_2 + x_3 = 1$ and two of the three constraints $x_i = 0$ for $i = 1,2,3$. Point D is not a basic solution because $D = (0,0,0)$, and thus the constraint $(1,1,1) \cdot \mathbf{x} = 1$ failed for D. Point E is a feasible solution (see Definition 3.3) but it is not basic because only two constraints are active. On E only $x_1 + x_2 + x_3 = 1$ and $x_2 = 0$ are active.

Theorem 3.19. Let P be an non empty polyhedron and $\mathbf{x}^* \in P$. Then the following statements are equivalent:

- $\mathbf{x}^*$ is an extreme point.
- $\mathbf{x}^*$ is a basic feasible solution.

Proof. For the proof of this theorem, see Reference [2], Theorem 2.3. □

3.4 Solve Linear programming problems

If we want to solve linear programming problems we write the polyhedron in a standard form: $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{Ax} \geq \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$, where $\mathbf{A} \in \mathbb{R}^{m \times n}$. Thus we have m equality constraints that defines the

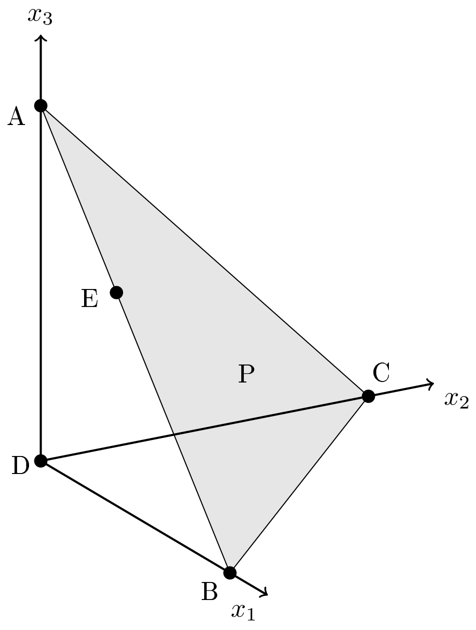


Figure 7: $P = \big\{(x_1, x_2, x_3) \in \mathbb{R}^3 \mid x_1 + x_2 + x_3 = 1, \mathbf{x} \geq \mathbf{0}\big\}$

polyhedron. For now we assume that the m rows of matrix $\mathbf{A}$ are linearly independent. The rows of matrix $\mathbf{A}$ are n-dimensional, thus $m \leq n$.

For a basic solution we need n linearly independent constraints that are active. Also every basic solution must satisfy the m equality constraints $\mathbf{Ax} = \mathbf{b}$. Thus there are m active constraints. To get a total of n active constraints we can choose $n - m$ variables x_i that must satisfy $x_i = 0$. The following theorem shows how we can choose these x_i variables.

Theorem 3.20. Let $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$ describe the constraints for a linear programming problem. Assume $\mathbf{A} \in \mathbb{R}^{m \times n}$ has linearly independent rows. Then, the vector $\mathbf{x} \in \mathbb{R}^n$ is a basic solution if $\mathbf{Ax} = \mathbf{b}$ and there exists indices $B(1), B(2), \ldots, B(m)$ such that:

1. The columns $\mathbf{A}_{(\bullet, B(i))}$ for $i = 1, 2, \ldots, m$ are linearly independent.
2. If $i \notin I = \{B(1), B(2), \ldots, B(m)\}$ then $x_i = 0$.

Proof. Let $\mathbf{x} \in \mathbb{R}^n$. Suppose $\exists B(1), B(2), \ldots, B(m)$ that satisfy the two conditions of the theorem. Because the active constraints $x_i = 0$ for $i \notin I$ do not contribute in $\mathbf{Ax}$ we can convert $\mathbf{x} \in \mathbb{R}^n$ to a (not longer) vector $\mathbf{x^B} \in \mathbb{R}^m$ where $x_i^B = x_{B(i)}$ for $i = 1, 2, \ldots, m$. Thus the following relation holds:

$$\sum_{i=1}^{m} \mathbf{A}_{(\bullet, B(i))} {x^B}_i = \sum_{i=1}^{n} \mathbf{A}_{(\bullet, i)} x_i = \mathbf{Ax} = \mathbf{b}$$

The m columns $\mathbf{A}_{(\bullet, B(i))}$ with $i = 1, 2, \ldots, m$ are linearly independent and thus form a basis for $\mathbb{R}^m$. The vector $\mathbf{b}$ is a linear combination of this basis with the coefficients x_i^B with $i = 1, 2, \ldots, m$. Thus the system with this set of active constraints has a unique solution.

There are n linearly independent active constraints and thus is $\mathbf{x}$ a basic solution. □

Now we can construct a basic solution using the following recipe:

Recipe 3.21. Construct a basic solution $\mathbf{x}$ for linear programming problem in standard form:

1. Choose m linearly independent columns $\mathbf{A}_{(\bullet, B(i))}$ for $i = 1, 2, \ldots, m$ of $\mathbf{A}$
2. Set $x_i = 0 \; \forall i \notin I = \{B(1), B(2), \ldots, B(m)\}$
3. Solve the (to $m \times m$ reduced) system $\mathbf{Ax} = \mathbf{b}$ for the unknowns $x_{B(i)}$ with $i = 1, 2, \ldots, m$

Note: if the solution also satisfies $\mathbf{x} \geq \mathbf{0}$ then the basic solution $\mathbf{x}$ is also feasible.

Definition 3.22. From the previous discussion we summarize some concepts in the following definitions:

- The variables $x_{B(i)}$ with $i = 1, 2, \ldots, m$ are called **basic variables**. The remaining variables are **nonbasic**.
- The matrix $\mathbf{B} = \left[\mathbf{A}_{(\bullet,B(1))}, \mathbf{A}_{(\bullet,B(2))}, \ldots, \mathbf{A}_{(\bullet,B(m))}\right] \in \mathbb{R}^{m\times m}$ is called a **basic matrix** and its columns are called **basic columns**
- The vector $\mathbf{x^B}$ is the vector containing the values of the basic variables. Thus

$$\mathbf{B}\mathbf{x^B} = \mathbf{b} \quad \Rightarrow \quad \mathbf{x^B} = \mathbf{B}^{-1}\mathbf{b}.$$

Example 3.23. (continued, previous part Example 3.10)
In the previous example we have converted the linear programming problem in standard form. To reduce the notation of this problem we go further in matrix notation for the constraints:

$$\begin{array}{rl} \text{minimize} & f_{\mathbf{c}}(\mathbf{x}) = -10x_1 - 12x_2 - 12x_3 \\ \text{subject to} & \mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & 2 & 2 & 1 & 0 & 0 \\ 2 & 1 & 2 & 0 & 1 & 0 \\ 2 & 2 & 1 & 0 & 0 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 20 \\ 20 \\ 20 \end{bmatrix} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

Note that the last three columns of matrix $\mathbf{A}$ are linear independent columns. We can choose columns $\mathbf{A}_{(\bullet,4)}, \mathbf{A}_{(\bullet,5)}$ and $\mathbf{A}_{(\bullet,6)}$ as the basic columns for the basic matrix $\mathbf{B}$.

This results in a basic solution $\mathbf{x} = (0, 0, 0, 20, 20, 20)$. This solution also satisfies $\mathbf{x} \geq \mathbf{0}$ and is thus a basic feasible solution.

There are in general many possibilities to choose m linearly independent columns out of the n columns of a matrix $\mathbf{A}$. We need a method to find an optimal solution that minimizes $f_{\mathbf{c}}(\mathbf{x})$.

3.5 Simplex Method

3.5.1 General overview

From Section 3.1 about the general and standard linear programming problems we saw that all linear programming problems can be converted to a standard form. The only type of constraints on the non-negative variables x_i are equality constraints. To discuss the simplex method we will begin with an minimization example to show the most important ingredients we will discuss further in this section:

Example 3.24.

$$\begin{array}{rl} \text{minimize} & f(\mathbf{x}) = 2x_2 - 4x_3 \\ \text{subject to} & \begin{cases} 2 = x_1 + 6x_2 - x_3 \\ 8 = -3x_2 + 4x_3 + x_4 \end{cases} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

In this example we see that we have $N = 4$ variables and $M = 2$ equality constraints. We see that the variables x_2 and x_3 are used in the function $f(\mathbf{x})$ that we want to minimize. If we set these two values to zero we can use the other two constraints to compute the values for x_1 and x_4. If we set x_2 and x_3 to zero we get a feasible solution $\mathbf{x} = (2, 0, 0, 8)$ for this problem.

From the coefficients of the variables in the function $f(\mathbf{x})$ we see that we can decrease the value of the function by increasing the value of x_3. The coefficient of x_3 is -4 and a negative quantity of a larger non-negative variable will result in a lower value for the function $f(\mathbf{x})$.

We can change the second constraint to express x_3 in terms of x_4 and x_2. This gives us:

$$x_3 = 2 + \frac{3}{4}x_2 - \frac{1}{4}x_4$$

Note that we have chosen the second constraint to express x_3 in terms of other variables. That is because we want to increase the value of x_3 to a possible maximum but the first constraint gives no bound for a maximum value of x_3 (i.e. we can increase x_3 to all values we want and we do not violate any non-negativity constraint on x_1) If we substitute the new expression for x_3 into the function $f(\mathbf{x})$ we get:

$$\begin{aligned} f(\mathbf{x}) &= 2x_2 - 4x_3 \\ &= 2x_2 - 4\left[2 + \frac{3}{4}x_2 - \frac{1}{4}x_4\right] \\ &= -8 - x_2 + x_4 \end{aligned}$$

and for the other constraint:

$$\begin{aligned} 2 &= x_1 + 6x_2 - x_3 \\ &= x_1 + 6x_2 - 2 - \frac{3}{4}x_2 + \frac{1}{4}x_4 \\ &= -2 + x_1 + \frac{21}{4}x_2 + \frac{1}{4}x_4 \\ \iff & \\ 4 &= x_1 + \frac{21}{4}x_2 + \frac{1}{4}x_4 \end{aligned}$$

Next we can repeat this step to further minimize the value of $f(\mathbf{x})$. In this case we can do this by increasing the value of x_2.

For a compact notation we introduce a so called *tableau* in which the calculations we did before reduces to row and column operations. The tableau of the initial problem is:

Step 0		x_1	x_2	x_3	x_4
cost	0	0	2	-4	0
$x_1 =$	2	1	6	-1	0
$x_4 =$	8	0	-3	**4**	1

On the first line of the table we see the so called 'cost-row' for all variables x_i. The values in this row corresponds with the coefficients of the cost-function $f(\mathbf{x})$.

Below the 'cost-row' we see the rows that describe the constraints of the linear programming problem. As described earlier we found the solution $\mathbf{x} = (2, 0, 0, 8)$ as a $\mathbf{x}$ that satisfies the constraints. In the table we can read-off the negative cost of this solution in the 'cost'-row. The non-zero elements of $\mathbf{x}$ correspond with rows of the table. We can see that through neglecting the columns of the zero-valued elements, in this case x_2 and x_3. Then we see in the bottom-right block the identity matrix $\mathbf{I}$ and in the bottum-left block the vector $\mathbf{b}$ for the equation $\mathbf{I}\mathbf{x}^* = \mathbf{b}$. From that equation we can read off the values for $\mathbf{x}^*$. We can reconstruct the vector $\mathbf{x}$ by using the values of $\mathbf{x}^*$ for the non-zero elements and set the other elements of $\mathbf{x}$ to zero.

After the first substitution the tableau becomes:

Step 1		x_1	x_2	x_3	x_4
cost	8	0	-1	0	1
$x_1 =$	4	1	$\frac{21}{4}$	0	$\frac{1}{4}$
$x_3 =$	2	0	$-\frac{3}{4}$	1	$\frac{1}{4}$

The algebra of the previous section can also be seen as row operations on this table. From step 0 we came to step 1 using the row operations 'add one row to another row' and 'multiply one row with a constant'.

In a linear programming problem we want to minimize the cost function. In the tableau this means that we want to maximize the number in the top-left corner of the tableau. Therefore we add the third row ($x_4 = 8|\ldots$) once to the first row. Then the cost coefficient of x_3 in the programming problem reduces to zero.

Then we want to reconstruct the identity matrix for the columns x_1 and x_3. This can be done by dividing the third row by 4 (this gives the 1 on the diagonal of the identity matrix) and add the new third row to the second row (this gives the 0 on the element on the second row in column x_3).

The last tableau can be constructed using the same type of operations: add $\frac{4}{21}$ times the second row to the fist row (this makes the cost of x_2 zero), multiply the second row with $\frac{4}{21}$ (this makes the diagonal element of the identity matrix one) and add $\frac{3}{4}$ of the new second row to the third row.

This result in the following tableau:

Step 2		x_1	x_2	x_3	x_4
cost	$8\frac{16}{21}$	$\frac{4}{21}$	0	0	$1\frac{1}{21}$
$x_2 =$	$\frac{16}{21}$	$\frac{4}{21}$	1	0	$\frac{1}{21}$
$x_3 =$	$2\frac{4}{7}$	$\frac{1}{7}$	0	1	$\frac{2}{7}$

This is the last tableau because no element in the cost-row is negative and thus are there no possibilities to minimize $f(\mathbf{x}) = 2x_2 - 4x_3$ further and thus $\mathbf{x} = \left(0, \frac{16}{21}, \frac{18}{7}, 0\right)$ is the solution vector to the objective function.

3.5.2 Optimality

In this section we will take a closer look at the simplex method and the ideas behind it. Therefore we continue our discussion which brought us by a basic feasible solution.

Definition 3.25. Let $\mathbf{x} \in P$ where P is a polyhedron. A vector $\mathbf{d} \in \mathbb{R}^n$ is a **feasible direction** at $\mathbf{x}$ if $\exists \theta > 0$ such that $\mathbf{x} + \theta \mathbf{d} \in P$.

For basic variables we saw that a vector $\mathbf{x^B}$ corresponding to a vector $\mathbf{x}$ with the following relation holds:

$$\mathbf{x}^B = \mathbf{B}^{-1}\mathbf{b}$$

Let j be the index of a nonbasic variable x_j, thus $x_j = 0$. We want to increase the value of x_j to θ and do not violate the constraints $\mathbf{x} \geq \mathbf{0}$. This means that $d_j = 1$ and $d_i = 0$ for all nonbasic x_i where $i \neq j$. We can also define a vector $\mathbf{d^B} = (d_{B(1)}, d_{B(2)}, \ldots, d_{B(m)}) \in \mathbb{R}^m$ as the corresponding feasible direction vector for the basic feasible solution $\mathbf{x^B}$. To get a feasible solution we require:

$$\mathbf{A}(\mathbf{x} + \theta \mathbf{d}) = \mathbf{b}$$

The solution $\mathbf{x}$ is feasible thus $\mathbf{Ax} = \mathbf{b}$. Since $\theta > 0$ we need thus $\mathbf{Ad} = \mathbf{0}$. Thus

$$\mathbf{0} = \mathbf{Ad} = \sum_{i=1}^{n} \mathbf{A}_{(i,\bullet)} d_i = \sum_{i=1}^{m} \mathbf{A}_{(B(i),\bullet)} d_{B(i)} + \mathbf{A}_{(j,\bullet)}$$

The matrix $\mathbf{B}$ is a basic matrix and thus invertible. We can thus compute the vector $\mathbf{d^B}$ in the j^{th} basic direction:

$$\mathbf{d^B} = -\mathbf{B}^{-1}\mathbf{A}_{(j,\bullet)}$$

If a solution is optimized in the direction of $\mathbf{d}$ by construction we have required that the new solution satisfies the constraints $\mathbf{Ax} = \mathbf{b}$. We need only to check if the constraints $\mathbf{x} \geq 0$ are satisfied by the new (optimized) solution. The value $x_j = \theta > 0$ and all other nonbasic variables are unchanged, thus $x_i = 0\ \forall i$ nonbasic and $i \neq j$. For basic variables we have a constraint $\mathbf{x^B} \geq 0$ with two possibilities:

- We have a strict constraint $\mathbf{x^B} > \mathbf{0}$. Then $\exists \theta > 0$ such that $\mathbf{x^B} + \theta \mathbf{d^B} \geq \mathbf{0}$.
- We have one (or more) basic variable $x_k^B = 0$. If $d_k^B \geq 0$ then we have also $x_k^B + \theta d_k^B \geq 0$. But if $d_k^B < 0$ then $\forall \theta > 0 : x_k^B + \theta d_k^B < 0$ and thus violates the non negativity constraint.

If we want to see the effects of this optimization we compute the cost that we have reduced. We can compute the cost for solution $\mathbf{x}$ with the cost function $\mathbf{c} \cdot \mathbf{x}$. Thus the cost of the optimized solution is:

$$\mathbf{c} \cdot (\mathbf{x} + \theta \mathbf{d}) = \mathbf{c} \cdot \mathbf{x} + \mathbf{c} \cdot \theta \mathbf{d}$$

Note that $d_i = 0$ and $d_j = 1$ for i nonbasic and $i \neq j$ with j is the index of the basic direction in which $\mathbf{d^B}$ is computed. If we write $\mathbf{c^B}$ for the cost vector of the basic variables then the reduced cost per θ in the direction of $\mathbf{d}$ becomes:

$$\mathbf{c} \cdot \mathbf{d} = c_j + \mathbf{c^B} \cdot \mathbf{d^B} = c_j - \mathbf{c^B} \cdot \mathbf{B}^{-1}\mathbf{A}_{(j,\bullet)}$$

Definition 3.26. Let $\mathbf{x}$ be a basic solution, $\mathbf{B}$ the associated basic matrix and $\mathbf{c^B}$ the cost vector of the costs of the basic variables. $\forall j$ the **reduced cost** $\bar{c}_j$ of variable x_j can be computed as follows:

$$\bar{c}_j = c_j - \mathbf{c^B} \cdot \mathbf{B}^{-1}\mathbf{A}_{(j,\bullet)}$$

3.5.3 Method

This leads us to the following recipe to compute the optimal solution:

Recipe 3.27. Simplex method for linear programming problem in standard form:

1. Compute an initial basic feasible solution $\mathbf{x}_0$
2. Pick a nonbasic variable x_i with a negative cost c_i, this variable will inter the basis.
3. Pick a basic variable $x_B(j)$ that leaves the basis.
4. Update tableau for this change of basis.
5. Terminate if $\mathbf{c} \geq 0$.

Example 3.28. (continued, previous part Example 3.23)
In this example we will execute the simplex method and see how it works. As noted in the previous example will we begin with the variables x_4, x_5 and x_6 as basic variables. This results in a basic feasible solution $\mathbf{x} = (0, 0, 0, 20, 20, 20)$. The first simplex method tableau is then:

Step 0		x_1	x_2	x_3	x_4	x_5	x_6	$\frac{x_{B(i)}}{u_i^1}$
$\mathbf{c}$	0	-10	-12	-12	0	0	0	
$x_4 =$	20	1	2	2	1	0	0	20
$x_5 =$	20	**2**	1	2	0	1	0	10
$x_6 =$	20	2	2	1	0	0	1	10

Now we see that three variables x_1, x_2 and x_3 have negative costs c_i. We need to pick some variable from this three and choose x_1. Then we compute the value $\frac{x_{B(i)}}{u_i^1}$ where the vector $\mathbf{u}^j$ is the data column below x_j. Using this computed quantity $\frac{x_{B(i)}}{u_i^1}$ we can pick a $x_{B(i)}$ that leaves the basis. We chooses the $x_{B(i)}$ that minimizes $\frac{x_{B(i)}}{u_i^1}$. In this case x_5. This gives us a pivot row and a pivot column. The pivot cell is marked in boldface.

Now we apply row operations on the tableau such that

- an identity matrix relation exists between the basis variables on the left-side of the tableau and the corresponding variables on to top-side.
- the cost for the variable that enters the basis is zero.

We do this by first adding the pivot row $\frac{10}{2} = 5$ times to the cost row. Then we divide the pivot row by the value of the pivot cell (thus by 2) to get the value 1 at the pivot cell. Next we add or subtract a multiple of the pivot row to the other rows to get zeros at the other cells in the pivot column. We say that x_1 enters the basis and x_5 leaves the bases.

After applying this row operations to the tableau we came to the next step:

Step 1		x_1	x_2	x_3	x_4	x_5	x_6	$\frac{x_{B(i)}}{u_i^2}$
$\mathbf{c}$	100	0	-7	-2	0	5	0	
$x_4 =$	10	0	$\mathbf{1\frac{1}{2}}$	1	1	$-\frac{1}{2}$	0	$6\frac{2}{3}$
$x_1 =$	10	1	$\frac{1}{2}$	1	0	$\frac{1}{2}$	0	20
$x_6 =$	0	0	1	-1	0	-1	1	0

We can read the values of the basic variables of the solution on the left-most column. The variables that are not in the left-most column are zero. Thus after this iteration we have $\mathbf{x} = (10, 0, 0, 10, 0, 10)$.

We see that there are variables with negative costs so we repeat the previous instruction. This results in the step: x_2 enters the basis and x_6 leaves the basis according to the described minimum criterion.

Step 2		x_1	x_2	x_3	x_4	x_5	x_6	$\frac{x_{B(i)}}{u_i^3}$
$\mathbf{c}$	100	0	0	-9	0	-2	7	
$x_4 =$	10	0	0	$\mathbf{2\frac{1}{2}}$	1	1	$-1\frac{1}{2}$	4
$x_1 =$	10	1	0	$1\frac{1}{2}$	0	1	0	$6\frac{2}{3}$
$x_2 =$	0	0	1	-1	0	-1	1	-

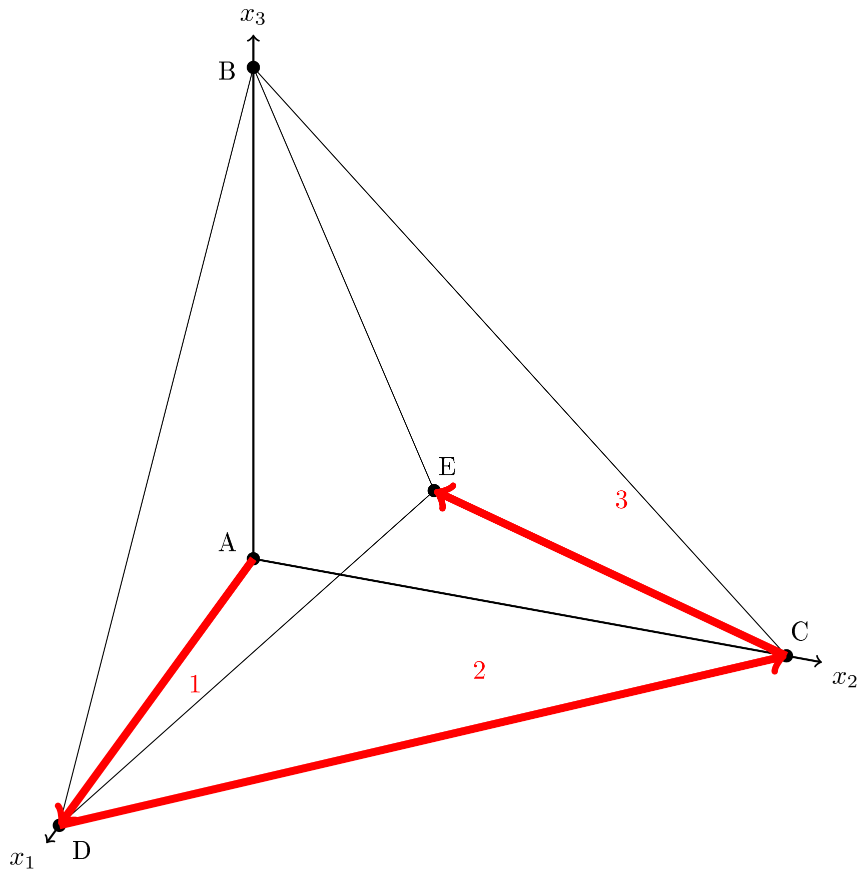


Figure 8: Sketch of the feasible set from Example 3.28 in (x_1, x_2, x_3)-space

This step did not decrease the cost but still we can continue to find a further optimization of this linear programming problem. The current solution is: $\mathbf{x} = (10, 0, 0, 10, 0, 0)$. For the next step x_3 enters the basis. When we look for the $\mathbf{u}$ vector we see that $u_3 < 0$. We do not allow this negative sign because that could result in a non-feasible solution. Thus we decide that x_4 leaves the basis.

Step 3		x_1	x_2	x_3	x_4	x_5	x_6
$\mathbf{c}$	136	0	0	0	$3\frac{3}{5}$	$1\frac{3}{5}$	$1\frac{3}{5}$
$x_3 =$	4	0	0	1	$\frac{2}{5}$	$\frac{2}{5}$	$-\frac{3}{5}$
$x_1 =$	4	1	0	0	$-\frac{3}{5}$	$\frac{2}{5}$	$\frac{2}{5}$
$x_2 =$	4	0	1	0	$\frac{2}{5}$	$-\frac{3}{5}$	$\frac{2}{5}$

Now we see that all costs are positive and thus we have reached the optimal feasible solution which is: $\mathbf{x} = (4, 4, 4, 0, 0, 0)$. Note that the last three variables x_4, x_5 and x_6 are slack variables that we have introduced. When de constraints $\mathbf{Ax} \leq \mathbf{b}$ describes a bounded set of resources, then the slack variables can be seen as some "rest consuming" variables that consumes resources but did not contribute in decreasing of the cost function. Then it looks intuitive that an optimal solution do not uses this "rest consuming" variables.

Geometrically we can see the feasible set as a simplex. The feasible set looks in (x_1, x_2, x_3)-space as a tetrahedron because we had three constraints that are active at each time. Then the simplex method searches for an optimal point at the corners of the feasible set. The simplex method found corner E by traveling from $A = (0, 0, 0)$ via $D = (10, 0, 0)$ and $C = (10, 10, 0)$ to $E = (4, 4, 4)$, as shown in Figure 8.

4 General methods for solving linear systems

4.1 Algorithm complexity

Before we can do some analysis on algorithms we need to introduce methodology to describe the complexity of an algorithm. Experiments can be useful but have also some limitations such as a limited set of input [11]. Therefore there are some rules that results in an analytic framework:

- All possible inputs are taken into account.
- The framework allows an evaluation of the efficiency of an algorithm, independent from the hardware or software environment.
- Can be performed by study the algorithm at a high level and does not require an implementation of the algorithm.

For some algorithms it is possible to count the primitive operations executed by the algorithm. However, if the complexity of the algorithm increases in most cases the absolute number of operations can not be computed. Therefore the Bigg-$\mathcal{O}$ notation is introduced. The Bigg-$\mathcal{O}$ notation gives a description of the asymptotic behavior for large input.

Definition 4.1. Let $f(n)$ and $g(n) : \mathbb{N}^+ \to \mathbb{R}$. $f(n)$ is $\mathcal{O}(g(n))$ if $\exists c > 0$ and an integer $N \geq 1$ such that:

$$f(n) \leq cg(n) \quad \forall n \geq N$$

Thus when the input is large enough the run time is bounded by a fixed multiple of $g(n)$.

Example 4.2. When the primitive operations of algorithm $\mathcal{A}$ can be computed as:

$$f(n) = 3n^3 + n^2 + 10$$

then algorithm $\mathcal{A}$ is $\mathcal{O}(n^3)$.

4.2 Gaussian elimination

From the discussion in the previous sections we want to find methods to solve the problem $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$. From linear algebra we know already the *Gaussian elimination* where we apply three types of row operations on the rows of $\mathbf{A}$ to get the matrix $\mathbf{A}$ in *row echelon form* or *reduced row echelon form.* The three types of row operations are 'exchange rows', 'add a scalar multiple of one row to another row' and 'multiply a row by a non-zero constant'.

More details about this direct method for solving linear systems can be found in [4]. In this book Burden and Faires computed also that the Gaussian elimination an algorithm is with computations in $\mathcal{O}(n^3)$. Intuitively this can be seen through the following reasoning. Assume matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is approximately a square matrix so $\mathcal{O}(n)$ =$\mathcal{O}(m)$. The Gaussian elimination needs to reduce for all n columns the n entries below the diagonal entry to zero. This can be done using addition of a scalar constant of one row to another row, which is also $\mathcal{O}(n)$. This gives a computational complexity of $\mathcal{O}(n^3)$.

4.3 Jacobi and Gauss-Seidel iterative method

If the dimensions of matrix $\mathbf{A}$ increases then the Gaussian elimination is less interesting due to its computational complexity of $\mathcal{O}(n^3)$. Therefore we introduce some iterative methods. In this iterative methods we approximate the solution of a system $\mathbf{Ax} = \mathbf{b}$. For large spare systems the iterative methods becomes efficient in computation and also storage [4].

Iterative methods for solving $\mathbf{Ax} = \mathbf{b}$ are based on the idea that we can convert that system to another system for a fixed $\mathbf{T}$ and $\mathbf{c}$ as follows:

$$\mathbf{x} = \mathbf{Tx} + \mathbf{c}.$$

Iterative methods come with an initial approximation $\mathbf{x}^{(0)}$ of the solution $\mathbf{x}$. By iteratively computing new approximations $\mathbf{x}^{(k)}$ this class of methods will result in a sequence of vectors $\{\mathbf{x}^{(k)}\}_{k=0}^{\infty}$ with the property $\lim\limits_{k\to\infty} \{\mathbf{x}^{(k)}\} = \mathbf{x}$.

Example 4.3. Given the linear system $\mathbf{Ax} = \mathbf{b}$:

$$\mathbf{Ax} = \begin{bmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 6 \\ 25 \\ -11 \\ 15 \end{bmatrix}$$

We can rewrite the system in the form $\mathbf{x} = \mathbf{Tx} + \mathbf{c}$ by using row i to express x_i in terms of the other rows. This results into:

$$\mathbf{x} = \mathbf{Tx} + \mathbf{c} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} & \frac{1}{10} & -\frac{1}{5} & 0 \\ \frac{1}{11} & & \frac{1}{11} & -\frac{3}{11} \\ -\frac{1}{5} & \frac{1}{10} & & \frac{1}{10} \\ 0 & -\frac{3}{8} & \frac{1}{8} & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} \frac{3}{5} \\ \frac{25}{11} \\ -\frac{11}{10} \\ \frac{15}{8} \end{bmatrix}$$

where we can read off the matrix $\mathbf{T}$ and the vector $\mathbf{c}$.

To solve the system $\mathbf{Ax} = \mathbf{b}$ we can introduce a random vector $\mathbf{x}^{(0)}$ and apply iteratively:

$$\mathbf{x}^{(k+1)} = \mathbf{Tx}^{(k)} + \mathbf{c}. \tag{2}$$

The method of rewriting a system $\mathbf{Ax} = \mathbf{b}$ into a system $\mathbf{x} = \mathbf{Tx} + \mathbf{c}$ and compute $\mathbf{x}^{(k)}$ is called the *Jacobi iterative method.*

This method can be improved when we note that $\mathbf{T} = \mathbf{L} + \mathbf{U}$, where $\mathbf{L}$ and $\mathbf{U}$ are strictly lower and upper triangular matrices. Another observation we made is that we divided row i of $\mathbf{A}$ by the value of $-a_{ii}$ to get the values of row i of $\mathbf{T}$ in Example 4.3. The diagonal elements becomes -1 and goes to the other side of the equality sign.

Thus we can $\mathbf{Ax} = \mathbf{b}$ also transform into another form, namely $\mathbf{Dx} = (\mathbf{L} + \mathbf{U})\mathbf{x} + \mathbf{b}$ using $\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}$. Thus if $\det \mathbf{D} \neq 0$ then the system becomes:

$$\mathbf{x} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x} + \mathbf{D}^{-1}\mathbf{b}.$$

For the non-zero elements l_{ij} of $\mathbf{L}$ we have that $i > j$. For the computation of $x_j^{(k+1)}$ we need a value for x_i. Assumed that $\mathbf{x}^{(k)}$ for $k \to \infty$ converges to $\mathbf{x}$ the already computed value $x_i^{(k+1)}$ is a better approximation for x_i then $x_i^{(k)}$. This improvements brings us to the *Gauss-Seidel iterative method*:

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\mathbf{Lx}^{(k+1)} + \mathbf{D}^{-1}\mathbf{Ux}^{(k)} + \mathbf{D}^{-1}\mathbf{b}. \tag{3}$$

For the details and analysis of both the Jacobi and the Gauss-Seidel iterative method we refer to [4].

4.4 Relaxation

Definition 4.4. Suppose $\bar{\mathbf{x}} \in \mathbb{R}^n$ is an approximation of the solution of the linear system $\mathbf{Ax} = \mathbf{b}$. The **residual vector** for $\bar{\mathbf{x}}$ with respect to this system is given by:

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\bar{\mathbf{x}}$$

Let $\mathbf{r}_i^{(k)}$ be the residual vector associated with $\mathbf{x}_i^{(k)}$, where the first i components of $\mathbf{x}_i^{(k)}$ are form $\mathbf{x}^{(k)}$ and the other components correspond to the components of $\mathbf{x}^{(k-1)}$. Thus $\mathbf{x}_i^{(k)}$ contains the appropriated solution for the system after $k-1$ full iterations and $i-1$ row updates for the next k^{th} iteration.

Thus the m^{th} component of $\mathbf{r}_i^{(k)}$ is:

$$\begin{aligned} \left(\mathbf{r}_i^{(k)}\right)_m &= b_m - \sum_{j=1}^{i-1} a_{mj} x_j^{(k)} - \sum_{j=i}^{n} a_{mj} x_j^{(k-1)} \\ &= b_m - \sum_{j=1}^{i-1} a_{mj} x_j^{(k)} - \sum_{j=i+1}^{n} a_{mj} x_j^{(k-1)} - a_{mi} x_i^{(k-1)} \end{aligned}$$

Thus for the i^{th} component of $\mathbf{r}_i^{(k+1)}$ we have:

$$\left(\mathbf{r}_i^{(k+1)}\right)_i = b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} - a_{ii} x_i^{(k)}$$

This gives us

$$a_{ii}x_i^{(k-1)} + \left(\mathbf{r}_i^{(k)}\right)_i = b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)} \tag{4}$$

From Equation 3 we saw that:

$$x_i^{(k+1)} = -\frac{1}{a_{ii}}\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \frac{1}{a_{ii}}\sum_{j=i+1}^{n} a_{ij}x_j^{(k)} + \frac{1}{a_{ii}}b_i.$$

Thus Equation 4 can be rewritten as:

$$a_{ii}x_i^{(k)} + \left(\mathbf{r}_i^{(k+1)}\right)_i = a_{ii}x_i^{(k+1)}$$
$$\Updownarrow$$
$$x_i^{(k+1)} = x_i^{(k)} + \frac{\left(\mathbf{r}_i^{(k+1)}\right)_i}{a_{ii}}$$

On the other hand we have for the i^{th} component of $\mathbf{r}_{i+1}^{(k+1)}$ and using the expression for $x_i^{(k+1)}$:

$$\begin{aligned}\left(\mathbf{r}_{i+1}^{(k+1)}\right)_i &= b_i - \sum_{j=1}^{i} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} \\ &= b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} - a_{ii}x_i^{(k+1)} \\ &= b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} + \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} - b_i \\ &= 0.\end{aligned}$$

Thus the Gauss-Seidel iterative method chooses $x_i^{(k+1)}$ in such a way that $\left(\mathbf{r}_{i+1}^{(k+1)}\right)_i = 0$.

If we want to minimize $\left\|\mathbf{r}_{i+1}^{(k+1)}\right\|$ it is not always true that this condition results in the fastest convergence. Therefore we can introduce a relaxation parameter $\lambda > 0$ and modify the iteration schema to:

$$x_i^{(k+1)} = x_i^{(k)} + \lambda\frac{\left(\mathbf{r}_i^{(k+1)}\right)_i}{a_{ii}}$$

The notion of relaxation can be generalized to other methods which result in a new class of methods called *relaxation methods.* When $0 < \lambda < 1$ the method is an *under-relaxation method* and if $\lambda > 1$ then the method is an *over-relaxation method.*

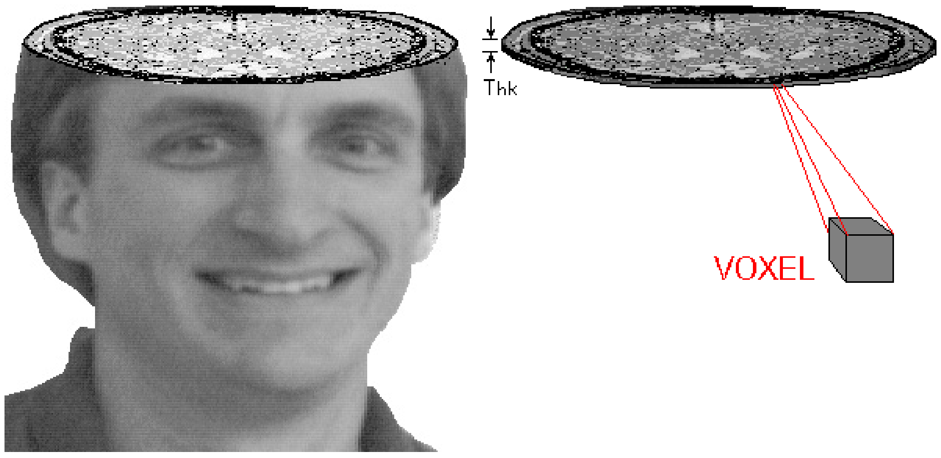


Figure 9: An object, and also a person, can be seen as a stack of thin slices that are built up of voxels [13].

5 Computer Tomography

Tomography refers to cross sectional imaging of an object from transmission or reflection data [14]. Fundamentally, tomographic imaging deals with reconstruction of an image from its projections. The projections are from an unknown function of two variables with has real and non-negative values [6].

For computer tomography an object can be seen as a stack of thin slices, see Figure 9. If we put a grid on one of the slices then we get small blocks as (minimal) volumes that can be sampled from the object we look at. The small volumes are called *voxels* [13]. If we want to see a picture of this slice, the slice is converted from a 3D volume to a 2D picture. If we put a Cartesian grid on the 2D picture we get small square points of picture-elements with a specified color. The points are called *pixels*. We see thus that voxels and pixels are related in a sense that a pixel shows a color on an image that represents a (physical of biological) property of a voxel as part of an object.

The projections can be seen as line integrals of some parameter of the object along that line. In the next part of this thesis we will focus on diagnostic medicine applications. In the context of diagnostic medicine a typical example of a property we visualize is the attenuation of X-rays propagating through a biological tissue. Also other phenomena such as response to magnetic resonance, ultrasound or radioisotopes can be used to derive some information from a biological tissue.

A projection is formed by a combination of line integrals. The different lines can be taken as parallel lines to get a parallel projection. This can for example be done by moving an X-ray source and a detector along parallel lines on opposite sides of the object we want to measure. An example of a parallel projection is shown in Figure 10.

It is also possible to use a single source and a line of detectors. In that case we get a fan beam projection. An example is shown in Figure 11. In this thesis we will use the parallel projections in our computations.

5.1 Shepp Logan Phantom

In the context of diagnostic medicine the algorithms used for computer tomography must reconstruct images of a special structure. To compare algorithms there exists a standard input that models a head. This ‘head phantom’ is defined as a set of ellipses with some properties. The advantage of ellipses in the context of projections is that the projection of an ellipse can be computed analytically. The ‘head phantom’ from Shepp and Logan [14, 15] defines a number of ellipses and gray intensities, as shown in the Table 1.

If we plot the ‘head phantom’ and stretch the gray values between 0.9 and 1.1 to the full range of 0 to 255 we get a picture like Figure 12.

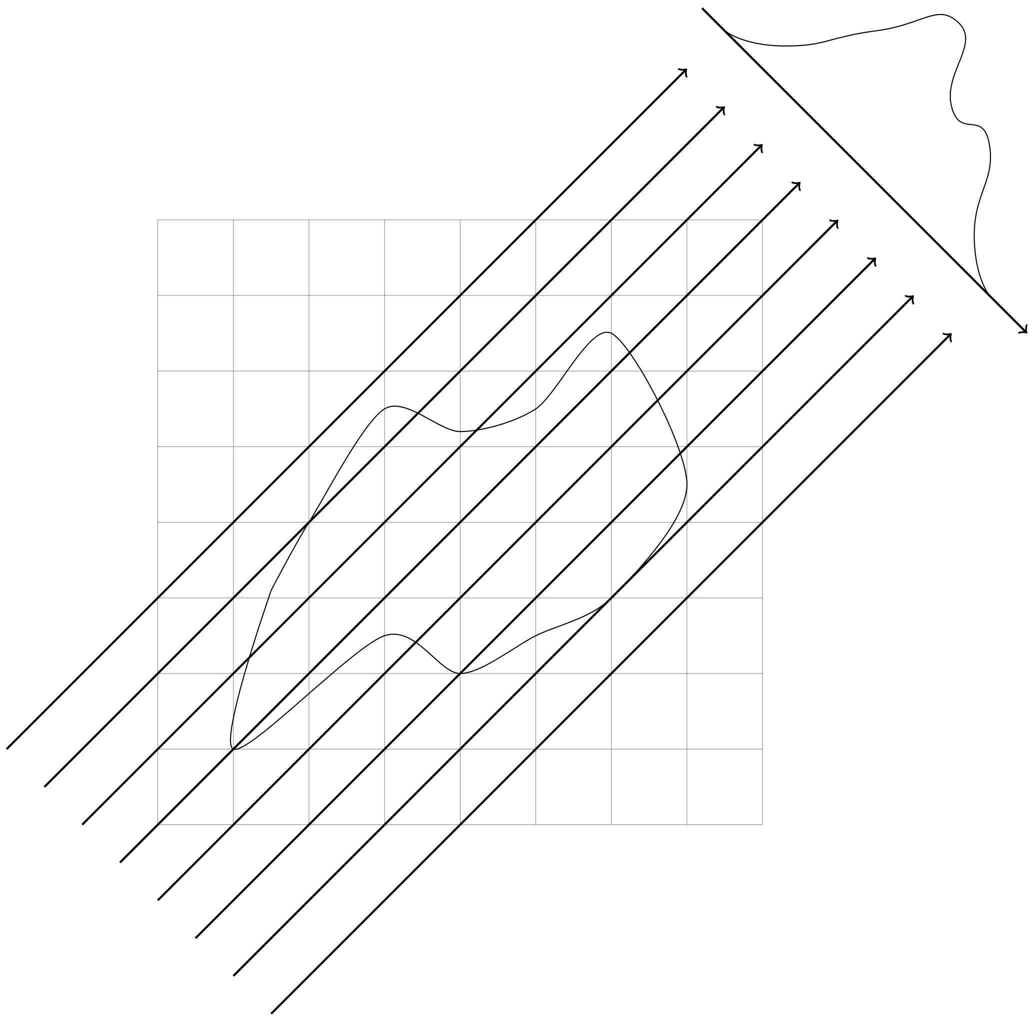


Figure 10: Parallel projection of a given angle (eg. $\theta = \frac{1}{2}\pi$)

Center of ellipse	Major axis	Minor axis	Rotation angle	Gray value
$(0, 0)$	0.92	0.69	90	2.00
$(0, -0.0184)$	0.874	0.6624	90	-0.98
$(0.22, 0)$	0.31	0.11	72	-0.02
$(-0.22, 0)$	0.41	0.16	108	-0.02
$(0, 0.35)$	0.25	0.21	90	0.01
$(0, 0.1)$	0.046	0.046	0	0.01
$(0, -0.1)$	0.046	0.046	0	0.01
$(-0.08, -0.605)$	0.046	0.023	0	0.01
$(0, -0.605)$	0.023	0.023	0	0.01
$(0.06, -0.605)$	0.046	0.023	90	0.01

Table 1: Definition of ellipses of the Shepp Logan Phantom

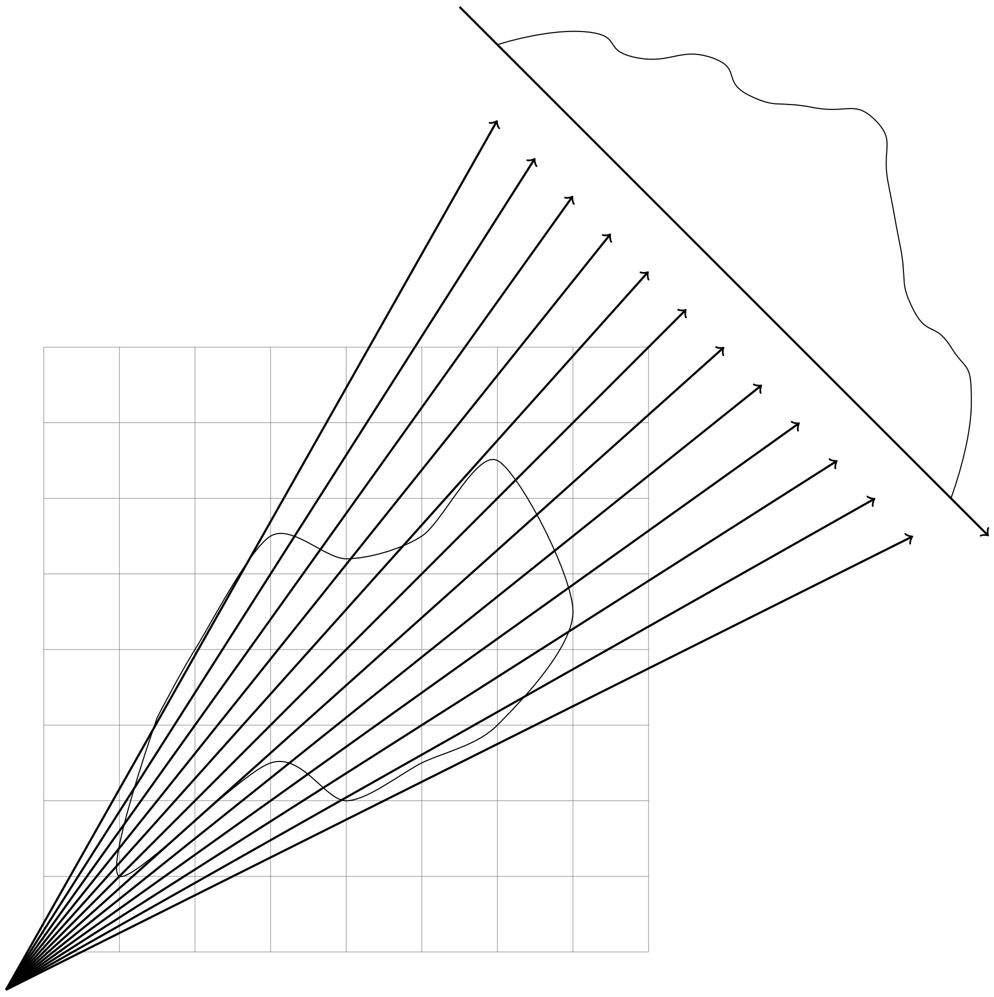


Figure 11: Fan beam projection

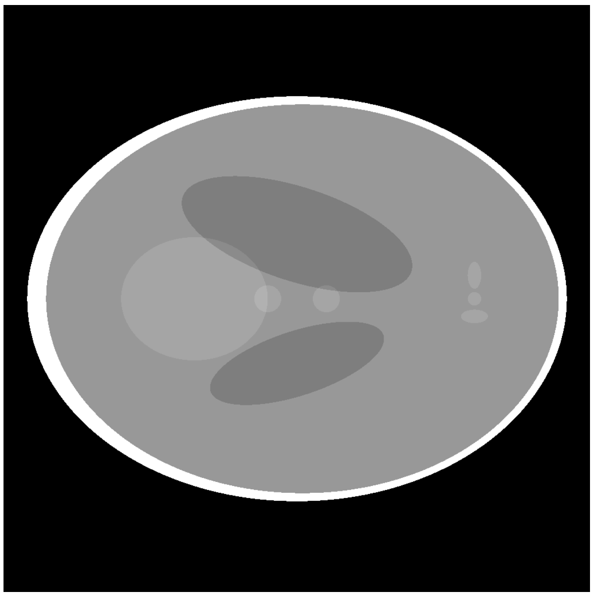


Figure 12: The Shepp Logan phantom with gray intensities 0.9 to 1.1 stretched to the full gray-range

5.2 Discretization of the model for transmission tomography

With the analytic definition of the Shepp Logan Phantom it is possible to compute the projections analytically and do the discretization a step later. If we want to use a computer for the tomography at some point we need to discretize the model. In this project we choose to discretize the input of the algorithm also instead of using the analytic expressions for the head phantom. This is done because this results in a program that can process the algorithm on all square images in stead of only the Shepp Logan phantom.

The discretization of the model leads to a finite dimensional linear algebra and so also to the optimization theory we discussed earlier [6].

If we use again the notation we stated already in the introduction of this thesis, we are looking for the inverse of operator $\mathcal{O}$ in the following equation:

$$\mathbf{y} = \mathcal{O}\mathbf{x}$$

In the context of diagnostic medicine this operator $\mathcal{O}$ is for example the projection with x-rays. The measured data from the detectors of Figure 10 is stored in $\mathbf{y}$. The values of this vector represent line integrals and because we have discretized the original object to the *image vector* $\mathbf{x} \in \mathbb{R}^n$ we can build a *projection matrix* $\mathbf{A} \in \mathbb{R}^{m\times n}$ such that the line integrals can be described as finite sums:

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

where $\mathbf{y} \in \mathbb{R}^m$ is the *measurements vector* with m measurements.

Thus summarizing the model we have built for the transmission tomography we have the following vectors:

- The vector $\mathbf{x} \in \mathbb{R}^n$ that contains the pixels of the image representing the object we want to see.
- The vector $\mathbf{y} \in \mathbb{R}^m$ that contains the data measured with the detectors. When we have d detectors and project on a angles then we have $m = ad$.
- The matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ that contains which pixels are present in a projection ray and it could eventual also store different weights for different pixels.

for which the following equation holds:

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

where $\mathbf{y}$ and $\mathbf{A}$ are known and $\mathbf{x}$ is unknown and we want to know $\mathbf{x}$. In the next section we will look at a method to come to the inversion problem so that we can reconstruct the image from the projection data.

6 Row-action method

In the context of linear programming problems for computer tomography we have some additional problems to solve $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$ where $\mathbf{A} \in \mathbb{R}^{m \times n}$. Therefore we want some other iterative methods then the general methods like Jacobi or Gauss-Seidel iterative methods. The problems in computer tomography are often undetermined, due to a lack of information, or over-determined [5]. When the system is over-determined we have a high change of a self-contradicting system. The systems are also huge systems to get a better quality image. A computer tomography ray hits only a couple of all pixels and thus the system is also sparse.

Given that the input of the system is often measured data, we have also inaccuracy in measurements. Therefore we want an iterative method that can deal with the noise in the data.

There are multiple possibilities do deal with inconsistency of systems of this form. Some approaches to deal with it are:

1. The feasibility approach; In this approach we seek $\mathbf{x}$ in a neighborhood of hyperplanes defined by $\mathbf{Ax} = \mathbf{b}$. To define a neighborhood we introduce a tolerance vector $\alpha \in \mathbb{R}^m$, such that the solution $\mathbf{x}$ must satisfy: $\mathbf{b} - \alpha \leq \mathbf{Ax} \leq \mathbf{b} + \alpha$.

2. The optimization approach; If a linear programming problem does not have a unique solution then we can reduce the solution space by applying an extra objective function $g(x) : \mathbb{R}^m \to \mathbb{R}$ on the solutions of the first linear programming problem. Another or additional approach is to use box constraints for $\mathbf{x}$ such as $\mathbf{w} \leq \mathbf{x} \leq \mathbf{v}$

3. The regularized approach; In this approach we solve $\mathbf{Ax} = \mathbf{b}$ subject to $\mathbf{x} \in Q$ where Q describes some extra constraints. This leads to minimize $f(\mathbf{x}) + rg(\mathbf{Ax} - \mathbf{b})$ subject to $\mathbf{x} \in Q$, where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^m \to \mathbb{R}$. An often used example is the least square regularization.

Definition 6.1. A **row-action method** is an iterative algorithm with the following properties:

- No changes are made to the original matrix.
- No operations are performed on the matrix as whole.
- A single iteration step uses only one row of matrix $\mathbf{A}$.
- In a single iteration step where $\mathbf{x}^{(k+1)}$ is calculated only the direct predecessor $\mathbf{x}^{(k)}$ is needed.

Example 6.2. Note that the previously described iterative methods, namely the Jacobi and the Gauss-Seidel method are not row-action methods because in each iteration function f in $\mathbf{x}^{(k+1)} = f(\mathbf{x}^{(k)})$ (see Equation 2 and also Equation 3) we multiply the input with a whole matrix. This violates the second requirement of Definition 6.1.

Definition 6.1 states explicitly that in one iteration step only one row of the matrix $\mathbf{A}$ may be used. Therefore we have the freedom to choose an order of rows on which we want to iterate the row-action method. The order on which we iterate over the rows of matrix $\mathbf{A}$ is described by the so called control sequence:

Definition 6.3. A sequence of indices $\{i_k\}_{k=0}^{\infty}$ according to the rows of matrix $\mathbf{A}$ is called a **control sequence** of a row-action method. That is, in the iteration from k to $k+1$ the i_kth row of $\mathbf{A}$ is used.

Example 6.4. Examples of control sequences are:

- **Cyclic control**, where $i_k = k \textbf{ mod } m + 1$.
- **Almost cyclic control**, where $i_k \in \{1, 2, \ldots, m\}$ $\forall k$ and $\exists C$ such that $\forall k \geq 0 : M \subseteq \{i_{k+1}, i_{k+2}, \ldots, i_{k+C}\}$. Thus at iteration i_k, after C steps the algorithm has at least once iterated over all rows of the system $\mathbf{Ax} = \mathbf{b}$.
- **Remotest set control**, i_k is determined such that $d(\mathbf{x}^{(k)}, Q_{i_k}) = \max_{i \in M} d(\mathbf{x}^{(k)}, Q_i)$. Thus the next row to iterate is the row that has the largest distance to the constraints.
- **Most violating constraint control**, i_{k+1} is determined by the constraint most violated by iteration $x^{(k)}$. i_{k+1} is the row on which the maximum distance is attained between $f_i(x^{(k)})$ and b_i.

6.1 Kaczmarz algorithm - ART

An example of a row-action method is the Kaczmarz algorithm [5, 6] or ART (Algebraic Reconstruction Technique). Assuming that a solution to $\mathbf{Ax} = \mathbf{b}$ exists the Kaczmarz algorithm will find a solution to the class of problems $\mathbf{Ax} = \mathbf{b}$. The algorithm is initialized with an arbitrary solution $\mathbf{x} \in \mathbb{R}^n$ and in each iteration the following computation is done:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \lambda_k \frac{b_{i_k} - \mathbf{A}_{(i_k,\bullet)} \cdot \mathbf{x}^{(k)}}{\left\|\mathbf{A}_{(i_k,\bullet)}\right\|^2} \mathbf{A}_{(i_k,\bullet)} \tag{5}$$

6.1.1 Geometrically interpretation

Geometrically the Kaczmarz algorithm projects in each step the current solution $\mathbf{x}^{(k)}$ on a hyperplane described by one of the m constraints in $\mathbf{Ax} = \mathbf{b}$. Assumed that they intersect in a point (which is required with the constraint that a solution must exists) this will lead to a sequence of solutions that converges to a solution that satisfies all the constraints at the same time.

6.1.2 Kaczmarz in diagnostic medicine

If we apply this algorithm to an image reconstruction problem in the context of diagnostic medicine we can make the equation $\mathbf{Ax} = \mathbf{b}$ more concrete. Following the discretized model for image reconstruction in Section 5.2 we can interpret the matrix $\mathbf{A}$ as a projection matrix. The rows of the projection matrix describe which pixels contributes to a given projection and with what weight.

The vector $\mathbf{x}$ as the unknown solution to this problem can be seen as the image of the object that we want to reconstruct. The vector $\mathbf{b}$ describes the measured data from the different projections.

Thus when we iterate in the Kaczmarz algorithm over the rows of matrix $\mathbf{A}$ we iterate over the projections that are taken from the object by the CT-scanner. In each iteration the algorithm changes the values on the entries of the image vector $\mathbf{x}$. For the next discussion we neglect relaxation, thus we can set the relaxation parameters to one: $\lambda_k = 1 \; \forall k$.

The change of the value of $\mathbf{x}$ in iteration i is done in such a way that $\mathbf{x}^{(k+1)}$ satisfies the constraint described by row i_k of the equation $\mathbf{Ax} = \mathbf{b}$. This can be done by adding a vector to $\mathbf{x}^{(k)}$ with data in the same proportion as the pixels contribute to the projection of that ray. Thus the vector we add to $\mathbf{x}^{(k)}$ is a multiple of the vector $\mathbf{A}_{(i_k,\bullet)}$. The difference between the projection data from the CT-scanner and the projection data that we could expect based on $\mathbf{x}^{(k)}$ can be computed by the dot product between the projection row and the image vector $\mathbf{x}^{(k)}$:

$$\delta_{i_k} = b_{i_k} - \mathbf{A}_{(i_k,\bullet)} \cdot \mathbf{x}^{(k)}$$

Combining this δ_{i_k} difference with the normalized projection row i_k of matrix $\mathbf{A}$, the vector $\frac{\mathbf{A}_{(i_k,\bullet)}}{\left\|\mathbf{A}_{(i_k,\bullet)}\right\|^2}$, we get the difference vector we add to in iteration $k+1$ to $\mathbf{x}^{(k)}$ to get $\mathbf{x}^{(k+1)}$, as described in Equation 5.

6.1.3 Convergence

In this thesis we do not treat the convergence of the Kaczmarz algorithm. However, we want to state something about the convergence. In the context of image reconstruction for medical imaging the existence of a solution is assumed, because the patient in the CT-scanner exists and the CT-scanner computes projections of this patient.

A bigger problem for image reconstruction is the uniqueness of a solution. In most cases the number of constraints is less than the number of pixels we want to see, so the system is under-determined. Also in this cases the Kaczmarz algorithm will find a solution and the following theorem and corollary state something about additional properties and a constraint of the solution of the Kaczmarz algorithm:

Theorem 6.5. Let

$$\lim_{k\to\infty} \sup |1 - \lambda_k| < 1.$$

If the system $\mathbf{Ax} = \mathbf{b}$ is consistent then the Kaczmarz algorithm described by Equation 5 converges to a solution of $\mathbf{Ax} = \mathbf{b}$. Moreover if $\mathbf{x}^{(0)}$ is in the range of $\mathbf{A}^T$ then the sequence $\left\{\mathbf{x}^{(k)}\right\}_{k=0}^{\infty}$ converges to the minimum-norm solution, that is:

$$\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{A}^{+}\mathbf{b}.$$

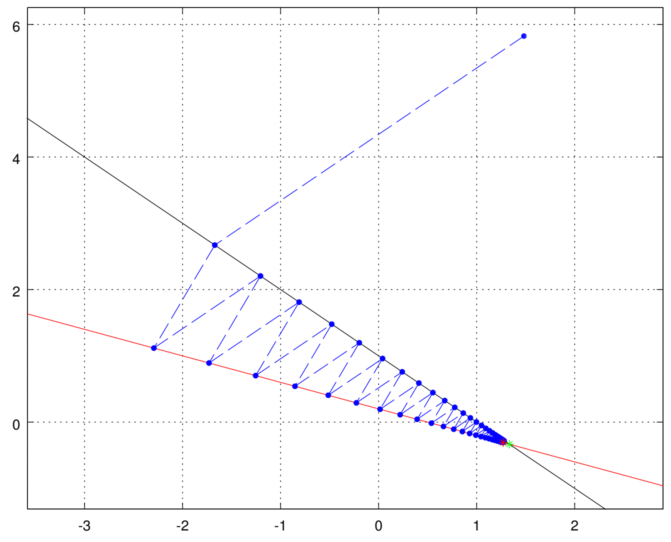


Figure 13: Plot of an execution of the Kaczmarz algorithm in a 2D space of Example 6.7

Proof. This theorem is treated by Reference [10] and references to proofs can be found there. □

Corollary 6.6. If the system $\mathbf{Ax} = \mathbf{b}$ has not a unique solution then the Kaczmarz algorithm finds the solution $\mathbf{x}$ that satisfies the linear programming problem:

$$\begin{aligned} \text{minimize} \quad & \|\mathbf{x}\| \\ \text{subject to} \quad & \mathbf{Ax} = \mathbf{b} \end{aligned}$$

Proof. By the previous theorem the Kaczmarz algorithm finds the minimum-norm solution $\mathbf{x}$ satisfying $\mathbf{Ax} = \mathbf{b}$. Thus $\forall \mathbf{y} \in \mathbb{R}^n : \|\mathbf{x}\| \leq \|\mathbf{y}\|$. This is the definition of a minimization problem. □

6.1.4 Interpretation of the algorithm

After a general description of the Kaczmarz algorithm and the geometry behind it we will look at an example. From this example we can see the comments we made earlier about the Kaczmarz algorithm.

Example 6.7. In this example we will give an intuitive idea about the Kaczmarz method and the geometry behind it. Figure 13 plots the iterations of the Kaczmarz algorithm for the following system with an arbitrary initial guess $\mathbf{x}^{(0)}$:

$$\mathbf{Ax} = \mathbf{b} = \begin{bmatrix} 1 & 1 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

We see the two constraint lines:

$$\begin{aligned} x_1 + x_2 = 1 \Leftrightarrow x_1 &= -x_2 + 1 \qquad \text{(black line)} \\ 2x_1 + 5x_2 = 1 \Leftrightarrow x_1 &= -\frac{5}{2}x_2 + \frac{1}{2} \qquad \text{(red line)} \end{aligned}$$

and the solution is alternating between the one and the other constraint.

In the following example we will go through the Kaczmarz row-action method to see how it works. For small systems like this example this is not the best method to solve systems because the convergence is not fast and the disadvantages of the Gaussian elimination are not high in systems of this size.

Example 6.8. (continued, based on Example 3.10)
In this example we will compute again a solution for the linear programming problem we used also to

illustrate the Simplex Method. For the Simplex Method we have introduced the slack variables but for the Kaczmarz algorithm we do not need them, so we solve the following system:

$$\mathbf{Ax} = \mathbf{b} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 20 \\ 20 \\ 20 \end{bmatrix}$$

We will begin with an initial guess for $\mathbf{x}$ as $\mathbf{x}^{(0)} = (1, 1, 1)$. For the computations we use no relaxation (thus $\lambda = 1$) and a cyclic control sequence (thus $i = k \textbf{ mod } 3 + 1$). To compute the next $\mathbf{x}$ image vector we apply equation 5:

$$\begin{aligned} \mathbf{x}^{(1)} &= \mathbf{x}^{(0)} + \frac{b_1 - \mathbf{a}_1 \cdot \mathbf{x}^{(0)}}{\|\mathbf{a}_1\|^2} \mathbf{a}_1 \\ &= (1,1,1) + \frac{20 - (1,2,2) \cdot (1,1,1)}{9} (1,2,2) \\ &= (1,1,1) + (\frac{15}{9}, 2\frac{15}{9}, 2\frac{15}{9}) \\ &= (2.6667, 4.3333, 4.3333) \end{aligned}$$

If we now compute $\mathbf{Ax}^{(1)}$ we get the following result:

$$\mathbf{Ax}^{(1)} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix} \begin{bmatrix} 2.6667 \\ 4.3333 \\ 4.3333 \end{bmatrix} = \begin{bmatrix} 20.000 \\ 18.333 \\ 18.333 \end{bmatrix}$$

we see that after the first iteration the first row satisfies the constraint defined by the initial system. If we proceed with the following iteration we get:

$$\begin{aligned} \mathbf{x}^{(2)} &= \mathbf{x}^{(1)} + \frac{b_2 - \mathbf{a}_2 \cdot \mathbf{x}^{(1)}}{\|\mathbf{a}_2\|^2} \mathbf{a}_2 \\ &= (2.6667, 4.3333, 4.3333) + \frac{20 - (2,1,2) \cdot (2.6667, 4.3333, 4.3333)}{9} (2,1,2) \\ &= (3.0370, 4.5185, 4.7037) \end{aligned}$$

and

$$\mathbf{Ax}^{(2)} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix} \begin{bmatrix} 3.0370 \\ 4.5185 \\ 4.7037 \end{bmatrix} = \begin{bmatrix} 21.481 \\ 20.000 \\ 19.815 \end{bmatrix}$$

If we proceed we get the following results for $\mathbf{x}^{(k)}$:

$$\begin{aligned} \mathbf{x}^{(1)} &= (2.666667, 4.333333, 4.333333) \\ \mathbf{x}^{(2)} &= (3.037037, 4.518519, 4.703704) \\ \mathbf{x}^{(3)} &= (3.078189, 4.559671, 4.724280) \\ \mathbf{x}^{(4)} &= (2.895290, 4.193873, 4.358482) \\ \mathbf{x}^{(5)} &= (3.183864, 4.338160, 4.647056) \\ \mathbf{x}^{(10)} &= (3.264204, 4.012754, 4.355144) \\ \mathbf{x}^{(100)} &= (4.003872, 3.999318, 3.998746) \\ \mathbf{x}^{(200)} &= (3.999992, 4.000002, 4.000007) \\ \mathbf{x}^{(300)} &= (4.000000, 4.000000, 4.000000) \end{aligned}$$

We see that after 300 iterations we get a result that is accurate up to 6 decimals according to what we found with the simplex method.

Remark: From the procedure above something is implicit: we have changed the less or equal sign in an equality sign because we *know* that we can use this constraint and that this satisfies the minimization problem. But if we introduce the slack variables x_4, x_5 and x_6 we used also for the Simplex Method we get another solution:

$$\mathbf{x}^{(100)} = (3.692287, 3.692314, 3.692305, 1.538475, 1.538448, 1.538458)$$

that also satisfies the constraint but not minimizes the cost function $f(\mathbf{x}) = -10x_1 - 12x_2 - 12x_3$. This cost-function is not used in the Kaczmarz algorithm. According to Corollary 6.6 the Kaczmarz algorithm uses another function to minimize, namely the cost function $f(\mathbf{x}) = \|\mathbf{x}\|$.

If we compute the difference in norm between $\mathbf{x}^{(100)}$ and the expected solution $\mathbf{x}^*$:

$$\begin{aligned}\mathbf{x}^{(100)} &= (3.692287, 3.692314, 3.692305, 1.538475, 1.538448, 1.538458)\\ \mathbf{x}^* &= (4, 4, 4, 0, 0, 0)\end{aligned}$$

we see indeed a small difference:

$$\left\|\mathbf{x}^{(100)}\right\| - \|\mathbf{x}^*\| \approx -10^{-5}.$$

6.2 Block-iterative method

In the Kaczmarz method as described in the previous section all rows are processed in a given sequence and one by one. If we take a closer look at the system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A} \in \mathbb{R}^{m \times n}$ we see that we can divide the system in blocks. Therefore we make a partition in the following form:

$$\mathbf{Ax} = \mathbf{b} = \left[\begin{array}{c}\mathbf{A}_{[\mathbf{1}]}\\ \hline \mathbf{A}_{[\mathbf{2}]}\\ \hline \vdots \\ \hline \mathbf{A}_{[\mathbf{M}]}\end{array}\right] \mathbf{x} = \left[\begin{array}{c}\mathbf{b}_{[\mathbf{1}]}\\ \hline \mathbf{b}_{[\mathbf{2}]}\\ \hline \vdots \\ \hline \mathbf{b}_{[\mathbf{M}]}\end{array}\right] \tag{6}$$

where $\mathbf{A}_{[\mathbf{i}]}$ is a submatrix of $\mathbf{A}$ and $\mathbf{b}_{[\mathbf{i}]}$ is a subvector of $\mathbf{b}$. To get this form we define M blocks based on the row indices of the constraints of $\mathbf{Ax} = \mathbf{b}$. Thus we choose the boundary indices that split the blocks $\{s_t\}_{t=0}^{M}$ such that:

$$0 = s_0 < s_1 < s_2 < \ldots < s_{M-1} < s_M = m \tag{7}$$

For all t with $1 \leq t \leq M$ we define the set of all indices of constraints contained in block t:

$$I_t = \{s_{t-1} + 1, s_{t-1} + 2, \ldots, s_t - 1, s_t\}$$

The set of I_t's give a partition:

$$\bigcup_{t=1}^{M} I_t = \{1, 2, \ldots m\} \tag{8}$$

With this partition we have defined blocks, and we can apply a block-iterative algorithm such as the block-iterative generalization of the Kaczmarz algorithm. In this block-iterative algorithm we compute in iteration k not only a difference between $\mathbf{x}^{(k+1)}$ and $\mathbf{x}^{(k)}$ for one row, but we compute that for more than one row and add that to the previous solution $\mathbf{x}^{(k)}$. In general form this becomes:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{A}^T_{[\mathbf{t(k)}]}\Phi_k \left(\mathbf{b}_{[\mathbf{t(k)}]} - \mathbf{A}_{[\mathbf{t(k)}]}\mathbf{x}^{(k)}\right)$$

where we can choose a control sequence $t(k)$ in which order we want to loop through the blocks, for example cyclic: $t(k) = k \ \mathbf{mod}\ M + 1$. In this equation $\Phi_k \in \mathbb{R}^{L \times L}$ is the relaxation matrix where L is the size of block $t(k)$, i.e. $L = |I_{t(k)}|$. The convergence of this method is stated in e.g. [10].

A special case of the block-iterative Kaczmarz algorithm is when the relaxation matrices Φ_k are diagonal matrices defined for $k \geq 0$ by:

$$\Phi_k = \lambda_k \mathrm{diag}(\phi_l^{(k)})$$

where for $l = 1, 2, \ldots, |I_{t(k)}|$, $\phi_l^{(k)}$ is defined by:

$$\phi_l^{(k)} = \frac{1}{\left\|\mathbf{A}_{(s_{t(k)-1}+l,\bullet)}\right\|^2}$$

Then we can rewrite the block-iterative Kaczmarz algorithm in a similar form to the original Kaczmarz algorithm:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \lambda_k \sum_{i \in I_{t(k)}} \frac{b_i - \mathbf{A}_{(i,\bullet)} \cdot \mathbf{x}^{(k)}}{\left\|\mathbf{A}_{(i,\bullet)}\right\|^2} \mathbf{A}_{(i,\bullet)} \tag{9}$$

From the algorithm step in Equation 9 we can go back to the Kaczmarz algorithm by choosing m blocks of size 1. Thus $s_0 = 0, s_1 = 1, \ldots, s_m = s_M = m$.

6.2.1 Cimmino's algorithm - SIRT

Another extreme of this block-iterative algorithm is when we choose only one block of size m. Thus $s_0 = 0, s_1 = S_M = m$. Thus to come in step $k+1$ from $\mathbf{x}^{(k)}$ to $\mathbf{x}^{(k+1)}$ we need to compute the differences for all rows in $\mathbf{Ax} = \mathbf{b}$ in stead of only one row. This method is known as Cimmino's algorithm or SIRT (Simultaneous Iterative Reconstruction Technique) [14].

One of the disadvantages of the Kaczmarz algorithm is that the reconstructed image shows 'stripping' [18]. This can be solved with Cimmino's algorithm by updating the pixels in the $\mathbf{x}$ image vector only after processing all rows. Then the $\mathbf{x}$ vector is updated with the difference vectors of all projection rows at once. This results only also in a big downside: Cimmino's algorithm has a slow convergence and therefore it takes a long time to reconstruct the image.

Remark: in our explanation we said that in each iteration the difference for that row is added to a difference vector. The addition of the difference vector to the image $\mathbf{x}$ is done only when the algorithm processed over all rows of the equation $\mathbf{Ax} = \mathbf{b}$. This makes the relation between Cimmino's algorithm and the block-iterative Kaczmarz algorithm clear, and is also explained by [6]. Some other authors do not use the addition of the *full* difference vector, but *average* the difference over all rows of the equation, such as [14].

This means that we have another notion of relaxation. The version where the averaged difference is added to image $\mathbf{x}$ can be seen as a version with relaxation of the algorithm with full addition of the difference, namely $\lambda = \frac{1}{\texttt{length}(\mathbf{b})}$.

Example 6.9. In this example we will give an intuitive idea about the block-iterative Kaczmarz method and the geometry behind it. We use therefore a 2D problem, thus the only block we can make is the full matrix $\mathbf{A}$ (thus this is also an example of Cimmino's algorithm for 2D). In this method we project again the point $\mathbf{x}^{(k)}$ on the two constraint lines as we also did for the standard Kaczmarz algorithm. But now we add both difference vectors $\mathbf{d_1}$ and $\mathbf{d_2}$ to the solution $\mathbf{x}^{(k)}$ to get the new solution $\mathbf{x}^{(k+1)}$. This is also shown in Figure 14a.

If we execute this method iteratively we get a picture like Figure 14b. In this figure we plot the iterations of the Block-iterative Kaczmarz algorithm for the following system with an initial guess $\mathbf{x}^{(0)} = (-1, \frac{3}{2})$:

$$\mathbf{Ax} = \mathbf{b} = \begin{bmatrix} 1 & 1 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

If we compute the first iteration of Cimmino's algorithm by hand we get the following results. As said earlier we begin with the initial guess $\mathbf{x}^{(0)} = (-1, \frac{3}{2})$. We use also a cyclic control sequence, thus $i = k \textbf{ mod } 2 + 1)$. Before we compute the next $\mathbf{x}$ image vector we unpack the summation in equation 9 and set the relaxation parameters $\lambda_k = 1$:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \frac{b_1 - \mathbf{A}_{(1,\bullet)} \cdot \mathbf{x}^{(k)}}{\left\|\mathbf{A}_{(1,\bullet)}\right\|^2} \mathbf{A}_{(1,\bullet)} + \frac{b_2 - \mathbf{A}_{(2,\bullet)} \cdot \mathbf{x}^{(k)}}{\left\|\mathbf{A}_{(2,\bullet)}\right\|^2} \mathbf{A}_{(2,\bullet)}$$

Now we can compute the first iteration:

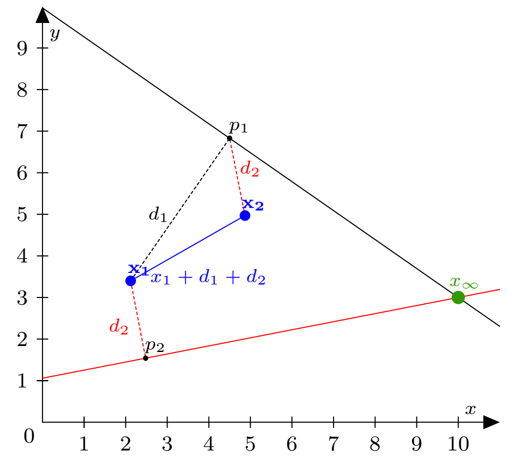
$$\begin{aligned} \mathbf{x}^{(1)} &= \left(-1, \frac{3}{2}\right) + \frac{1 - (1,1) \cdot \left(-1, \frac{3}{2}\right)}{2} (1,1) + \frac{1 - (2,5) \cdot \left(-1, \frac{3}{2}\right)}{29} (2,5) \\ &= \left(-1, \frac{3}{2}\right) + \frac{1}{4} (1,1) - \frac{9}{58} (2,5) \\ &= (-1.06034, 0.97414) \end{aligned}$$

The next iterations can be computed in the same way. If we look at the found $\mathbf{x}^{(1)}$ we see that this is indeed the fist computed point in the plot of Cimmino's algorithm as shown in figure 14b.

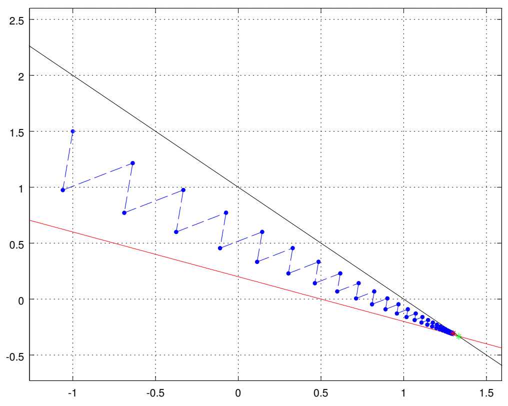
Important is the difference with the standard Kaczmarz algorithm: now we compute the part $\frac{b_i - \mathbf{A}_{(i,\bullet)} \cdot \mathbf{x}^{(k)}}{\left\|\mathbf{A}_{(i,\bullet)}\right\|^2} \mathbf{A}_{(i,\bullet)}$ for all projections before the results are added to the solution vector $\mathbf{x}$.

6.2.2 Block-Kaczmarz algorithm - SART

Another Row-action method for reconstruction of images from projections is the Block-Kaczmarz algorithm or Simultaneous Algebraic Reconstruction Technique (SART). In problems of two dimensions



(a) Sketch of iteration of Block Kaczmarz or Cimmino's algorithm on 2D



(b) Plot of iterations of the Block Kaczmarz for problem of example 6.9

Figure 14: Sketch and plot illustrations for Example 6.9

we do not have any choice for defining blocks in the Block Kaczmarz method. When the dimensions increase, and that is the case in image reconstruction, we have many choices and are not forced to make the choice to iterate over all projections before we update the $\mathbf{x}$ image vector.

The SART method is an iterative method what must yield reconstructions of a good quality and accuracy in only one iteration [1, 14]. Therefore Anderson and Kak describe some main features of SART. Some of them are related to some modifications in the model behind image reconstruction. To reduce the noise of the general Kaczmarz algorithm they also suggest to apply the correction term not after each projection (as in the Kaczmarz algorithm) or after each full iteration (as in Cimmino's algorithm) but after the computation of all projections of a specific angle.

Another way to improve the quality of the reconstruction is to give only partial weights on the picture elements on the boundaries of a ray.

6.3 Classification of block-iterative methods

In the previous sections we discussed multiple row-action methods which can be used to find solutions for huge systems. In this section we will introduce a classification for row action methods. This classification is based on the classification of algorithms in [6].

We divide the row action methods in four classes based on two axes, namely whether the algorithm is sequential or parallel and whether the algorithm is performed on a row or a block of rows:

- *Sequential Algorithm.* The algorithms in this class are controlled with a control sequence $\{i(k)\}_{k=0}^{\infty}$. The algorithm performs the row operations sequentially until a stopping criterion is fulfilled.

- *Parallel Algorithm.* The row operations are performed in parallel on all rows. The next iterative solution $\mathbf{x}^{(k+1)}$ is generated based on the intermediate results of the other processors.

- *Sequential Block-Iterative Algorithm.* The system $\mathbf{Ax} = \mathbf{b}$ is divided into blocks as shown in the section about block-iterative methods. Then the blocks are processed sequentially following a control sequence.

- *Parallel Block-Iterative Algorithm.* The block iterations are performed in parallel on all blocks. The next iterative solution $\mathbf{x}^{(k+1)}$ is generated based on the intermediate results of the other processors.

Example 6.10. The Kaczmarz algorithm (Equation 5) is an example of a Sequential Algorithm. All rows are processed sequentially and one by one according to a control sequence.

Example 6.11. With a similar reasoning the Block Kaczmarz algorithm (Equation 9) is a Block-Iterative Algorithm. In this case blocks are defined and the algorithm performs the row operations in parallel on all rows in a block.

In this case we see a summation over all $i \in I_{t(k)}$. This summation can be parallelized such that multiple processors perform the vector multiplication such that a master processor only needs to sum the results of the multiplications done by the other processors. Therefore the Block-iterative Kaczmarz algorithm is a Parallel Block-Iterative Algorithm.

7 Implementation

After the analysis of some row-action methods we will implement some of the algorithms and discuss the behavior and some of the parameters with which we can adjust the behavior of the algorithm. The implementation of the algorithms for this thesis is done in the object oriented programming language C#. The benefits of C# as programming language for this implementation are for example operator overloading and a clear usage of delegates with lambda expressions.

The C# program as result of this implementation is compilable with Mono[1] and thus platform independent.

7.1 Program pipeline

Before we go in details we describe some main functions of our program to get an idea about what is going on and how that is related to image reconstruction and the application of diagnostic medicine. The main part is the reconstruction of an image based on projection data. Before we can reconstruct images we need two preparation steps to create data that we can use for reconstruction.

7.1.1 Generate Shepp-Logan Phantom

The first phase of or program is the phase where a standardized image is generated. Based on the definition of the Shepp-Logan 'head phantom', see Table 1, our program can generate a picture of this 'head phantom' in all resolutions we want.

7.1.2 Generate projections

The second phase is to generate projections based on an 'original' input image. We want to treat also other input images then the Shepp-Logan Phantom and so we compute the projections by constructing rays over the input image and sum the gray-value of the pixels that are on the given line.

The computation of the ray sums on the horizontal or vertical lines of the image are clear. But we need to say something more about how we selected the pixels that are on a line with a given angle θ. Given that we know two end points $\mathbf{p_1} = (x_1, y_1)$ and $\mathbf{p_2} = (x_2, y_2)$ where $\mathbf{p_1}$ and $\mathbf{p_2}$ are on two different borders of the input image we can select pixels between $\mathbf{p_1}$ and $\mathbf{p_2}$ using Bresenham's algorithm [16].

Bresenham's algorithm, see Pseudocode 1, is a known algorithm from Computer Graphics to draw discrete lines on a pixel grid given a start point and an endpoint. Given a start point $\mathbf{p_1}$, an end point $\mathbf{p_2}$ and a linear function $f : [x_1, x_2] \to \mathbb{R}$, then the algorithm selects connected points with the shortest distance to the line given by the function. If the line that must be drawn is increasing and in the first quadrant then the algorithm starts at the start point $\mathbf{p_1}$ and decides in each step if it must pick the east or the north-east pixel next to the current pixel. For a detailed discussion of Bresenham's algorithm we refer to [16].

```
int y = y1;
double d = f(x0 + 1, y0 + 0.5)
for (int x = x0; x =< x1; x++) {
  do_something_with_pixel(x,y);
  if (d < 0)
  {
    y = y + 1;
    d = d + x1-x0 + y0-y1
  }
  else
    d = d + y0-y1
}
```

Code 1: Pseudocode of Bresenham's algorithm.

In this case we do not use Bresenham's algorithm to draw lines at a picture but we use the handle `do_something_with_pixel(x, y)` to select that pixel for the ray and add its gray-value to the ray-sum.

A start point $\mathbf{p_1}$ of a ray is chosen as boundary point of the input-image. Depending on the angle of the ray we use the pixels at the bottom and left sides or the bottom and right sides as start points for the rays with the given angle. The endpoint of the ray is selected by adding $\delta = (\cos\theta, \sin\theta)$ to

[1]Mono is a cross platform open source .NET development framework, see `http://www.mono-project.com`

$\mathbf{p_1}$ until we hit another boundary of the image. The pixel where we leave the image when we add something in the direction of δ is called $\mathbf{p_2}$.

7.1.3 Reconstruction phase

The third and main phase of the program is the part where the reconstruction of the input image is done based on the projection data from the second phase.

One of the properties of a row-action method is that it uses in one iteration only one row of the system $\mathbf{Ax} = \mathbf{b}$. The data of that row is used in some linear algebra computations. Therefore we implemented some objects in a LinearAlgebra project. An example of a class in that project is the class Vector, for the important parts see Section A.1. With the operator overloading for this class we can implement the usual operations for addition and multiplication of constants and vectors and also vectors and vectors. This makes the source-code more readable and the implementation of the algorithms more in line with the equations for the iteration of the algorithms in Equation 5 and 9.

7.2 Kaczmarz - ART

Equation 5 describes the iteration that is done in the Kaczmarz algorithm. In the implementation we have bounded the number of iterations with `MaxRowOperations` which describes the total number of rows that can be processed in the algorithm.

In every iteration the Kaczmarz algorithm chooses an index `idx`. The index `idx` represents the row of the system $\mathbf{Ax} = \mathbf{b}$ that will used in this iteration of the algorithm. The choice of `idx` based on the iteration number can be controlled with a given function `controlSequence(i, m)`.

We assume a (almost) cyclic control sequence and choose `idx` == 0 as the indicator that the algorithm has processed a full iteration over all rows of the system.

From the implementation of the Kaczmarz algorithm in Code 2 we see that in all iterations the image vector `x` is updated in such a way that row `idx` satisfy the constraint:

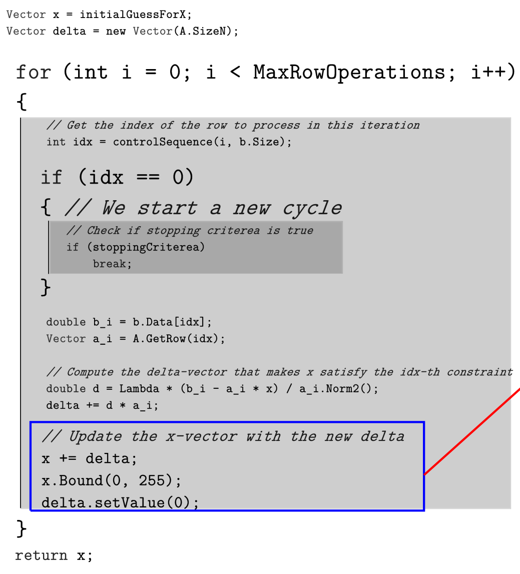
$$\mathbf{a}_i x_i = b_i$$

On the image vector `x` we apply also a `Bound(min, max)`. The `Bound` sets all elements in `x` below a given minimum to the minimum value and all elements above a given maximum value to the maximum value. This is a little modification of the Kaczmarz algorithm but with this operation we can apply an additional constraint we know already before: all gray intensities are between 0 and 255.

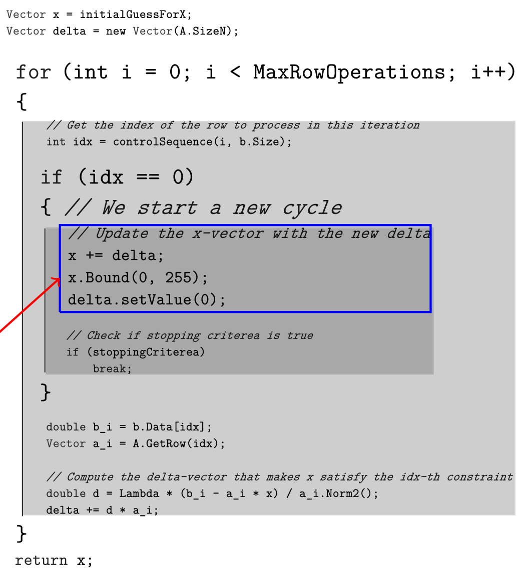
This bound is applied after each update of the `x` vector so that we do not use illegal color values in the next iterations of the algorithm. Under the given constraints we saw in the experiments that the Kaczmarz algorithm changed only a small number of pixel-values with the `Bound` method.

```
Vector x = initialGuessForX;
Vector delta = new Vector(A.SizeN);
for (int i = 0; i < MaxRowOperations; i++)
{
    // Get the index of the row to process in this iteration
    int idx = controlSequence(i, b.Size);

    if (idx == 0)
    { // We start a new cycle
        // Check if stopping criterea is true
        if (stoppingCriterea)
            break;
    }

    double b_i = b.Data[idx];
    Vector a_i = A.GetRow(idx);

    // Compute the delta-vector that makes x satisfy the idx-th constraint
    double d = Lambda * (b_i - a_i * x) / a_i.Norm2();
    delta += d * a_i;

    // Update the x-vector with the new delta
    x += delta;
    x.Bound(0, 255);
    delta.SetValue(0);
}
return x;
```

Code 2: Code that applies the Kaczmarz algorithm following Equation 5.



```
Vector x = initialGuessForX;
Vector delta = new Vector(A.SizeN);

for (int i = 0; i < MaxRowOperations; i++)
{
    // Get the index of the row to process in this iteration
    int idx = controlSequence(i, b.Size);

    if (idx == 0)
    { // We start a new cycle
        // Check if stopping criterea is true
        if (stoppingCriterea)
            break;
    }

    double b_i = b.Data[idx];
    Vector a_i = A.GetRow(idx);

    // Compute the delta-vector that makes x satisfy the idx-th constraint
    double d = Lambda * (b_i - a_i * x) / a_i.Norm2();
    delta += d * a_i;

    // Update the x-vector with the new delta
    x += delta;
    x.Bound(0, 255);
    delta.setValue(0);
}
return x;
```



```
Vector x = initialGuessForX;
Vector delta = new Vector(A.SizeN);

for (int i = 0; i < MaxRowOperations; i++)
{
    // Get the index of the row to process in this iteration
    int idx = controlSequence(i, b.Size);

    if (idx == 0)
    { // We start a new cycle
        // Update the x-vector with the new delta
        x += delta;
        x.Bound(0, 255);
        delta.setValue(0);

        // Check if stopping criterea is true
        if (stoppingCriterea)
            break;
    }

    double b_i = b.Data[idx];
    Vector a_i = A.GetRow(idx);

    // Compute the delta-vector that makes x satisfy the idx-th constraint
    double d = Lambda * (b_i - a_i * x) / a_i.Norm2();
    delta += d * a_i;
}
return x;
```

(a) Overview of implementation Kaczmarz algorithm (b) Overview of implementation Cimmino's algorithm

Figure 15: Comparison of implementation of Kaczmarz and Cimmino's algorithm. The block that updates **x** is moved to execute only if the algorithm starts a new iteration.

7.3 Cimmino - SIRT

Cimmino's algorithm is based on the same idea as the Kaczmarz algorithm. We can see that also back in the implementation of Cimmino's algorithm. The addition of the delta vector which is done in the Kaczmarz algorithm after each iteration is moved in the implementation of Cimmino's algorithm to the part that is done when a new cycle starts, as illustrated Figure 15 and seen in Code 3.

Also in Cimmino's algorithm implementation we can control the choice of `idx` using a control sequence (although a change in a cyclic control sequence of Cimmino's algorithm does not result in another computation in the iteration. The changes are at most a change in the numerical error of the solution of the iteration) and bound the resulting **x** to the gray intensity range.

```
Vector x = initialGuessForX;
Vector delta = new Vector(A.SizeN);

for (int i = 0; i < MaxRowOperations; i++)
{
    // Get the index of the row to process in this iteration
    int idx = controlSequence(i, b.Size);

    if (idx == 0)
    { // We start a new cycle
        // Update the x-vector with the new delta
        x += delta;
        x.Bound(0, 255);
        delta.SetValue(0);

        // Check if stopping criterea is true
        if (stoppingCriterea)
            break;
    }

    double b_i = b.Data[idx];
    Vector a_i = A.GetRow(idx);

    // Compute the delta-vector that makes x satisfy the idx-th constraint
    double d = Lambda * (b_i - a_i * x) / a_i.Norm2();
    delta += d * a_i;
}
return x;
```

Code 3: Code that applies Cimmino's algorithm following Equation 9.

7.4 Block-Kaczmarz - SART

When we implement the Block-Kaczmarz algorithm we need to introduce some additional bookkeeping to know which block we process. Therefore we use an array with indices that defines the blocks as described in Equation 7. We can use different types of partitions as suggested by the discussion of Equation 7. A natural partition can be the partition that groups the rows of the system according to the rays of one direction into one block. In the phase of the generation of the projections the indices of rays that belongs to the same angle are already stored in such an array. So the default partition is the partition into blocks with rays that corresponds to the same direction.

This partition related to directions has also the advantage (in the continuous case) that all pixels are hit at most once per block. Due to the discretization we used and the choice of Bresenham's algorithm to construct projection lines this advantage is not always true.

In the Block-Kaczmarz implementation we compute for all rows in a block also the difference vector as in the Kaczmarz implementation and add that to a delta-vector. When all rays in a block are processed we add the delta-vector to the image vector **x** and process the rows of the next block.

In the Block-Kaczmarz implementation we iterate using a standard cyclic control over the rows in a block. Also here a change in this sequence results in another order of addition of delta-vectors and not another summation. Therefore we neglect this possibility for control and use the cyclic control $t(k) = k \textbf{ mod } m$.

We can control the choice of the next block, the `currentBlockId`, using a control sequence. Also in this implementation we bound the resulting **x** to the gray intensity range.

```
Vector x = initialGuessForX;
Vector delta = new Vector(A.SizeN);

int currentBlockId = 0;
int blocksProcessed = 0;
int blockLine = 0;

for (int i = 0; i < MaxRowOperations; i++)
{
    // Get the index of the row to process in this iteration
    int idx = blocks[currentBlockId] + blockLine;

    if (blocks == null || idx == blocks[currentBlockId+1])
    { // We came to a row that belongs to another block
        // Update the x-vector with the new delta
        x += delta;
        x.Bound(0, 255);
        delta.SetValue(0);

        // Enter the next block
        blockLine = 0;
        blocksProcessed++;
        currentBlockId = controlSequence(blocksProcessed, blocks.Length - 1);

        // Recompute the idx based on the next block
        idx = blocks[currentBlockId] + blockLine;

        if (idx == 0)
        { // We start a new cycle
            // Check if stopping criterea is true
            if (stoppingCriterea)
                break;
        }
    }

    double b_i = b.Data[idx];
    Vector a_i = A.GetRow(idx);

    // Compute the delta-vector that makes x satisfy the idx-th constraint
    double d = Lambda * (b_i - a_i * x) / a_i.Norm2();
    delta += d * a_i;
    blockLine++;
}
return x;
```

Code 4: Code that applies the Block-Kaczmarz algorithm following Equation 9.

When we take a closer look at the Block-Kaczmarz implementation and the Kaczmarz implementation we see indeed that the Block-Kaczmarz is a generalization of the Kaczmarz algorithm. To see that we construct blocks of size one, thus:

$$\texttt{blocks} = [0, 1, 2, \ldots, m-1, m];$$

For all iteration $i \geq 1$ of the `for` loop the following condition is true:

```
idx == blocks[currentBlockId+1]
```

This because $0 \leq$ `idx` $\leq m$ is the potential row of the system $\mathbf{Ax} = \mathbf{b}$ that is used in iteration i. All integers between 0 and m are a boundery of a block, and thus `idx == blocks[currentBlockId+1]`. Thereby the body of the `if` statement on line 13 is always executed. The only difference between the two implementations is the iteration-number. The computation of iteration n in the Kaczmarz implementation is done in iteration $n+1$ of the Block-Kaczmarz implementation because the update of image $\mathbf{x}$ for the previous block is done before the update of the delta-vector for the current ray.

A similar reasoning can be used to show that Cimmino's algorithm is another extreme version of the Block-Kaczmarz. Therefore we choose only one block, thus:

$$\texttt{blocks} = [0, m];$$

The condition on line 13 is not the same but equivalent to `idx == 0`. The condition is true when `idx` $= m$ and before the `idx` is used it is in the recomputation of `idx` set to 0. That makes this version the same as Cimmino's algorithm.

8 Practical analysis

In this section we will do some experiments with the implementations of the algorithms for solving image reconstruction problems. In the experiments we will discuss some variables or parameters and what the influence of that parameter is on the computation or the result. Therefore we discuss in Section 8.1 the influence of the relaxation parameter. In Section 8.2 we will consider the relation between the block size and the convergence of the Block-Kaczmarz algorithm. In Section 8.3 we look at the relation between the number of directions and the number of rays from the input of the image reconstruction. The influence of a different control sequence is shown in Section 8.4. We summarize our practical results in Section 8.5 and we discuss also some possibilities to execute the Block-Kaczmarz algorithm in parallel to reduce the computation time of the Block-Kaczmarz implementation.

8.1 Relaxation parameter

Until now we did not spent a lot of attention to the relaxation parameter in the row action methods. The analysis of the relaxation parameter in equations like Equation 5 is often done in practice [12]. So before we can compare the three implementations of the row action method we need also to look at the relaxation parameter to optimize the row action method.

8.1.1 Introduction

The relaxation parameter is a weight-parameter for the delta-vector we add to the solution $\mathbf{x}^{(k)}$ to come to the next iteration $\mathbf{x}^{(k+1)}$. If we decrease the relaxation parameter the iterations are more conservative to the computations already done. So the speed of convergence slows down, but the quality of the image increases [4]. On the other hand when we use over-relaxation the speed of converges increases, but the quality of the result image reduces.

An optimal relaxation parameter minimizes the execution time of the algorithms and also results in a specified quality of the image. Therefore we ask ourselves:

> What is the best relaxation parameter for the Kaczmarz (ART), Cimmino (SIRT) and Block-Kaczmarz (SART) algorithm for a good execution time and also a good image?

8.1.2 Method

In this experiment we use the Shepp-Logan head phantom as sample input to look at the relaxation parameter. We use a 128×128 pixels image with the gray-values between 0.9 and 1.1 stretched to the whole gray-range, as in Figure 12. In this experiment we use projections from 64 directions with 128 rays each. This results in $64 * 128 = 8.192$ constraints for $128^2 = 16.384$ unknown pixel values. In this way the system is under-determined as in most image reconstruction problems. But due to the relative high number of directions and rays this setup will result in reasonably good quality images.

We introduce the following measure to compare the results of the iterative reconstruction algorithms with the original image of the head phantom, $\mathbf{x}$:

$$\delta(\mathbf{x}^{(k)}) := \left\| \mathbf{x}^{(k)} - \mathbf{x} \right\| \tag{10}$$

Note that this distance cannot be used in practice. This distance function requires that we know the original image $\mathbf{x}$. For this experiment it is a good distance measure because we want a reconstruction of the original image with a given quality. This distance is also independent of the choice of relaxation parameter λ but the rate of convergence is not a criteria in this distance and stopping criteria.

If we want to use the algorithms in practice we need other measures. The choice of a good stopping criteria for practice is out of the scope of this experiment. For alternative distances we could use for example the rate of convergence. We could also say that a good image has a high contrast and a low level of noise. Mentioned that we have increased the contrast of the Shepp-Logan phantom this could also be a nice measure, but due to the simplicity of the δ-measure we choose the norm of the difference as measure for $\mathbf{x}^{(k)}$.

Now we need to define what we mean with a 'good quality' image. In terms of the $\delta(\mathbf{x})$-function of Equation 10 we say that all images $\mathbf{x}^{(k)}$ with $\delta(\mathbf{x}^{(k)}) \leq 1000$ have a 'good quality'. This criterion can serve as a stopping criterion for the algorithms.

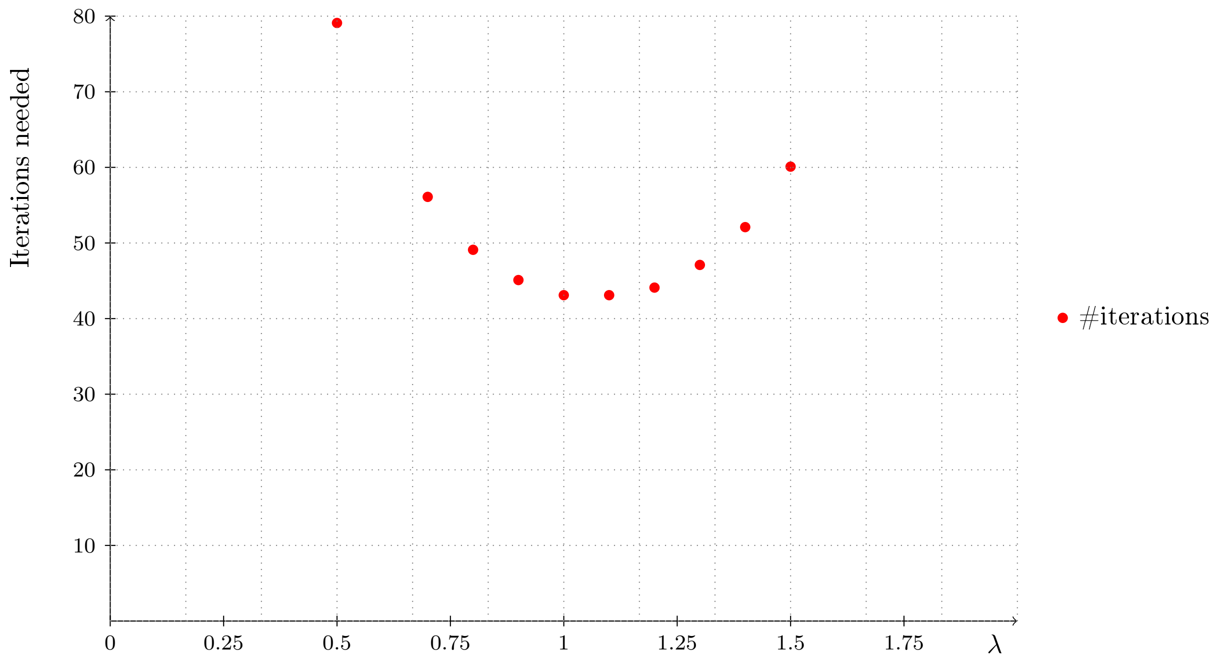


Figure 16: Plot of total number of iterations needed for the implementation of the Kaczmarz algorithm to achieve the 'quality' $\delta \leq 1000$.

8.1.3 Hypothesis

If we plot computation time against the relaxation parameter we expect to find a parabola-like figure from which we can read-off the best relaxation parameter. From that relaxation parameter there is a branch to lower relaxation parameters in which the computation time increases. This is because the relaxation is too conservative to converge fast to the original good image. On the other branch, where the relaxation parameter increases, the computation time also increases. This is because the algorithm jumps faster to other images towards the original image but it needs multiple iterations to establish a good quality image.

8.1.4 Results

When we apply the Kaczmarz implementation we choose $\lambda \in \{0.5, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.5\}$. When we run the algorithm the computation time does not vary much when the total number of full iterations stays the same. The computation time grows linearly with the number of iterations needed. On the machine where we tested the implementations the execution of one full iteration of the Kaczmarz algorithm tooks approximately $2.2 \cdot 10^3$ ms.

When we plot the total iterations needed for different λ to achieve the 'good quality' image we see a plot as in Figure 16. The red points are the absolute number of iterations needed to terminate the algorithm, i.e. achieve an image of good quality.

When we apply the Block-Kaczmarz implementation we choose $\lambda \in \{0.5, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.5\}$. When we run the algorithm also for the Block-Kaczmarz implementation the computation time does not vary much when the total number of full iterations stay the same. The computation time grows linearly with the number of iterations needed. On the machine where we tested the implementations the execution of one iteration tooks approximately $1.9 \cdot 10^3$ ms. The plots of the total number of iterations needed for the Block-Kaczmarz implementation is shown in Figure 17.

The implementation of Cimmino's algorithm has some varying results. For some values of the relaxation parameter λ the algorithm is unstable or the algorithm jumps between two different images where both series of images did not converge to the original image $\mathbf{x}$. For example, λ values in the order of the λ for the Kaczmarz implementation result in a bad image and no convergence to original image $\mathbf{x}$. Some values in the range $0.01 \leq \lambda \leq 0.06$ result in a sequence of images that sometimes converges to $\mathbf{x}$. Also in this interval for some λ values the algorithm did not converge, such as $\lambda = 0.048$. If the algorithm convergences then the convergence is very slow. Therefore we have increased our notion of a 'good' quality to $\delta \leq 3000$. In Figure 18 we see the total number of iterations needed to achieve the

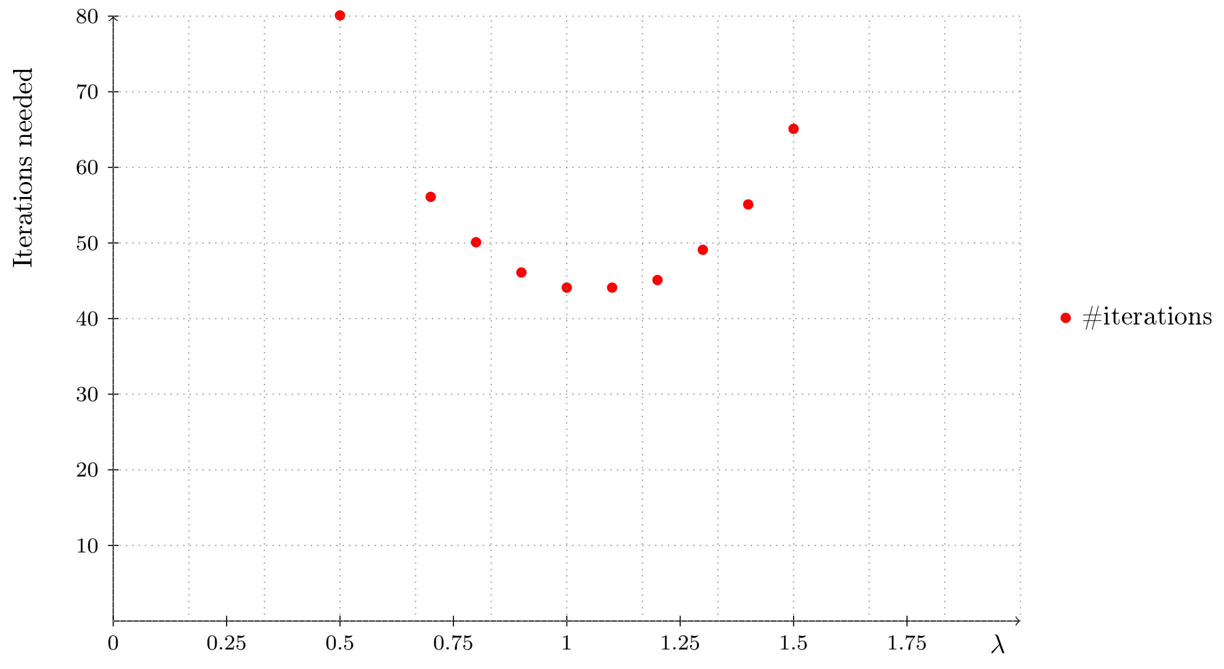


Figure 17: Plot of total number of iterations needed for the implementation of the Block-Kaczmarz algorithm to achieve the 'quality' $\delta \leq 1000$.

'good' quality and terminate the implementation of Cimmino's algorithm.

Remark: We have used 64 direction from which we have taken rays through the object. Therefore all pixels will have approximately 64 value updates when all lines in the system are processed once. Thus if we choose a relaxation parameter $\lambda = \frac{1}{64} \approx 0.016$ then we find indeed some λ-values where the algorithm converges.

8.1.5 Conclusion

If we consider the iterations needed for the Kaczmarz algorithm and also the Block-Kaczmarz algorithm we see that there is a bandwidth around some λ for which the algorithms converge. For both algorithms we can choose λ such that $0.8 \leq \lambda \leq 1.2$. For that range the number of iterations is minimal. This makes the Kaczmarz algorithm and its variations a set of good algorithms from where we can go further to improve the algorithm. The value $\lambda = 1$ or 1.1 is for the Kaczmarz and the Block-Kaczmarz algorithm approximately the center of the bandwidth and results in an image with the 'best quality' with 43 (for the Kaczmarz) or 44 (for the Block-Kaczmarz) needed iterations.

For Cimmino's algorithm a good choice of λ is hard. For multiple λ the algorithm does not converge and if a λ is found for which the algorithm converges, then there can be another relaxation parameter in the neighborhood of that λ that does not converge. Also the computation time of Cimmino's algorithm is very long. Compared with the computation time of the Kaczmarz and Block-Kaczmarz algorithm – which computes a better quality image – Cimmino's algorithm results in lower quality images. Therefore we exclude Cimmino's algorithm in the further experiments.

8.1.6 Discussion

Under the given constraints described in the sections introduction (Section 8.1.1) and method (Section 8.1.2) we can find a relaxation parameter λ. The result is that $\lambda = 1$ is the best relaxation parameter in this scenario for Kaczmarz and Block-Kaczmarz. Thus we do not need any relaxation. From this experiment we see that under the given constraints there exist a bandwidth in which the relaxation parameter can be chosen. But from the results of this experiment we do not know how the relaxation parameter performs in other situations, such as a stronger under-determined system. Because $\lambda = 1$ is equivalent to no *over-* or *under-relaxation* and relaxation is an optional parameter for Kaczmarz and Block-Kaczmarz we use $\lambda = 1$ in the further experiments.

About the observation that Cimmino's algorithm is not always stable, we can imagine that it is related to the fact that after each update of the $\mathbf{x}^{(k)}$ image vector we apply a pre-known constraint

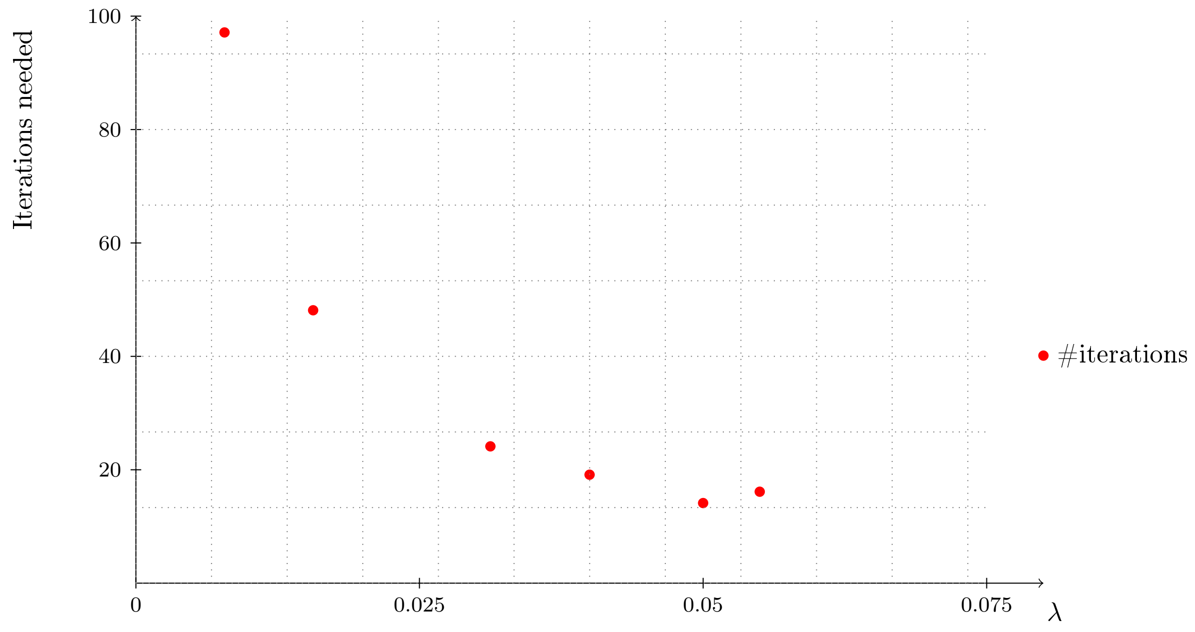


Figure 18: Plot of total number of iterations needed for the implementation of Cimmino's algorithm to achieve the 'quality' $\delta \leq 3000$.

on $\mathbf{x}^{(k+1)}$. We know that $\mathbf{x}$ is a gray-scale image representation and thus: $0 \leq \mathbf{x}^{(k+1)} \leq 255$.

Another source from where the elements of $\mathbf{x}^{(k+1)}$ can run out of the range $[0, 255]$ is that a set of multiple constraints all want to change the same element in $\mathbf{x}^{(k)}$. Therefore it is possible that the value of some elements of $\mathbf{x}^{(k+1)}$ are over updated, because from multiple constraints that element of $\mathbf{x}^{(k+1)}$ will receive a strong increase or decrease.

The convergence of the Block-Kaczmarz for difference block sizes is the subject of the next experiment.

8.2 Block size and convergence of Block-Kaczmarz

As we saw in the discussion of the Block-Kaczmarz algorithm we can see the Kaczmarz algorithm and Cimmino's algorithm as special cases of the Block-Kaczmarz algorithm. In the experiment of Section 8.1 we saw that Cimmino's algorithm is not stable in the scenario where we took 64 directions with 128 rays per direction for a 128×128 pixel picture.

8.2.1 Introduction and method

In this experiment we will analyze the influence of the total number of blocks on the convergence of the Block-Kaczmarz implementation. Therefore we use again the scenario where we want to reconstruct an 128×128 pixel image from projection data of 64 directions with 128 rays per direction. When we apply the Block-Kaczmarz algorithm with a custom number of blocks we use almost equal size partitions.

To get equal size partitions we choose a number of blocks as a power of two. We applied the Block-Kaczmarz algorithm on the 14 partitions with $2^0, 2^1, 2^2, \ldots, 2^{13} = 8192$ blocks.

Also in this experiment we use the δ measure of Equation 10. Again we want a quality of the reconstructed image $\mathbf{x}^{(k)}$ such that $\delta(\mathbf{x}^{(k)}) \leq 1000$.

8.2.2 Results

If we run the Block-Kaczmarz algorithm with the different number of blocks in a partition we get a total number of iterations needed as shown in Table 2. We see that when the number of blocks is greater than 64 we see nothing special on the total number of iterations. Therefore we plot the results with a logarithmic scale on the 'number of blocks'-axis in Figure 19. For a detailed view we plot the data $0 \leq \#\text{Blocks} \leq 64$ on a linear axis in Figure 20.

Number of blocks	Total iterations
8192	43
4096	43
2048	44
1024	44
512	44
256	44
128	44
64	44
54	49
48	59
38	70
32	73
30	164
16	-
8	-
2	-
1	-

Table 2: Total number of iterations needed for different number of blocks in a partition for the Block-Kaczmarz algorithm to achieve the 'quality' $\delta \leq 1000$

To see the convergence for some different block sizes we plot the δ-distance between the result of a given iteration with the original image in Figure 21.

8.2.3 Conclusion

As indicated in Figure 19 we see the total number of iterations needed to achieve a given quality of the image. All runs with #Blocks ≥ 64 need 43-44 iterations. Thus when the size of the blocks are between $\frac{8192}{8192} = 1$ and $\frac{8192}{64} = 128$ rows we have convergence. All these blocks have rays grouped that are from one direction. For each direction we have 128 rays thus when we add additional rows to the blocks then each block is built up of rows that are related to rays from at least two directions. The computations for rays from two different directions can influence each other which can lead to instability and non-convergence of the algorithm.

When we have 32 blocks with $\frac{8192}{32} = 256$ rows all blocks contains the full set of constraints for two directions. This example shows that in this case the algorithm with 32 blocks converges also but they need more iterations to achieve the same quality. This shows that the instability of the system increases. When we do also a run of the Block-Kaczmarz algorithm with 30 blocks we came slowly (in 164 iterations) also to a solution. But when the number of blocks decreases further we see the error is not decreasing in all iterations, as seen in Figure 21.

We see in Figure 21 also that the error of Cimmino's algorithm alternates between two main error values. This is an side-effect of the `Bound` we applied after each iteration. When we ignore the `Bound` in this implementation we see the error increasing in stead of decreasing.

8.3 Different type of input

In this experiment we will look at some parameters of the input from the CT-scanner that will influence the quality of the reconstructed image.

8.3.1 Introduction

The total number of input constraints for image reconstruction algorithms can be seen as depending on two variables. The first variable D is the number of directions from which projections are taken. The second variable R is the number of rays (in the CT-scanner the number of detectors) in a projection. In this experiment we will look how the quality of a image is related to these two variables D and R. This is done by generating images by varying the parameters D and R. We fixed the total number of rays that may be processed to $2 * 128^2 = 32.768$ rays. Thereby we can compare the performance of

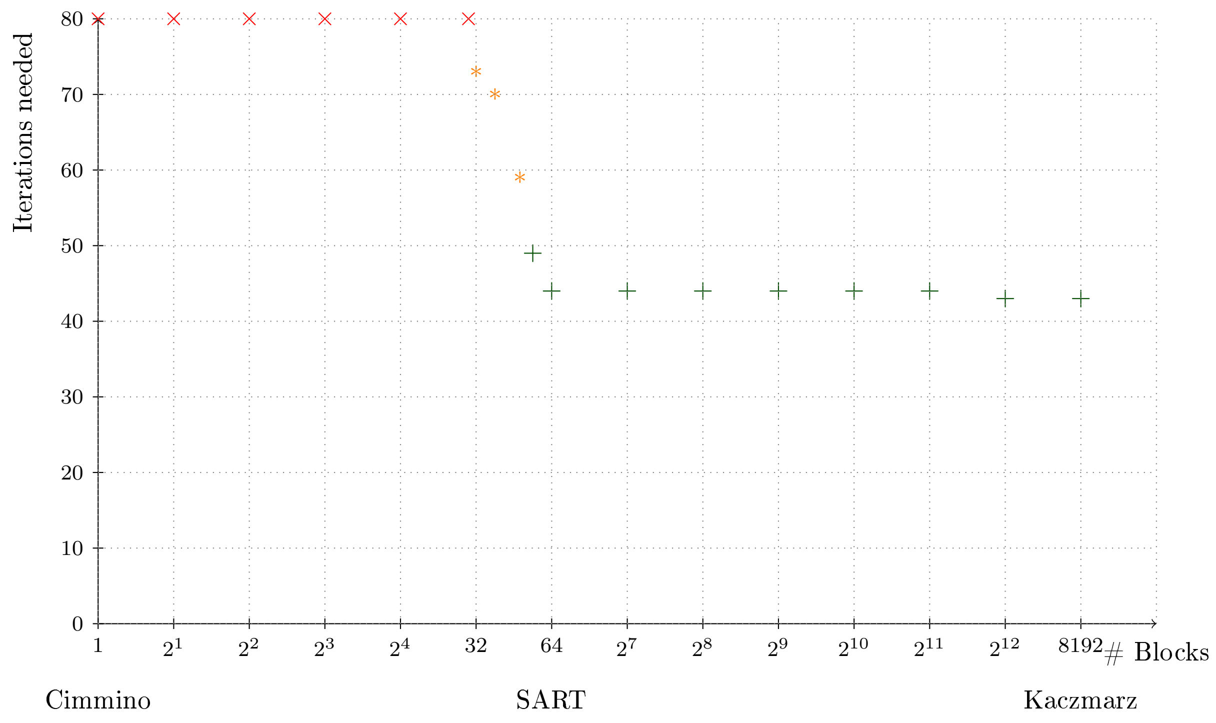


Figure 19: Plot of total number of iterations needed for the Block-Kaczmarz implementation to achieve the ‘quality’ $\delta \leq 1000$ with different block sizes.

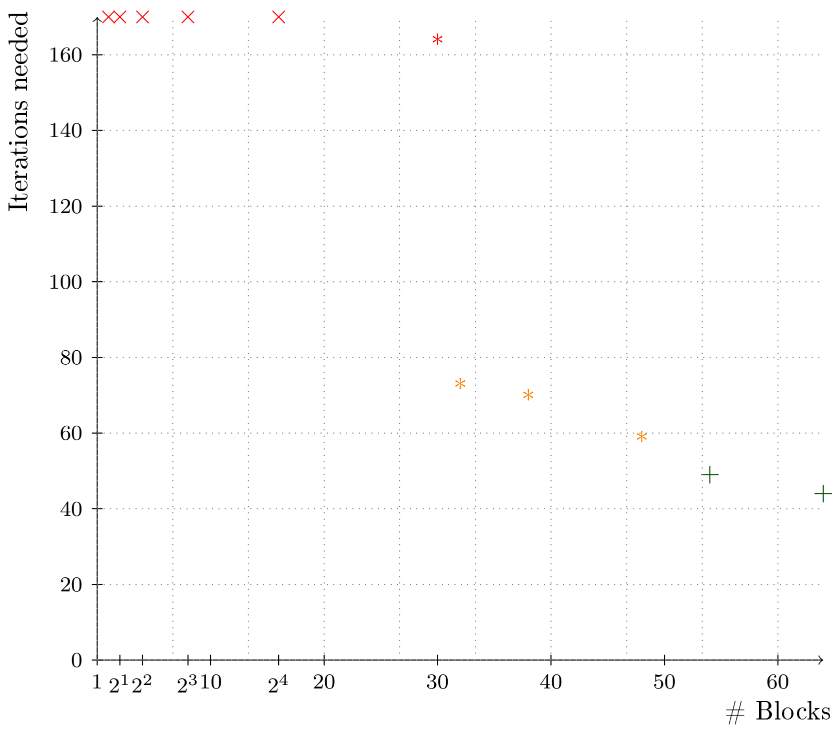


Figure 20: Plot of total number of iterations needed for the Block-Kaczmarz implementation to achieve the ‘quality’ $\delta \leq 1000$ with different block sizes in the range where convergence goes over to non-convergence.

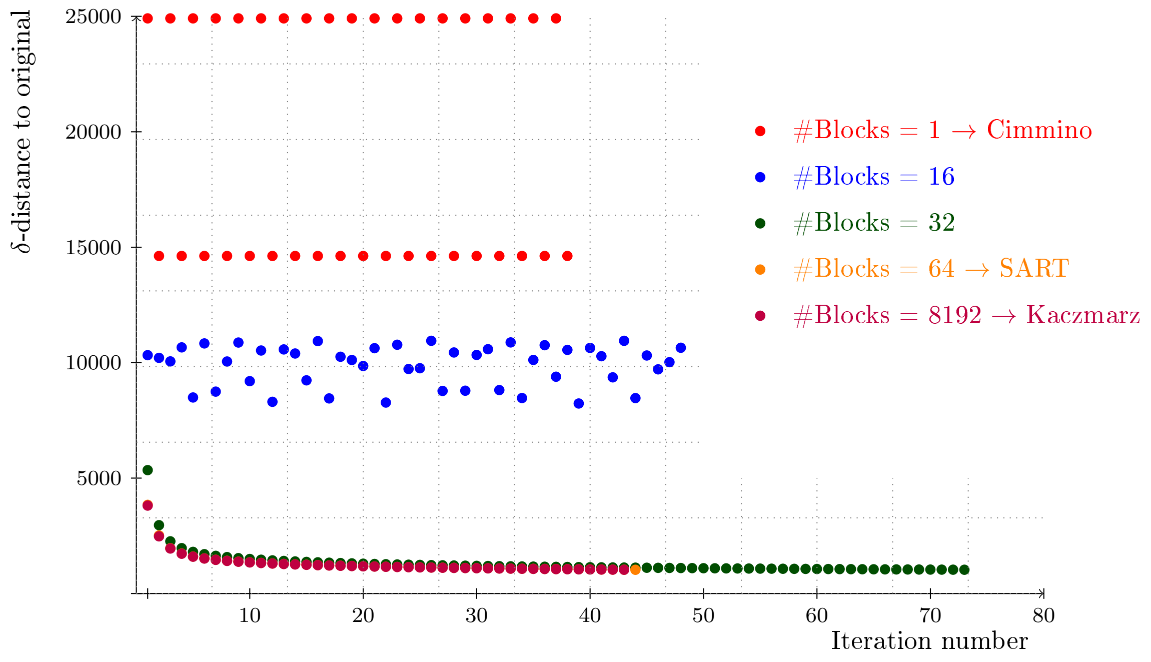


Figure 21: Plot of error as a function of the iteration number for the Block-Kaczmarz implementation with different block sizes.

the algorithms and also the results from different input parameters because we give all runs the same amount of computation resource.

8.3.2 Results

If we apply the Kaczmarz and the Block Kaczmarz algorithm to the given input variables with $(D, R) \in \{(4, 128), (16, 128), (64, 128), (128, 64), (128, 16), (128, 4)\}$ and $\lambda = 1$ then we get result images as in Figure 22 and Figure 23.

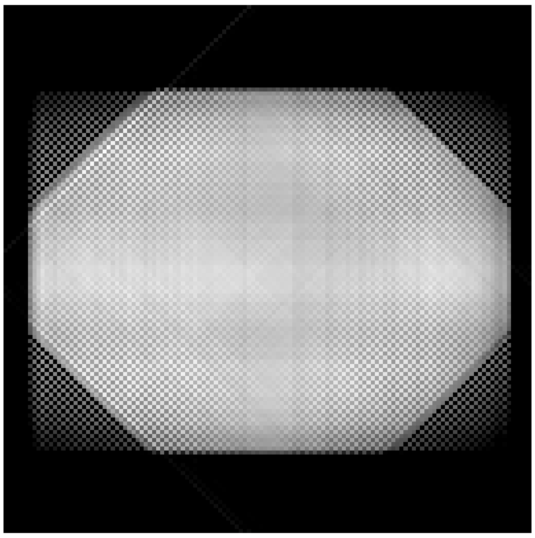
In Figure 22 we see the results of the Kaczmarz and the Block-Kaczmarz algorithm with fixed number of rays per direction. In Figure 23 we see the results of the Kaczmarz and the Block-Kaczmarz algorithm with fixed number of directions.

8.3.3 Conclusion

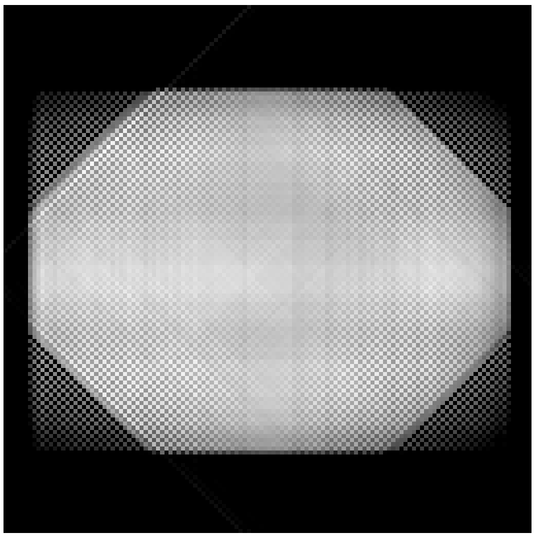
The first conclusion of this experiment is that the result of the Kaczmarz algorithm and the Block-Kaczmarz algorithm is almost the same. Visualy we do not see much difference and only in the computed δ-distance we see for some input parameters a small difference. This leads us to our first conclusion that the Block-Kaczmarz algorithm is a better algorithm for image reconstruction. This is because the Block-Kaczmarz algorithm is prepared to be used for parallel computations. So we can use the power of multiple processors to process all rays of one direction in parallel. The computation of the difference between $\mathbf{x}^{(k)}$ and $\mathbf{x}^{(k+1)}$ can be done in parallel where each processor handles one ray (or row in the system $\mathbf{Ax} = \mathbf{b}$).

The other conclusion of this images is that when D or R increases the quality of the resulting image will also increase. We see also that for a good result we need some balance between D and R. Both variables have its own type of artifact in the resulting images. The quantity D is related to expressing details in the reconstructed image. The quantity R is related to artifacts as like a sort of 'lines' through the image.

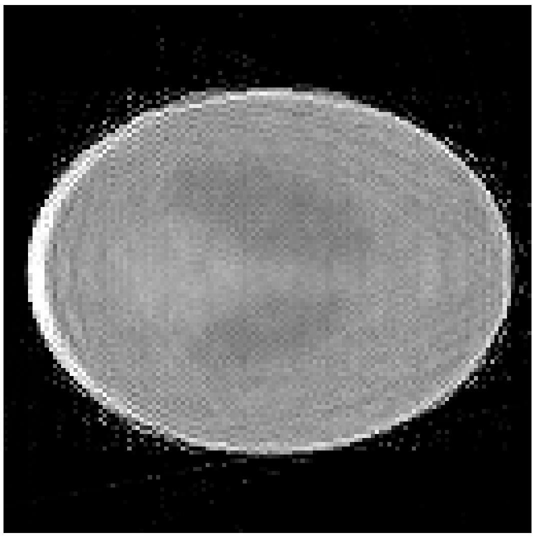
This is because the quantity D stands for the number of directions from which projection information is known. When there are more directions the algorithms can better distinguish gray intensities between pixels that are on a projection line. The quantity R describes something about how 'dense' the projection information is over the image. When the density of the projection data in a image reduces, the projection of that direction describes only for some 'lines' in the image how the gray intensities are distributed.



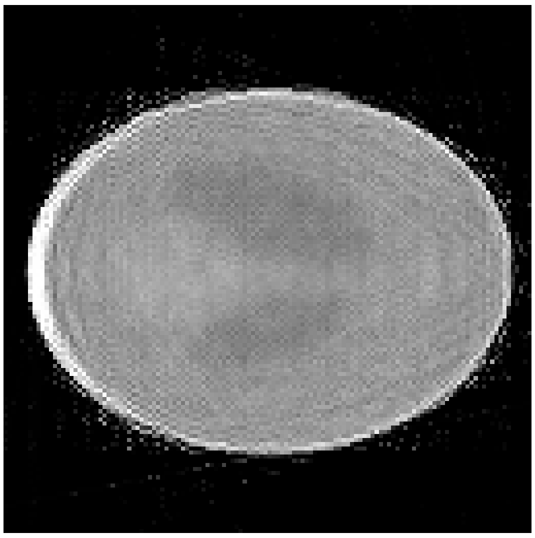
(a) Kaczmarz $D = 4, R = 128, \delta = 5027$



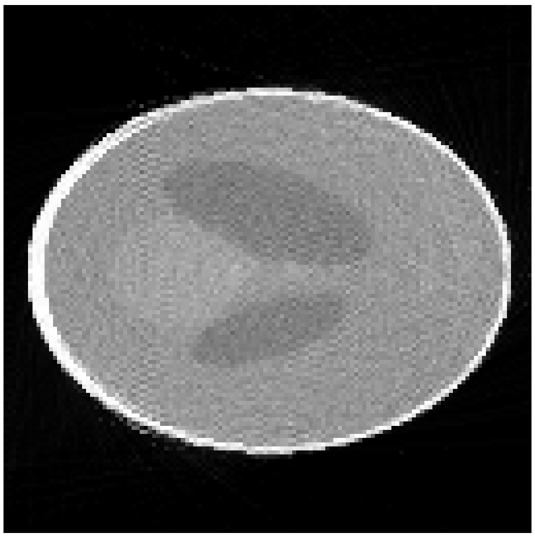
(b) Block Kaczmarz $D = 4$, $R = 128$, $\delta = 5027$



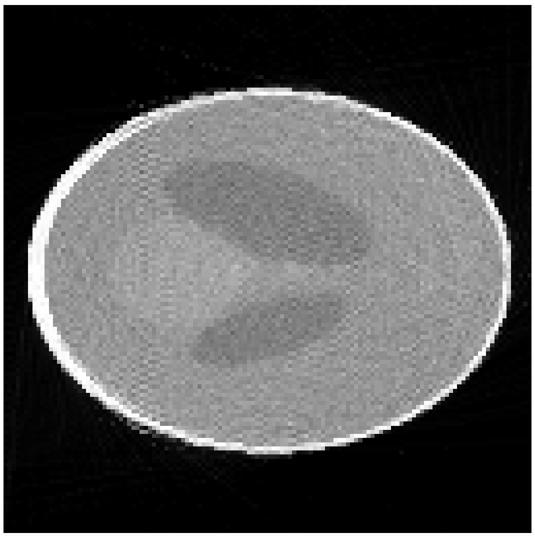
(c) Kaczmarz $D = 16, R = 128, \delta = 2852$



(d) Block Kaczmarz $D = 16$, $R = 128$, $\delta = 2853$

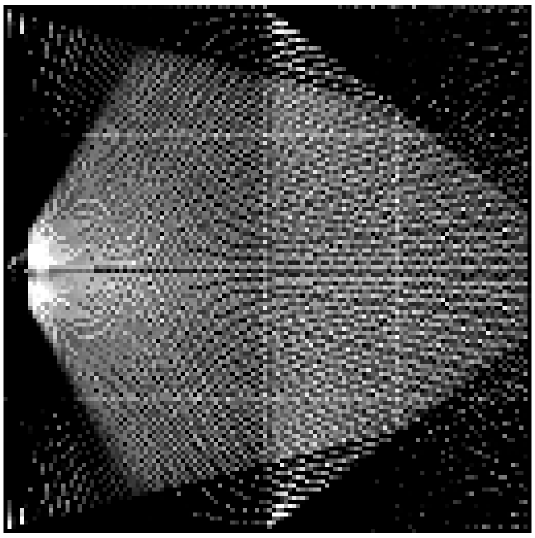


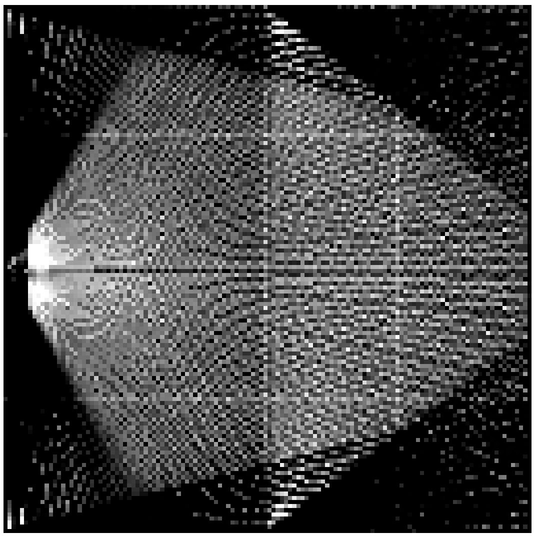
(e) Kaczmarz $D = 64, R = 128, \delta = 1693$

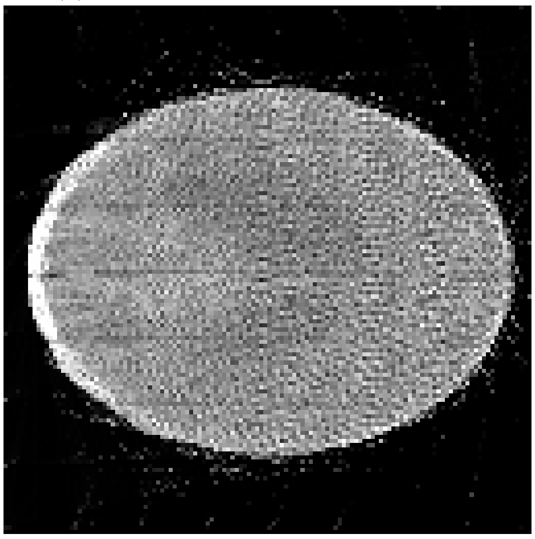


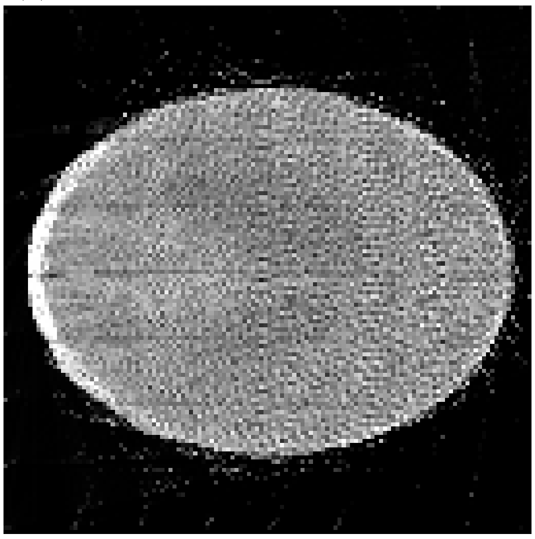
(f) Block Kaczmarz $D = 64$, $R = 128$, $\delta = 1700$

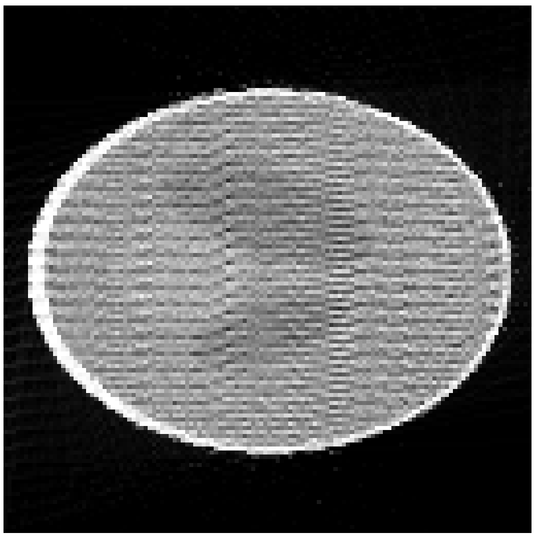
Figure 22: Reconstructed images of the 128×128 head phantom from the Kaczmarz and Block Kaczmarz implementation with a fixed number of rays per direction after processing 32768 rays. This results in 64 iterations for $D = 4$, 16 iterations for $D = 16$ and 4 iterations for $D = 64$.


(a) Kaczmarz $D = 128, R = 4, \delta = 8457$


(b) Block Kaczmarz $D = 128$, $R = 4$, $\delta = 8457$


(c) Kaczmarz $D = 128, R = 16, \delta = 4122$


(d) Block Kaczmarz $D = 128$, $R = 16$, $\delta = 4119$


(e) Kaczmarz $D = 128, R = 64, \delta = 2948$

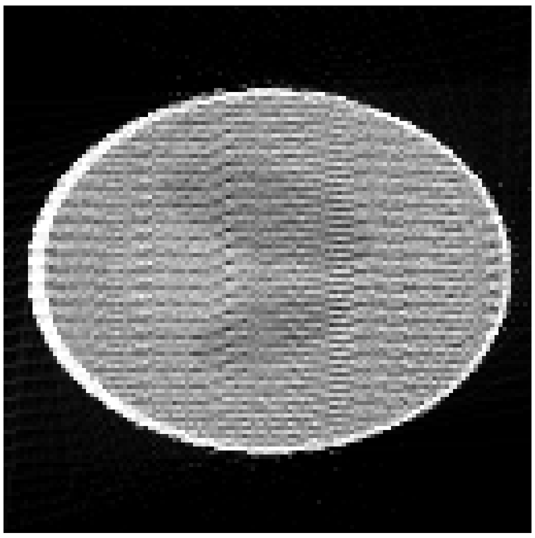
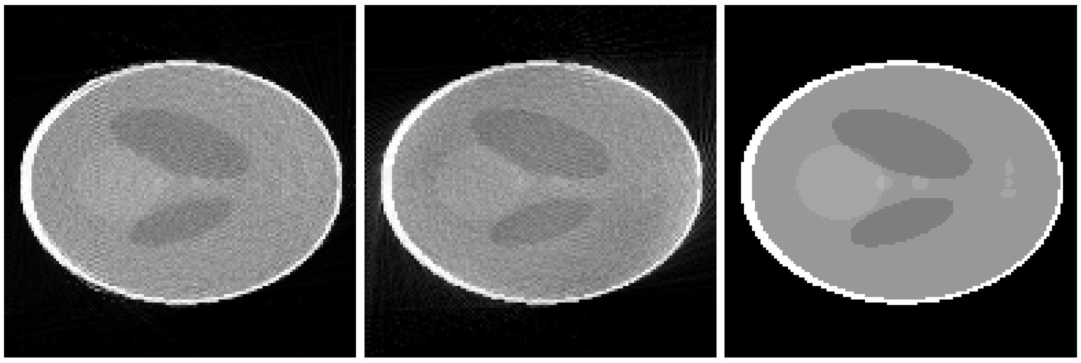

(f) Block Kaczmarz $D = 128$, $R = 64$, $\delta = 2953$

Figure 23: Reconstructed images of the 128×128 head phantom from the Kaczmarz and Block Kaczmarz implementation with a fixed number of directions after processing 32768 rays. This results in 64 iterations for $R = 4$, 16 iterations for $R = 16$ and 4 iterations for $R = 64$.



(a) $D = 64, R = 128, \delta = 1700$ (4 iterations) (b) $D = 128$, $R = 128$, $\delta = 2198$ (2 iterations) (c) Original image $\delta = 0$

Figure 24: Result images of the Kaczmarz algorithm where $D = 128, R = 128$ has insufficient computation resources to get a better quality than $D = 64, R = 128$

Because the human eye is sensitive for differences in gray intensities we want to reduce the 'line'-effects over increase contrast. So more rays gives more information about the original images and will result in a better quality image but when we need to reduce the total number of rays we will advice to choose D and R in such a way that $D \leq R$.

8.3.4 Discussion

The statement "more rays give more information and results in a better quality image" we made in the conclusion is only true when the computation time is unlimited, or at least we can apply multiple iterations of the algorithm. When we have more rays one iteration needs also more time to compute. In general an iteration took $D \times R$ rays to process. When we look for example at the case where $R = 128$ and $D = 128$ the quality of the result image is lower as the quality of the result image of $R = 128$ and $D = 64$, as seen in Figure 24. This is because the quality of the reconstruction increases visually and in δ-distance a lot in the first approximately 4 iterations. This example shows that 4 iterations of the Kaczmarz algorithm of a half determined system results in a better quality image as 2 iterations of a fully determined system.

In this experiment we did not call our parallel Kaczmarz implementation SART. This is because Anderson and Kak describe in Reference [1] more additional features for SART to increase the quality of the result image, and also to decrease D and R. One of the – not implemented – features is the usage of interpolation when they introduced bilinear elements for the approximation of the line integrals. This gives an additional perspective on SART but in this experiment we cannot show the additional performance of these features.

8.4 Controlling the Block-Kaczmarz algorithm with the control sequence

Until now we had said that row action methods we introduced do in each iteration something with one row of the system $\mathbf{Ax} = \mathbf{b}$. The row that is used can be controlled with a control sequence. But we have always used the standard cyclic control sequence: $i_k = k \textbf{ mod } m + 1$, where m is the total number of rows of the system $\mathbf{Ax} = \mathbf{b}$.

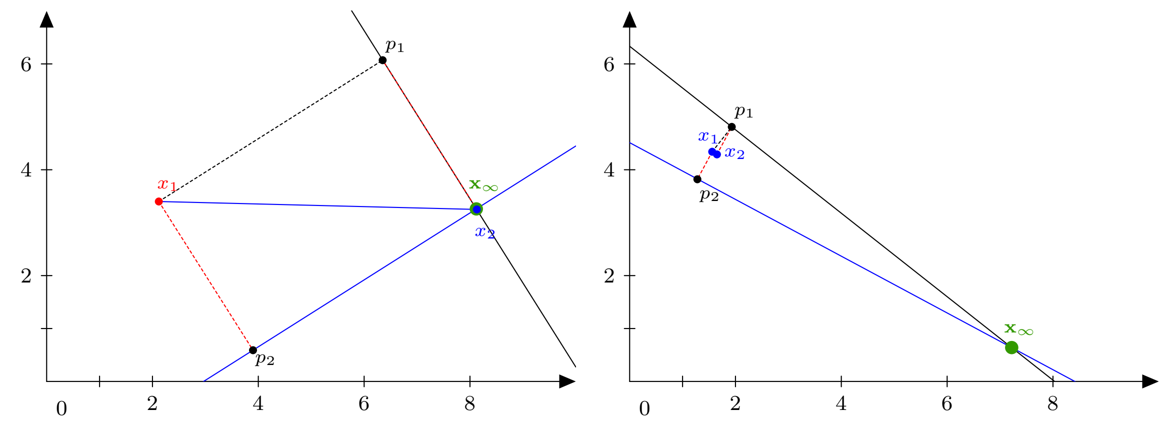
8.4.1 Introduction

In this experiment we will think about another control sequence and use that control sequence to reconstruct images with the Block-Kaczmarz algorithm according to that control sequence.

8.4.2 Hypothesis

As describes in Section 6.3 a control sequence is a sequence of indices of rows in the system $\mathbf{Ax} = \mathbf{b}$. Because we have a finite number of rows we can choose in the sequence this sequence will be cyclic.

As mentioned in the discussion of the Simplex Method and the Kaczmarz algorithm the step in each iteration can be seen as projecting the current solution to the next line, hyperplane or in general boundary of the feasible set. The convergence of the Kaczmarz algorithm depends on the angle



(a) Two constraints are perpendicular, then $\mathbf{x}_2 = \mathbf{x}_\infty$ (b) Angle between constraints low, the algorithm converges very slowly to $\mathbf{x}_\infty$

Figure 25: Sketch of two type of constraints: perpendicular and almost parallel.

Iteration	'Standard control'	'Perpendicular control'
1	3810	3119
2	2486	2120
3	1931	1783
4	1701	1620

Table 3: δ-quality of the reconstruced image for different control sequences for $D = 64, R = 128$

between the boundaries of the feasible set. In 2D we can visualize that with two lines in Figure 25. To increase the convergence of the algorithm we want to project in the next iteration on a boundary that is almost perpendicular to the current boundary.

From the construction of the blocks for the Block-Kaczmarz algorithm we know that the blocks are ascending sorted on projection angle θ in the system $\mathbf{Ax} = \mathbf{b}$. Thus with the idea that in the next iteration we want to project onto a boundary that is almost perpendicular to the current boundary we think that we can increase the convergence of the Block-Kaczmarz algorithm. In the standard cyclic control the algorithm steps through the blocks in ascending order. Thus on each iteration the algorithm projects onto a boundary that has a angle $\frac{180^\circ}{D}$, where D is the total number of projection directions.

So we introduce the following control sequence: Let $c = i \textbf{ mod } m$ be the index of the standard control sequence. Then we divide the range $[0, m]$ into two parts and use the even indices for the fist part and the odd indices for the second part:

$$i_k = \begin{cases} \dfrac{c}{2}, & c \text{ even} \\ \dfrac{1}{2}m + \dfrac{c-1}{2}, & c \text{ odd} \end{cases}$$

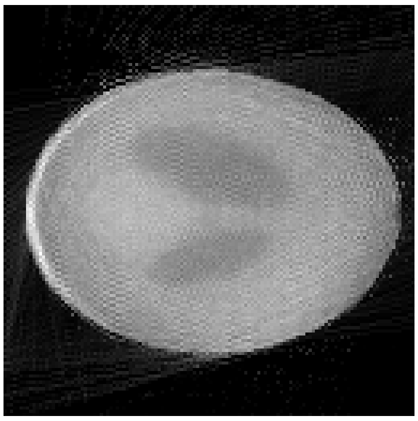
To make the idea of this control sequence clear we require also that m as the number of blocks is even so that we can divide the blocks into two groups of the same size.

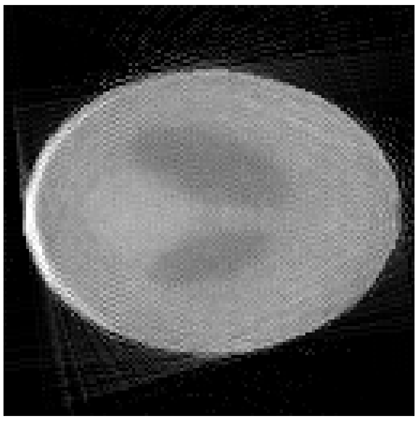
8.4.3 Results

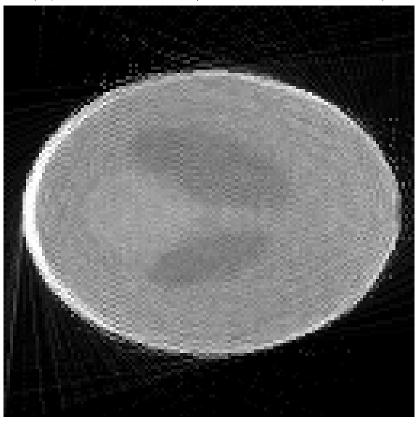
If we look at the intermediate images generated by both versions of the control sequence we see small differences between these two implementations. The intermediate images for $D = 32$ and $R = 128$ of the first four iterations are shown in Figure 26. Visually the differences are not large but if we compute the δ-distances, according to Equation 10, we see some difference as shown in Table 4.

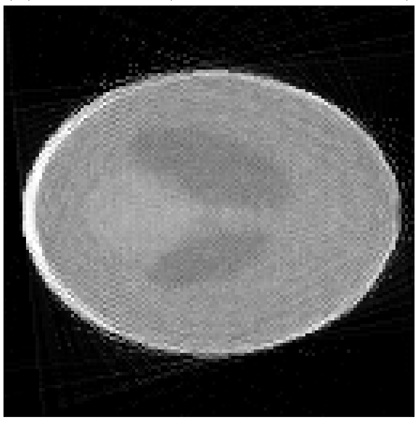
When we apply the input $D = 64$ and $R = 128$ to the *standard control sequence* and the *perpendicular control sequence* after each iteration we get the δ-distance as error to the original image $\mathbf{x}$ as shown in Table 3.

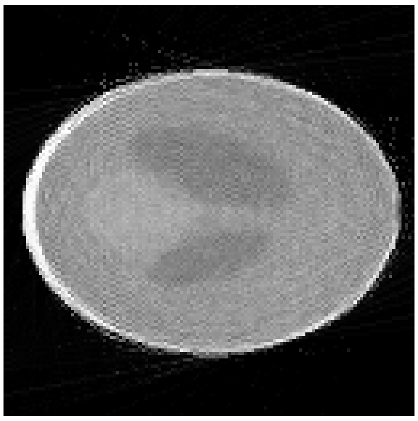
Visually the differences between the pictures are not big, but if we look at the image of the first iteration we can see some differences, see Figure 27.

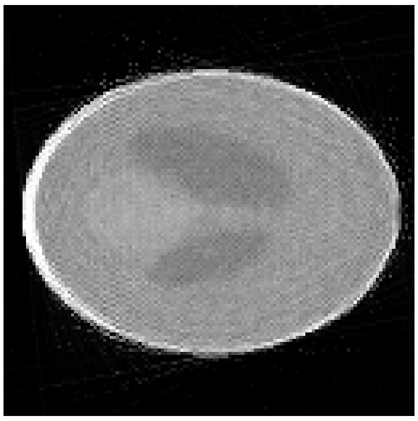

(a) Iteration 1 (Standard control)

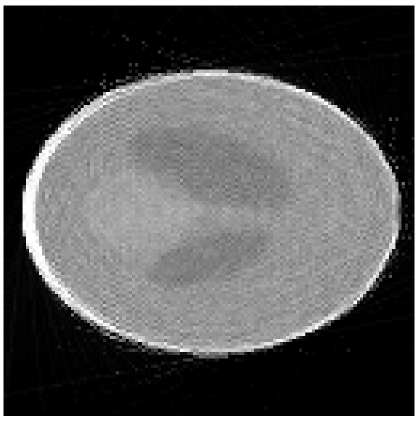

(b) Iteration 1 (Perpendicular control)


(c) Iteration 2 (Standard control)


(d) Iteration 2 (Perpendicular control)


(e) Iteration 3 (Standard control)


(f) Iteration 3 (Perpendicular control)


(g) Iteration 4 (Standard control)

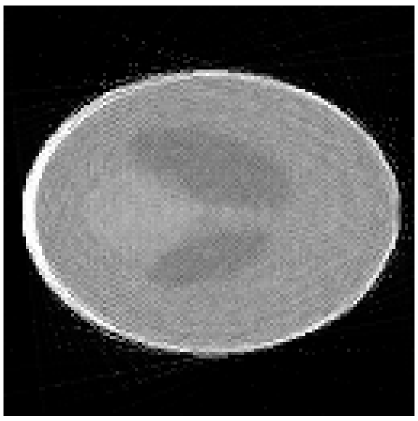
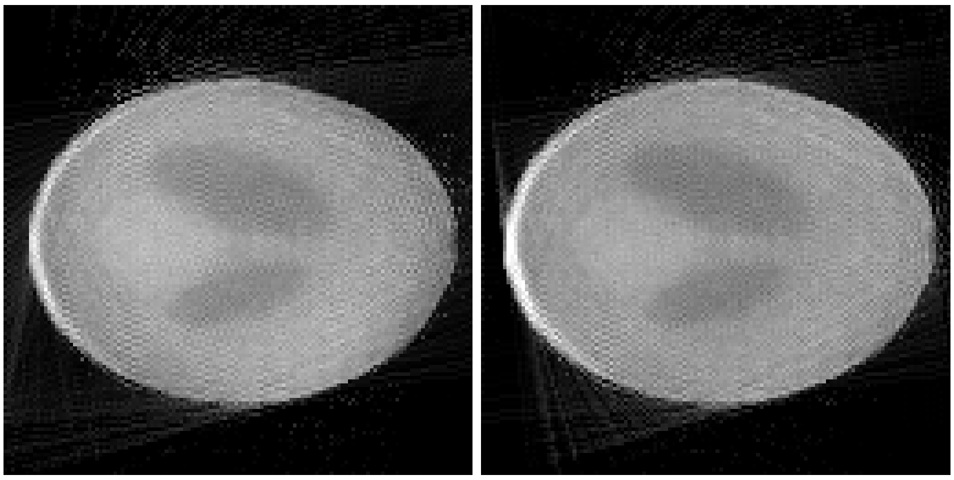

(h) Iteration 4 (Perpendicular control)

Figure 26: Intermediate images in reconstruction process of the 128×128 head phantom from the Kaczmarz implementation with $D = 32$ and $R = 128$.

Iteration	‘Standard control’	‘Perpendicular control’
1	3982, Figure 26a	3491, Figure 26b
2	2958, Figure 26c	2754, Figure 26d
3	2557, Figure 26e	2509, Figure 26f
4	2408, Figure 26g	2384, Figure 26h
5	2312	2301
6	2248	2239
7	2197	2189
8	2154	2148

Table 4: δ-quality of the reconstruced image with different control sequences for $D = 32, R = 128$



(a) Reconstruction with standard control, $\delta = 3982$

(b) Reconstruction with perpendicular control, $\delta = 3491$

Figure 27: Reconstructed images of the ‘head phantom’ after one iteration $D = 32, R = 128$.

8.4.4 Conclusion

The previous two examples show that in δ-distance the first iterations get a better image with the 'perpendicular control' than with the standard control. When the algorithm applies multiple iterations on the image the effect of the 'perpendicular control' will fade out. Thus if we restrict our program to do only a couple of iterations one can think about how the best result can be achieved with the limited computational resource. The notion of 'perpendicular projection' can be one that increases the quality of the resulting image.

8.5 Remarks

From the experiments in this section we can conclude that the Kaczmarz and the Block-Kaczmarz algorithms can be used for image reconstruction. Both algorithms support relaxation but in Section 8.1 we saw that the algorithms are stable enough that they do not need relaxation. Cimmino's algorithm is not always stable as we saw in the analysis of the influence of the block-size. Also Cimmino's algorithm has a slow convergence. The quality of the reconstructed images of Kaczmarz and Block-Kaczmarz are almost the same. We have also seen that when computation time is not bounded we can say that projection information from more directions and with more rays will result in better images.

8.5.1 Parallelization

Our practical analysis discussion came to a point that we can parallelize the Block-Kaczmarz implementation. In each iteration of the Block-Kaczmarz we can divide the rays in a block over a number of processors. The processors can compute the Kaczmarz algorithm for the rows in parallel and this results in a double, like the `d`-variable in our implementations. The resulting variables are given to a master processor that handles the updates of the $\mathbf{x}$ image vector. To reduce the number of inter-process communications the projection matrix $\mathbf{A}$ and the projection vector $\mathbf{b}$ need to be broadcast to all processors in advance. After each update of the image vector $\mathbf{x}$ a processor broadcasts the $\mathbf{x}$ vector to the other processors. Then the next block can be computed. Because most computations can be done in parallel we expect that with enough memory to store the data $\mathbf{A}$ and $\mathbf{b}$ we can get a high speed-up.

The blocks are not the only type of data that can be distributed over multiple processors. Also the parallel geometry gives suggestions for parallelizations. The data collection leads to a parallel execution of a sequential block-iterative method by grouping disjoint subsets of rays which belong to the same view [17]. Also other image decompositions can be found [6].

The Block-Kaczmarz algorithm as result of our discussion is sequentially not an improvement of the Kaczmarz algorithm. But because of the parallelization we think that in this direction the Block-Kaczmarz can become a better iterative algorithm for image reconstruction. The quality of the image stays the same in comparison with Kaczmarz but due to a expected good speed up we can reduce the computation time for one iteration and so the computation time of the whole Block-Kaczmarz algorithm.

9 Conclusion

In this thesis we discussed some mathematical background on optimization and linear programming. From there we saw a first iterative method to solve linear systems: the Simplex Method. We saw also some basic methods for solving linear systems. Because we deal with large systems there are different reasons why we would not use that method to solve our system. For the solution of the systems we looked to row action methods and discussed some of the properties of the Kaczmarz algorithm.

From there we gave the linear system an explicit context in diagnostic medicine. There we saw that computer tomography is a technique that applies something, such as X-rays, to a patient and records the X-rays that are not absorbed by the patient to get projection data. From this projection data we used the Kaczmarz algorithm to reconstruct an image of the interior of the body. We looked also to some alternatives as the Block-Kaczmarz (SART) implementation. This algorithm has the advantage that it can be parallelized.

We considered also the stability of the Block-Kaczmarz method with different block sizes. We saw that the Block-Kaczmarz is stable if we use at least 32 blocks for a configuration with projections from 64 directions and 128 rays per direction. This shows that the Block-Kaczmarz algorithm converges when we use at most information of two projection directions for one update of the image vector $\mathbf{x}$.

From the context of diagnostic medicine there is also a remark on the total number of rays: diagnostic medicine wants to minimize the number of rays. For some diagnostic applications in medicine the rays need to be taken with radioactive and/or other harmful material. With a minimal number of rays the damage can be minimized. But the other side of the discussion is also that the physician needs to recognize the structures on the scan.

This gives a perspective on further research of the Block Kaczmarz algorithm and also the additional features of SART to decrease the number of projection rays and also increase the image quality in one iteration.

References

[1] ANDERSEN, A., AND KAK, A. Simultaneous Algebraic Reconstruction Technique (SART): A superior implementation of the ART algorithm. *Ultrasonic Imaging 6*, 1 (1984), 81 – 94.

[2] BERTSIMAS, D., AND TSITSIKLIS, J. *Introduction to Linear Optimization*, 1st ed. Athena Scientific, 1997.

[3] BRONSTEIN, A., BRONSTEIN, M., AND KIMMEL, R. *Numerical Geometry of Non-Rigid Shapes*, 1 ed. Springer Publishing Company, Incorporated, 2008.

[4] BURDEN, R., AND FAIRES, J. *Numerical Analysis Seventh Edition*. Brooks Cole, 2001.

[5] CENSOR, Y. Row-Action Methods for huge and sparse systems and their applications. *SIAM Review 23* (1981), 444–466.

[6] CENSOR, Y. Parallel application of block-iterative methods in medical imaging and radiation therapy. *Mathematical Programming 42*, 1 (April 1988), 307–325.

[7] COLLEY, S. *Vector Calculus*. Pearson Prentice Hall, 2006.

[8] DANTZIG, G. B. *Linear programming and extensions*. Rand Corporation Research Study. Princeton Univ. Press, Princeton, NJ, 1963.

[9] DANTZIG, G. B., AND THAPA, M. N. *Linear Programming 1: Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[10] EGGERMONT, P., HERMAN, G., AND LENT, A. Iterative algorithms for large partitioned linear systems, with applications to image reconstruction. *Linear Algebra and its Applications 40*, 0 (1981), 37 – 67.

[11] GOODRICH, M. T., AND TAMASSIA, R. *Algorithm design : foundations, analysis, and Internet examples*. Wiley, New York, 2002.

[12] HERMAN, G. T., LENT, A., AND LUTZ, P. H. Relaxation Methods for Image Reconstruction. *Commun. ACM 21*, 2 (Feb. 1978), 152–158.

[13] HORNAK, J. P. The Basics of MRI, 1996.

[14] KAK, A. C., AND SLANEY, M. *Principles of Computerized Tomographic Imaging*. IEEE Press, New York, 1988.

[15] SHEPP, L., AND LOGAN, B. The Fourier reconstruction of a head section. *IEEE Transactions and Nuclear Science NS-21* (1974), 21–43.

[16] SHIRLEY, P., AND MARSCHNER, S. *Fundamentals of Computer Graphics*, 3rd ed. A. K. Peters, Ltd., Natick, MA, USA, 2009.

[17] STAVROS A. ZENIOS, AND YAIR CENSOR. Parallel computing with block-iterative image reconstruction algorithms. *Applied Numerical Mathematics 7*, 5 (1991), 399 – 415.

[18] TESSA, V. H., SARAH, W., MAGGIE, G., JOOST, B. K., AND JAN, S. The implementation of iterative reconstruction algorithms in MATLAB, 2007.

A Code samples

Most of the comments and method information is removed to reduce the size of this code-samples. Also some less important methods are changed or removed.

A.1 Vector implementation

```
namespace IRT.LinearAlgebra
{
    public class Vector
    {
        public int Size { get; set; }
        public double[] Data { get; set; }
        public static Vector operator +(double c, Vector v);
        public static Vector operator +(Vector v1, Vector v2);
        public static Vector operator -(Vector v1, Vector v2);
        public static Vector operator *(double c, Vector v);
        public static double operator *(Vector v1, Vector v2);
        public static Vector operator >=(Vector v, double c);
        public static Vector operator <=(Vector v, double c)
        {
            Vector res = new Vector(v.Size);
            for (int i = 0; i < res.Size; i++)
            { res.Data[i] = v.Data[i] <= c ? 1 : 0; }
            return res;
        }

        public Vector ElementProduct(Vector v)
        {
            Vector prod = new Vector(Size);
            for (int i = 0; i < prod.Size; i++)
            { prod.Data[i] = Data[i] * v.Data[i]; }
            return prod;
        }

        public double VectorProduct(Vector v)
        {
            double prod = 0.0;
            for (int i = 0; i < Size; i++)
            { prod += Data[i] * v.Data[i]; }
            return prod;
        }

        public Vector CrossProduct(Vector v)
        {
            Vector cross = new Vector(v.Size * v.Size);
            for (int i = 0; i < v.Size; i++)
            {
                for (int j = 0; j < Size; j++)
                { cross.Data[i * v.Size + j] = Data[j] * v.Data[i]; }
            }
            return cross;
        }

        public double Norm2() { return VectorProduct(this); }

        public void Bound(double minValue, double maxValue)
        {
            for (int i = 0; i < Size; i++)
            {
                if (Data[i] < minValue) Data[i] = minValue;
                if (Data[i] > maxValue) Data[i] = maxValue;
            }
        }
    }
}
```

Code 5: Vector.cs

A.2 Block-Kaczmarz implementation

```
using System;
using IRT.LinearAlgebra;

namespace IR.Iterative
{
    public class BlockKaczmarz : ReconstructionAlgorithm
    {
        public bool UsePerpendicularControl { get; set; }

        public BlockKaczmarz() : this(false)
        {
        }

        public BlockKaczmarz(bool usePerpendicularControl)
        {
            MaxRowOperations = 128 * 128 * 128;
            SaveInterval = 0;
            Lambda = 1;
            Epsilon = 1000;
            OriginalImage = null;
            UsePerpendicularControl = usePerpendicularControl;
        }

        private int PerpendicularBlockIndex(int iterationId, int numberOfBlocks)
        {
            int c = iterationId % numberOfBlocks;
            if (c % 2 == 0)
                return c / 2;
            else
                return numberOfBlocks / 2 + (c - 1) / 2;
        }

        public override Vector Apply(Matrix A, Vector b)
        {
            int[] blocks = new int[2];
            blocks[0] = 0;
            blocks[1] = b.Size;
            if (UsePerpendicularControl)
                return Apply(A, b, blocks, PerpendicularBlockIndex);
            else
                return Apply(A, b, blocks, (i, m) => i % m);
        }

        public Vector Apply(Matrix A,
                            Vector b,
                            int[] blocks,
                            Vector guessX,
                            ControlDelegate controlSequence)
        {
            Vector x = guessX;
            Vector delta = new Vector(A.SizeN);

            int currentBlockId = 0;
            int blocksProcessed = 0;
            int blockLine = 0;

            for (int i = 0; i < MaxRowOperations; i++)
            {
                int idx = blocks[currentBlockId] + blockLine;

                if (blocks == null || idx == blocks[currentBlockId+1])
                { // We came to a row that belongs to another block

                    // Update the x-vector with the new delta
                    x += delta;
                    x.Bound(0, 255);
                    delta.SetValue(0);

                    // Enter the next block
                    blockLine = 0;
                    blocksProcessed++;
                    currentBlockId = controlSequence(blocksProcessed,
                                                            blocks.Length - 1);

                    // Recompute the idx based on the next block following ctrlSeq
                    idx = blocks[currentBlockId] + blockLine;

                    if (idx == 0)
                    {
                        // We start a new cycle
                        if (Distance(x, i) < Epsilon)
                            break;
                    }
                }

                double b_i = b.Data[idx];
                Vector a_i = A.GetRow(idx);

                // Compute the delta-vector that makes x satisfy the idx-th constraint
                double d = Lambda * (b_i - a_i * x) / a_i.Norm2();
                delta += d * a_i;

                blockLine++;
            }
            return x;
        }
    }
}
```
Code 6: BlockKaczmarz.cs

A.3 Projector implementation

```
using System;
using System.Collections.Generic;
using System.IO;
using IRT.LinearAlgebra;
using IRT.SheppLoganPhantom;

namespace IRT.Projector
{
    public class Projector
    {
        private Vector Image { get; set; }
        private int SqrtSize { get; set; }

        Vector projection;
        Vector projectedFields;
        int projectionFieldId;

        public Projector(Vector image)
        {
            Image = image;
            SqrtSize = (int)Math.Sqrt(Image.Size);
        }

        public Equation GenerateProjections(int numberOfProjections, int numberOfRays)
        {
            List<Vector> projections = new List<Vector>();
            Matrix A = new Matrix(Image.Size);

            for (int p = 0; p < numberOfProjections; p++)
            {
                double theta = p * Math.PI / numberOfProjections;
                List<IntegerPoint2D> linePoints = new List<IntegerPoint2D>();

                if (theta == 0)
                {
                    for (int j = 0; j < SqrtSize; j++)
                    { linePoints.Add(new IntegerPoint2D(0, j)); }
                }
                else if (2 * p == numberOfProjections) // theta = 1/2 pi
                {
                    for (int i = 0; i < SqrtSize; i++)
                    { linePoints.Add(new IntegerPoint2D(i, 0)); }
                }
                else if (2 * theta < Math.PI) // theta < 1/2 pi
                {
                    for (int j = SqrtSize - 1; j >= 0; j--)
                    { linePoints.Add(new IntegerPoint2D(0, j)); }

                    for (int i = 1; i < SqrtSize; i++)
                    { linePoints.Add(new IntegerPoint2D(i, 0)); }
                }
                else // theta > 1/2 pi
                {
                    for (int j = 0; j < SqrtSize; j++)
                    { linePoints.Add(new IntegerPoint2D(0, j)); }

                    for (int i = 1; i < SqrtSize; i++)
                    { linePoints.Add(new IntegerPoint2D(i, SqrtSize - 1)); }
                }

                if (numberOfRays < linePoints.Count)
                {
                    // There are too many starting points for rays, remove starting
                    // points in a uniform maner
                    int lines = linePoints.Count;
                    double stepSize = -1.0 * lines / (lines - numberOfRays);
                    double currentRemoveId = linePoints.Count + stepSize;

                    while ((int)currentRemoveId >= 0)
                    {
                        if ((int)currentRemoveId < linePoints.Count)
                            linePoints.RemoveAt((int)currentRemoveId);
                        currentRemoveId += stepSize;
                    }
                }

                // Compute lines and line-sums
                Box2D field = new Box2D(new IntegerPoint2D(SqrtSize, SqrtSize));
                projectionFieldId = 0;
                projection = new Vector(linePoints.Count);

                foreach (IntegerPoint2D pnt in linePoints)
                {
                    projectedFields = new Vector(Image.Size);
                    IntegerPoint2D.InvokeOnLine(pnt, theta, field, CollectProjection);
                    projectionFieldId++;
                    A.AddRow(projectedFields);
                }

                projection.Save2Text("projections-" + theta + ".txt");
                projections.Add(projection);
            }

            Dictionary<Matrix, List<Vector>> respData = new Dictionary<Matrix, List<
    Vector>>();
            respData.Add(A, projections);

            // Concate the projection-vectors into one big b-vector
            Vector b = new Vector(A.SizeM);
            List<int> blocks = new List<int>();

            int bIndex = 0;

            // At pos 0 the first array will start
            if (projections.Count > 0)
                blocks.Add(bIndex);
            for (int i = 0; i < projections.Count; i++)
            {
                for (int j = 0; j < projections[i].Data.Length; j++)
                {
                    b.Data[bIndex] = projections[i].Data[j];
                    bIndex++;
                }

                blocks.Add(bIndex);
            }

            return new Equation(A, b, blocks);
        }

        private bool CollectProjection(int x, int y)
        {
            if (x < SqrtSize && y < SqrtSize)
            {
                double color = Image.Data[x * SqrtSize + y];
                projection.Data[projectionFieldId] += color;
                projectedFields.Data[x * SqrtSize + y] = 1;

            }
            return true;
        }
    }
}
```

Code 7: Projector.cs