Utrecht University

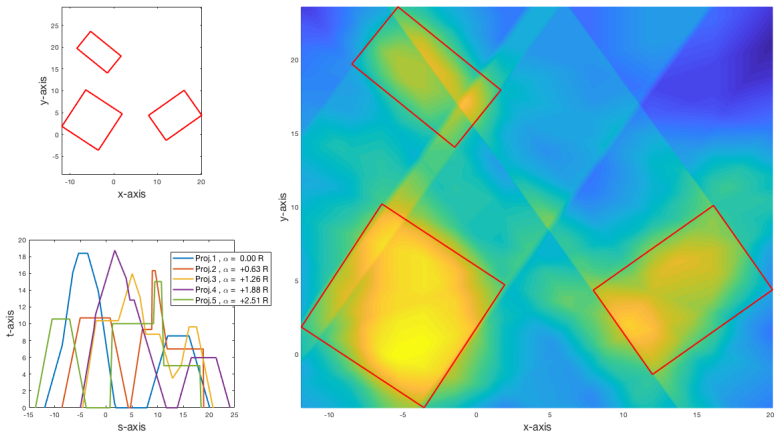
Faculty of Science

Department of Mathematics

Bachelor Thesis

Image reconstruction for rectangular shapes in computed tomography

Reconstructing a CT scan image by a new algebraic method and by binary linear programming on the premise of the image consisting of rectangles



Author:
Janneke Hutter

Supervisor:
Dr. T. van Leeuwen

January 11, 2019

Abstract

Typical computed tomography reconstruction algorithms require the CT scanner to scan around the full 180° of the object, discretizing the image reconstruction problem in order to make an approximation of the original image. We restrict the problem to the premise of non-overlapping rectangles forming the image and compute the exact Radon Transform of the image. Then the inverse problem is: given a certain number of projections, find the configuration of rectangles (of different sizes and orientations) which best match with the projection data. Given this assumption, a new algebraic method is developed. For larger problems (many projections of many rectangles), this method becomes computationally inefficient and instead a binary linear programming problem is formulated. Both methods require only a few CT scan measurements and produce accurate approximations of the original image.

Contents

1 Introduction

What do Egyptian animal mummies, sea ice, sixteenth-century mountain crystal and vascular plants have in common? They were all subjected to X-ray Computed Tomography (CT) scans to reveal their inner secrets and structures.

CT scans are used in a wide range of applications. To the public, CT scans are usually associated with the medical world, and indeed this is a field which greatly benefits from developments in CT scan image reconstruction algorithms. Those advancements, such as a 25 to 80% decrease in radiation dose due to the development of the Iterative Reconstruction method [17], are most prominent in the news.

However, CT scans have many more applications. For example, they are used by the Brooklyn Museum to study Egyptian animal mummies [15], or by the Rijksmuseum to reveal hidden secrets about historic pieces of art, such as sixteenth-century mountain crystal or Chinese puzzle balls [24]. Furthermore CT scans are used to observe the structure of complex food systems on the nanometer scale [14], and detailed plant vascular anatomy [16]. They are also applied to analyze different crystal structures, such as the internal structure of sea ice [13], colloidal crystals made of polystyrene plastic [8], or micro-particles in batteries [11].

With some of these applications, such as analyzing crystal or protein structures, an assumption of the rectangular (in 2 dimensions) or cuboidal (in 3 dimensions) nature of the scanned structure would be quite fitting. This premise is what this paper builds on: using prior knowledge of non-overlapping rectangular shapes forming the image to explore new methods for image reconstruction from CT scan data. This CT scan data consists of many projections, or X-ray scans, along different projection angles. The image reconstruction problem, which is an inverse problem, is thus: given a certain number of projections along different angles, what is the configuration of rectangles which best match these projections?

We will first explore how CT scans work (Sect.2), their relation to the Radon Transform, and the current algorithms to solve the inverse problem by Filtered Back Projection and algebraic reconstruction techniques. These methods require the CT scanner to go around the full 180° and produce approximations based on discretization of the image.

In Sect.3 we first examine the forward problem by developing a method to algebraically compute the exact Radon Transform (or projection function) of an image consisting of rectangles.

We solve the inverse problem algebraically in Sect.4, in the case of one and then two projections of a single rectangle. The existence, uniqueness and stability of the solution found by this algebraic method is explored.

Finally, a method using Binary Linear Programming (abbreviated BLP) is developed in Sect.5. This method involves back projection to find candidate rectangles, formulating the optimization problem, and solving it by the simplex method in combination with branch and bound. A simplified example is given in Sect.5.5 of how the BLP method could be implemented and used to solve the inverse problem.

We see that adding the premise of the rectangular nature of the image allows for incorporation of this knowledge into the image reconstruction problem, resulting in more accurate approximations of the original image with only a few projection measurements.

Author contributions

A new method was developed to compute the exact Radon Transform of an image consisting of rectangles. Thus far only approximations using a discretization of the Radon Transform existed. Under the assumption of the non-overlapping rectangular nature of the scanned objects, an algebraic method was developed to solve the image reconstruction problem in the case of one or two projections of one rectangle. Furthermore, for many projections of many rectangles, a reconstruction algorithm using binary linear programming was developed.

Notation

We will denote vectors with boldface: $\mathbf{x} \in \mathbb{R}^n$, and matrices in bold capital letters: $\mathbf{A}$ is a $m \times n$ matrix. We parametrize a rectangle $R = [x_{\min}, y_{\min}, x_{\max}, y_{\max}, \phi]$ with lower left vertex $[x_{\min}, y_{\min}]$ and top right vertex $[x_{\max}, y_{\max}]$ in normalized pose and ϕ the angle to rotate the normalized rectangle to its original pose. We indicate the image domain in the $x, y-$coordinate system and the projection domain in the $s, t-$ coordinate system. A projection of rectangle R along projection angle α is parametrized by $P_R(s, \alpha) = [s_1, s_2, s_3, h]$, where s_1, s_2, s_3 are breakpoints and h is the maximum height of the projection. Alternatively, we may write a projection as a set of breakpoints S and corresponding function values F: $P_R(s, \alpha) = [S; F]$, and the projection function (Radon Transform) which evaluates the height of projection i along angle α_i for any $s \in \mathbb{R}$ is denoted as $f^{(\alpha_i)}(s)$.

2 CT scans

This section will outline how CT scans work, the relation to the Radon Transform (Sect.2.1), and the existing reconstruction algorithms which can be subdivided into analytic methods (Sect.2.2), and algebraic methods (Sect.2.3).

For one full scan of a slice of an object, the X-ray generator of the CT scanner rotates around the object, and thousands of measurements are gathered by the X-ray detectors which are positioned on the opposite side of the X-ray source. To mathematically model the behavior of the X-rays and the results of the scan, some assumptions are made about the ideal behavior of the X-rays. First, that they are **monochromatic**, meaning the energy level and frequency of propagation is constant. Furthermore, we assume the X-ray beams have zero width and are not diffracted or refracted as they propagate, so they do not bend as they travel through a substance. As each X-ray beam passes through different substances in the image, a proportion of the photons are absorbed. This proportion is called the **attenuation coefficient** μ of the medium [6, p.3]. Now given these assumptions, we can express the observed X-ray beam intensity I after it has passed a distance L through a medium as:

$$I = I_0 e^{-\mu L} \tag{1}$$

where I_0 is the initial beam intensity (in number of photons per second per unit cross-sectional area) [5, p. 11].

If we characterize object R by a continuous function $\mu(x, y)$ on $\mathbb{R}^2$, then the X-ray beam traverses object R from the source to the detector along line L, as shown in Fig.1. The ratio of the observed intensity versus the input intensity can thus be expressed as the following integral along the beam path L [5, p. 12]:

$$\frac{I}{I_0} = e^{-\int_L \mu(x,y)dl} \tag{2}$$

Denoting the projection angle as α and the distance of the X-ray beam to the origin as s, line L traversed by the X-ray beam is thus: $x\cos(\alpha) + y\sin(\alpha) = s$. By taking the natural logarithm of Eq.2 we obtain the **projection function** of object R for s and angle α [5, p.12]:

$$P_R(s, \alpha) = \int_L \mu(x, y)dl \tag{3}$$

When all these projections $P_R(s, \alpha)$ for all angles α (between 0 and π) are combined, we obtain the CT scan data for a single slice (of zero width) of the object. This data is usually visualized with a **sinogram**, which is a greyscale plot of s versus α [6, p.14].

2.1 The Radon Transform

We will now show that the expression of Eq.3 is in fact the Radon Transform of the attenuation coefficient function μ.

Let $h : \mathbb{R}^2 \to \mathbb{R}$ be a function and line L be defined by $x\cos(\alpha) + y\sin(\alpha) = s$ as above, then the **Radon Transform** $\mathscr{R}$ of the function h is defined as the line integral of h along line L [20, p.12]:

$$\mathscr{R}(h)(s, \alpha) := \int_L h(x, y)dl \tag{4}$$

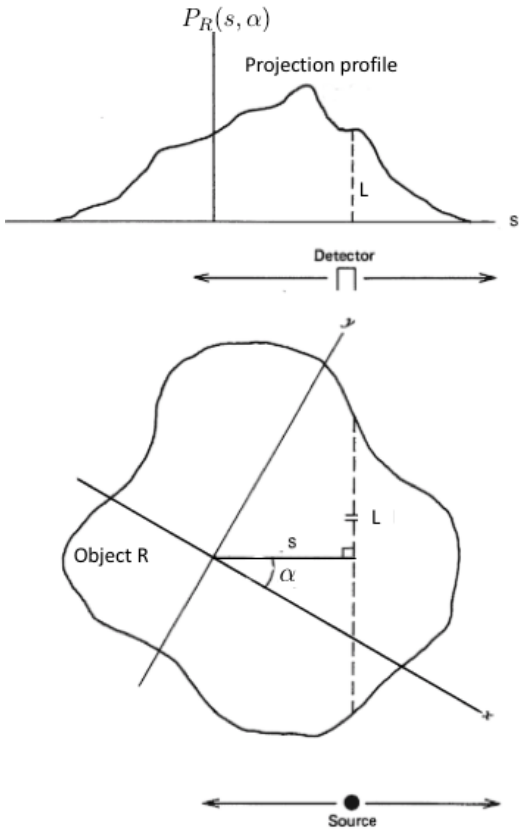


Figure 1: The entire object R is scanned by the X-rays, yielding a projection profile (figure edited, original: [5, Figure 1.6])

If the function $h(x, y)$ is the attenuation coefficient function $\mu(x, y)$, then the projection function of an object R along angle α is the Radon Transform of the attenuation coefficient function:

$$P_R(s, \alpha) = \mathscr{R}(\mu)(s, \alpha)$$

where

$$\mathscr{R}(\mu)(s, \alpha) = \int_L \mu(x, y)dl$$

Thus the Radon Transform forms the mathematical basis of the CT scan data.

Furthermore, the Radon Transform is related to the Fourier Transform by the Central Slice Theorem. First, we define the 1-dimensional Fourier Transform $\mathcal{F}$ of a function h on $\mathbb{R}$ with $\int_{-\infty}^{\infty} |h(x)|dx < \infty$ for any $\xi \in \mathbb{R}$ as [4]:

$$\mathcal{F}(h)(\xi) := \int_{-\infty}^{\infty} h(x)e^{-2\pi i x\xi}dx$$

Likewise, the 2-dimensional Fourier Transform $\mathcal{F}_2$ of a function h on $\mathbb{R}^2$ for any $\boldsymbol{\xi} \in \mathbb{R}^2$ is defined as [4]:

$$\mathcal{F}_2(h)(\boldsymbol{\xi}) := \int_{-\infty}^{\infty} h(\boldsymbol{x})e^{-2\pi i \boldsymbol{x}\cdot\boldsymbol{\xi}}d\mathbf{x}$$

Central Slice Theorem: let h be any suitable function defined on $\mathbb{R}^2$ and s and α be any real numbers, then [6, p.71]:

$$\mathcal{F}_2(h)(s\cos(\alpha), s\sin(\alpha)) = \mathcal{F}(\mathscr{R}(h))(s, \alpha) \tag{5}$$

If we once again identify the function h with the attenuation coefficient function μ, this theorem states that the 1-dimensional Fourier Transform of the Radon Transform of μ in (s, α) is the 2-dimensional Fourier Transform of μ in $(s\cos(\alpha), s\sin(\alpha))$.

Mathematically speaking, if μ were a continuous function, inverting the Radon Transform would yield the unique original image R. However, in practice, the image domain (x, y plane) is discretized and the scans are performed for a finite set of s and α values. Due to the finiteness of the data, the original image can only be approximated [9, p.265]. Tomographic reconstruction techniques are applied to approximate the attenuation coefficient function, which tells us the shape of the scanned object.

Two main tomographic reconstruction techniques are an analytic method called Filtered Back Projection (abbreviated FBP), outlined in Sect.2.2, and algebraic reconstruction techniques (Sect.2.3). The FBP algorithm is relatively computationally undemanding, having a computational complexity of $\mathcal{O}(K^3)$ for an image of $K \times K$ pixels [19]. However the produced images often contain artifacts [3], high noise and impaired image resolution, due to the algorithm's assumption of a perfect scanner and very simplified physical laws of the X-ray interactions. On the other hand, algebraic iterative methods have a much higher computational complexity, but the images have fewer artifacts, less noise and a better resolution [2].

In Sect.4 we develop a new algebraic method to solve the image reconstruction problem and in Sect.5 a method using binary linear programming. Both methods incorporate the prior knowledge of the rectangular nature of the scanned objects. Whereas the existing tomographic techniques require the scan to be done at small intervals all the way around the full 180°, these new methods only need a few projections in order to yield good results, thus saving a lot of computation and scanning time.

We now first outline the existing tomographic techniques, which approximate the original image using no prior knowledge of its shape.

2.2 Filtered Back Projection

The Filtered Back Projection method is computationally much more efficient than algebraic reconstruction techniques. It is currently the most implemented CT scan image reconstruction method and so a short outline of this method follows.

Let $(x_0, y_0) \in \mathbb{R}^2$ be a random point on the Cartesian grid of the image domain. Let $f^{(\alpha_l)}(s) = P_R(s, \alpha_l)$ be the projection function of image R along projection angle α_l for a random projection $l \in 1, ...p$.

Let Tc_{α_l} be the clockwise rotation matrix defined as:

$$Tc_\theta = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \tag{6}$$

Let (s^l, t^l) be the point (x_0, y_0), transformed to the projection domain (s, t plane). Then:

$$\begin{pmatrix} s^l \\ t^l \end{pmatrix} = \begin{pmatrix} \cos(\alpha_l) & \sin(\alpha_l) \\ -\sin(\alpha_l) & \cos(\alpha_l) \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} = \begin{pmatrix} \cos(\alpha_l)x_0 + \sin(\alpha_l)y_0 \\ -\sin(\alpha_l)x_0 + \cos(\alpha_l)y_0 \end{pmatrix}$$

So the intensity of projection l evaluated at (x_0, y_0) is given by:

$$f^{(\alpha_l)}\Big|_{(x_0,y_0)} = f^{(\alpha_l)}(s^l) = f^{(\alpha_l)}(\cos(\alpha_l)x_0 + \sin(\alpha_l)y_0)$$

In other words, for any point $(x_0, y_0) \in \mathbb{R}^2$ and projection angle α_l, the line $L = \cos(\alpha_l)x_0 + \sin(\alpha_l)y_0$ is the unique line (X-ray beam) which passes through point (x_0, y_0) [6, p.39].

In practice, as mentioned earlier, the Cartesian grid of the image domain is discretized and we have a finite number of measurements for finite values of α and s. We sum the intensities of all the X-rays lines which pass through each point on the grid, for all the points on the Cartesian grid. This produces the discrete back projection function [9, p.125].

Formally defined, let $\mathcal{B} : \mathbb{R}^2 \to \mathbb{R}$ be the discrete back projection function, so the total intensity function due to all projections at each point in the image domain. Then for any point (x_0, y_0) on the Cartesian grid, the back projection function is defined as [6, p.113]:

$$\mathcal{B}(f)(x_0, y_0) := \sum_{l=1}^{n} f^{(\alpha_l)}(\cos(\alpha_l)x_0 + \sin(\alpha_l)y_0) \tag{7}$$

where the angles α_l are evenly spaced between 0 and π, so $\alpha_l = \frac{(l-1)\pi}{p}$. Thus the discrete back projection image is obtained by evaluating $\mathcal{B}(f)(x_0, y_0)$ at each point (x_0, y_0) on the discretized image grid.

The resulting back projection image can thus be considered as a smoothed out version of the original image [6, p.72]. Therefore, a convolution filter, which uses the relation between the Radon and Fourier Transforms in Eq.5, is applied to the projection data prior to computing the back projection [9, p.140]. This results in the Filtered Back Projection image, as first a filter is applied to the projection data, and then the back projection function is computed.

2.3 Algebraic reconstruction techniques

There are many different forms of Algebraic reconstruction techniques (ART), but they are all based on the following concept, derived from [6, p.138].

Let the discretized Cartesian grid size of the image to be constructed be $K \times K$ pixels. We number the pixels 1 to K^2 and let x_k be the color or value of the k^{th} pixel.

For any (x, y) in the image domain, we define the pixel basis function as:

$$b_k(x, y) = \begin{cases} 1 & \text{if } (x, y) \text{ lies inside pixel number } k \\ 0 & \text{if } (x, y) \text{ does not lie inside pixel number } k \end{cases}$$

Thus the image can be represented by the function:

$$h(x, y) = \sum_{k=1}^{K^2} x_k \cdot b_k(x, y)$$

A CT scan is the Radon Transform of the image. So for the finite set of J measurements, we define j^{th} measurement p_j taken along the line characterized by (s_j, α_j) as:

$$p_j = \mathscr{R}(h)(s_j, \alpha_j) = \sum_{k=1}^{K^2} x_k \cdot \mathscr{R}(b_k)(s_j, \alpha_j) \text{ for } j = 1, ...J \tag{8}$$

Note that by the definition of the Radon Transform, applied to the pixel basis function, the term $\mathscr{R}(b_k)(s_j, \alpha_j)$ is in fact the length of the intersection of X-ray beam characterized by (s_j, α_j) through pixel k. We denote this intersection length by $r_{jk} = \mathscr{R}(b_k)(s_j, \alpha_j)$ for $j = 1, ..., J$ and $k = 1, ...K^2$. We can thus consider r_{jk} as the "weight" of how much pixel k contributes to projection measurement j. Substituting this into Eq.8, we get:

$$p_j = \sum_{k=1}^{K^2} x_k \cdot r_{jk} \text{ for } j = 1, ...J \tag{9}$$

Let $\mathbf{W}$ be the $J \times K^2$ matrix of elements r_{jk}. Let $\mathbf{p} \in \mathbb{R}^J$ be the projection data vector, and $\mathbf{x} \in \mathbb{R}^{K^2}$ be the vector containing the color values of all the pixels, thus $\mathbf{x}$ is the image we wish to find. Then the system of Eq.9 can be written as:

$$\mathbf{p} = \mathbf{W}\mathbf{x} \tag{10}$$

Thus the image reconstruction problem is written as a system of J linear equations in K^2 unknowns [end of derivation from [6, p.139]]. The formulation of the image reconstruction problem in this form, is the basis for the algebraic reconstruction techniques.

Each X-ray beam passes in a straight line through the object so only a very small fraction of the pixels contribute to each measurement p_i. Therefore, $\mathbf{W}$ is a very sparse matrix. In practice, it is usually also rank deficient and there not invertible. Thus besides the immense computational task of inverting $\mathbf{W}$ if it *were* nonsingular (and thus invertible), we cannot simply compute $\mathbf{x}$ by $\mathbf{x} = \mathbf{W}^{-1}\mathbf{p}$.

There are many iterative reconstruction methods to solve for $\mathbf{x}$ in Eq.10. They all produce a sequence of solution vectors $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, ...$ which converges to $\mathbf{x}^*$ [9, p.193]. How that sequence of solution vectors is generated differs per iterative technique, but thus a general outline of how the solution $\mathbf{x}$ is approximated by iterative reconstruction methods is given.

3 The exact Radon Transform

Before delving into the inverse problem, we first examine the forward problem of computing the Radon Transform of an image consisting of rectangles. The Radon Transform is an integral so usually it is computed by discretizing the integral into a sum and thus approximating the Radon Transform of an image. However, a rectangle is not a nice function to integrate over. Instead, we developed an algebraic approach, with the advantage of the result being the exact, not approximated, Radon Transform of that image.

First, the Radon Transform of a single rectangle is computed algebraically, and then that of several rectangles.

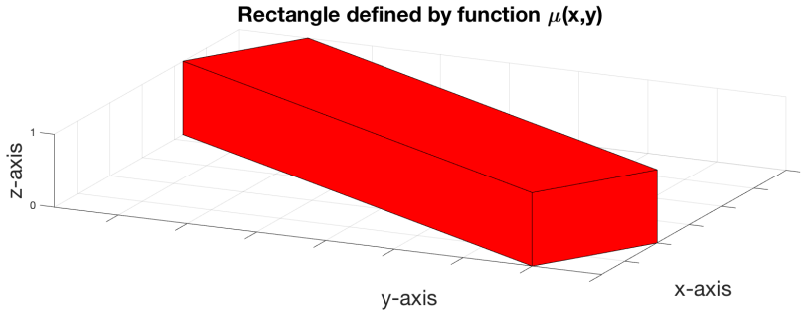


Figure 2: A rectangle is defined by piecewise constant function $\mu(x, y)$

3.1 The Radon Transform of one rectangle

For an image consisting of one rectangle R, we define the attenuation coefficient function $\mu : \mathbb{R}^2 \to \mathbb{R}$ as the piecewise constant function:

$$\mu(x, y) = \begin{cases} 1 & \text{if } (x, y) \in R \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

Fig.2 shows how a rectangle is defined by the piecewise constant function $\mu(x, y)$. Therefore, in the case of a rectangular object, the Radon Transform of μ can be considered as a function of the length travelled by the X-ray beam through the rectangle.

We parametrize a rectangle $R = [x_{\min}, y_{\min}, x_{\max}, y_{\max}, \phi]$, with $[x_{\min}, y_{\min}]$ and $[x_{\max}, y_{\max}]$ being the lower left and upper right vertices respectively in its normalized pose, and ϕ is the angle which rotates the rectangle to its original pose. Let the 2-dimensional transformation matrix T_ϕ be the counter-clockwise rotation by ϕ radians:

$$T_\phi = \begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \tag{12}$$

Then the rectangle in its original pose is obtained by multiplying the rotation matrix T_ϕ with the vertices of the rectangle in its normalized pose.

Let the projection angle α be the angle between the normal to the X-rays (so the s-axis in the projection domain) and the x-axis in the image domain, shown in Fig.3. Equally, α is the angle between the X-rays and the y-axis.

Algorithm 1 *Computing the Radon Transform of rectangle R along angle α (Matlab code in Appx.A.1)*

I Knowing the rotation angle ϕ of the rectangle with the x-axis and X-ray projection angle α, calculate the angle $\theta' = \phi - \alpha$.

II Transform the rectangle to the projection domain (s, t-coordinate system) by multiplying the θ' rotation matrix $T_{\theta'}$ defined as in Eq.12 by the vertices in the normalized pose. Now the X-rays come in vertical lines from above.

III Renumber the rectangle's vertices in the projection domain as: (s_1, t_1), (s_2, t_2), (s_3, t_3), (s_4, t_4), with the s-coordinates sorted in ascending order (shown in Fig.3).

IV Calculate the angle of rotation between the rectangle and the s-axis in the projection domain. This could just be equal to θ', but depending on α and ϕ it could also be negative or larger than $\pi/2$. To compensate for this, calculate $\theta = \theta' \mod (\frac{\pi}{2})$.

V Compute the width w of the rectangle, which is the distance between vertices 1 and 2.

VI Now compute the maximum height h of the projection using simple geometry: $h = w \sec(\theta)$

VII Parametrize the found projection as $P_R(s, \alpha) = [s_1, s_2, s_3, h]$. Using these parameters the Radon Transform can now be written as a piecewise linear function using Lagrange interpolation, explained below.

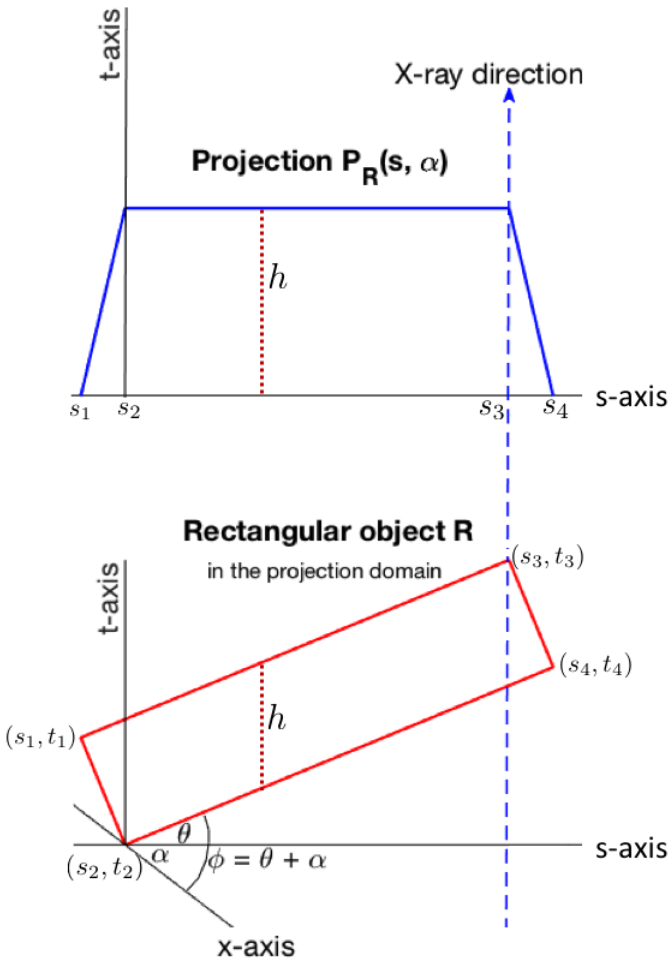


Figure 3: The projection of a rectangle is a piecewise linear function

3.1.1 Lagrange interpolation

The projection function is a piecewise linear function so we rewrite $P_R(s,\alpha) = [s_1, s_2, s_3, h]$ as $P_R(s,\alpha) = [S;F]$ where S is the set of all breakpoints of a projection and F is the set of function values corresponding to those abscissae. So for a projection of one rectangle, $S = \{s_j\}_{j=1}^4 = \{s_1, s_2, s_3, s_4\}$ where $s_4 = s_3 + (s_2 - s_1))$ (due to the symmetry of the projection of a rectangle) and $F = \{t_j\}_{j=1}^4 = \{0, h, h, 0\}$. We use linear piecewise Lagrange interpolation to construct the Radon Transform, or projection function $f^{(\alpha)} : \mathbb{R} \to \mathbb{R}$ that evaluates the "height" of the projection $P_R(s,\alpha) = f^{(\alpha)}(s)$ along angle α for any $s \in \mathbb{R}$.

The linear form for Lagrange interpolation is as follows [1, p.331-332]:

$$f^{(\alpha)}(s) = \sum_{j=1}^{n} t_j L_j(s) \tag{13}$$

where $t_j = f^{(\alpha)}(s_j)$ are the data ordinates that correspond to abscissae s_j and L_j are Lagrange polynomials (to be defined further on), for which must hold [1, p.302-305]:

$$L_j(s_i) = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \tag{14}$$

It is clear that for any given breakpoint s_i, Eq.13 returns the height of the projection at that breakpoint as desired:

$$\begin{aligned} f^{(\alpha)}(s_i) &= \sum_{j=1}^{n} t_j L_j(s_i) \\ &= t_1 \cdot L_1(s_i) + ... + t_i \cdot L_i(s_i) + ... + t_n \cdot L_n(s_i) \\ &= t_1 \cdot 0 + ... + t_i \cdot 1 + ... + t_n \cdot 0 \\ &= t_i \end{aligned}$$

Due to the linear nature of the projections, hat functions are used to define the Lagrange polynomials L_j. Given a random sequence of consecutive breakpoints s_{j-1}, s_j, s_{j+1}, the hat function is defined as [1, p.344-347]:

$$L_j(s) = \begin{cases} \dfrac{s - s_{j-1}}{s_j - s_{j-1}} & s_{j-1} \leq s < s_j \\ \dfrac{s - s_{j+1}}{s_j - s_{j+1}} & s_j \leq s < s_{j+1} \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

So the hat function $L_j(s)$ is one at $s = s_j$ and decreases linearly towards zero to the left and to the right of point s_j, reaching zero when $s = s_{j-1}$ or s_{j+1}. The hat function is implemented in Matlab code in Appx.A.2.

Thus given a random point s between $s_i \leq s \leq s_{i+1}$, evaluating the function $f^{(\alpha)}(s) = \sum_{j=1}^{n} t_j L_j(s)$ comes down to evaluating:

$$\begin{aligned} f^{(\alpha)}(s) &= t_i \cdot L_i(s) + t_{i+1} \cdot L_{i+1}(s) \\ &= f^{(\alpha)}(s_i) \cdot L_i(s) + f^{(\alpha)}(s_{i+1}) \cdot L_{i+1}(s) \end{aligned}$$

since $L_j(s) = 0$ for $j < i-1$ and $j \geq i+1$. Now we can evaluate the projection function for any value $s \in \mathbb{R}$ as follows:

Algorithm 2 *Evaluating the projection function $f^{(\alpha)}(s)$ for any point $s \in \mathbb{R}$ given a parametrized projection $P_R(s, \alpha) = [S; F]$ (Matlab code in Appx.A.3)*

I First verify whether s falls within the domain of the projection function ($s \geq S(1)$ and $s \leq S(n)$, where n is the number of breakpoints of the given projection). If so, continue. Else, return 0.

II Find index i such that s lies between breakpoints s_i and s_{i+1}.

III Calculate the hat function values of L_i and L_{i+1} in s.

IV Multiply these values by the function values $f^{(\alpha)}(s_i) = F(i)$ and $f^{(\alpha)}(s_{i+1}) = F(i+1)$ respectively and sum them up. This yields the value of the projection function at point s.

3.2 Radon Transform of several rectangles

The Radon transform is linear [20, p.8.12], so calculating the Radon Transform, or total projection, of an image consisting of several rectangles along a certain angle α is equal to the sum of the separate projections for each rectangle.

3.2.1 Combining two projections

We first combine the projections of two rectangles, and repeat the process by iteratively adding a new projection to our total projection.

Algorithm 3 *Computing the sum $f^{(\alpha)}$ of two projections (Matlab code in Appx.A.4)*

I Let $P_{R_1}(s, \alpha) = f^{(1,\alpha)}(s) = [S^1; F^1]$ and $P_{R_2}(s, \alpha) = f^{(2,\alpha)}(s) = [S^2; F^2]$ be two projections of different rectangles R_1 and R_2 along the same angle α.

II The new set of breakpoints S is the union of both sets of breakpoints: $S = S^1 \cup S^2$.

III For each $s \in S$, compute $f^{(\alpha)}(s)$ as the sum of the two projection functions evaluated at s using Algorithm 2:

$$f^{(\alpha)}(s) = f^{(1,\alpha)}(s) + f^{(2,\alpha)}(s) \quad \forall s \in S$$

IV All these function values $f(s)$ corresponding to breakpoints $s \in S$ form list F. Thus the combination of the two projections is parametrized by S and F: $P_{(R_1,R_2)}(s, \alpha) = f^{(\alpha)}(s) = [S; F]$.

3.2.2 The total projection

Given an image consisting of n rectangles, the total projection function (Radon Transform of that image) can now easily be computed iteratively.

Algorithm 4 *Computing the Radon Transform of an image consisting of n rectangles (Matlab code in Appx.A.5)*

I Calculate the projection of rectangle i using Algorithm 1.

II If it was the first rectangle ($i = 1$), save its projection as the total projection $P_{R_{\text{total}}}(s, \alpha) = [S; F]$.

III Else, combine its projection with total projection thus far using Algorithm 3.

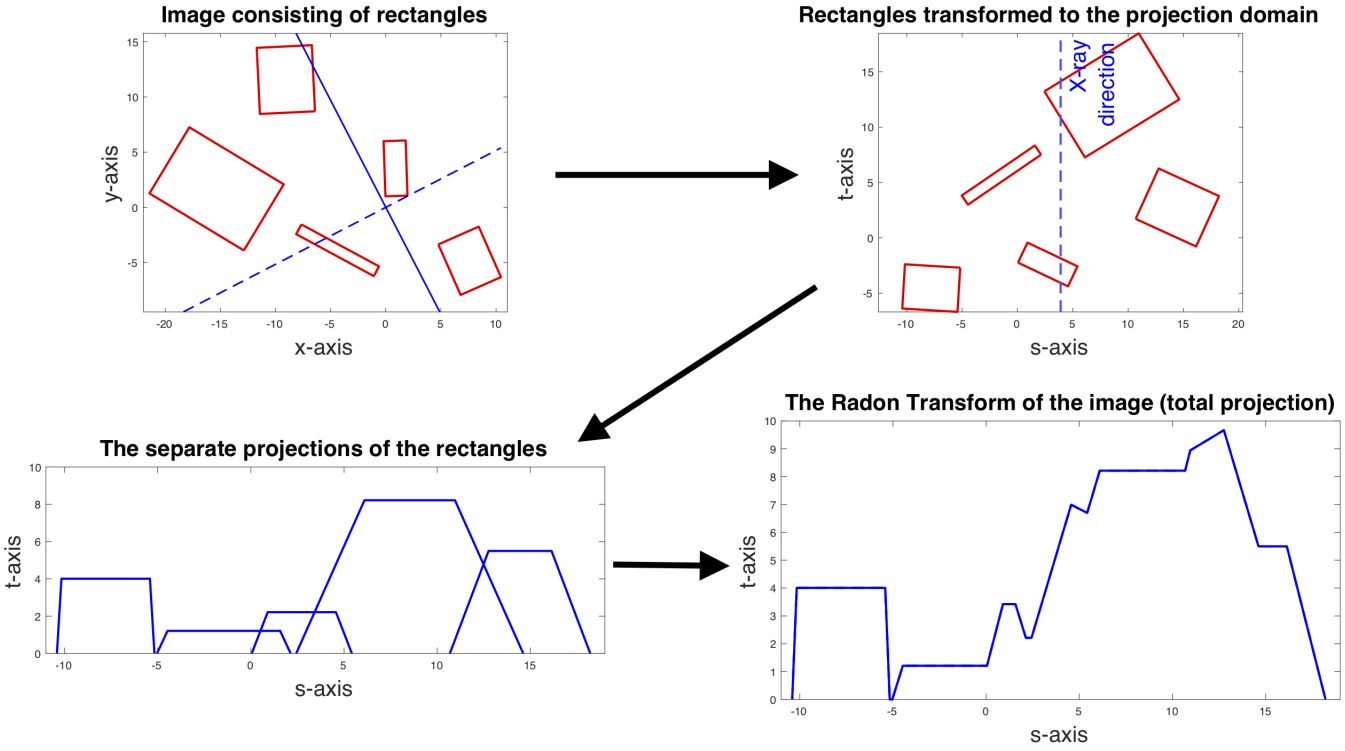


Figure 4: An image consisting of rectangles (X-rays are in the direction of the dotted blue line) and the total projection function, the Radon Transform

IV Save the combined projection as the new total $P_{R_{\text{total}}}(s, \alpha)$.

V Repeat steps I - IV for $i = 1$ to n.

Fig.4 shows five random rectangles in their original poses in the top left corner; the X-rays come parallel to the dotted blue line. The top right corner shows these rectangles transformed by $\theta = \phi - \alpha$ to the projection domain. In the bottom left corner the individual projections along angle α of the rectangles are plotted, and the bottom right corner shows the Radon Transform of the original image, which is the total projection function computed as described above.

4 Algebraic method

We will now develop a new algebraic method to solve the inverse problem of finding the rectangle from one or two projections.

4.1 Computing a rectangle from one projection

4.1.1 The algorithm

Given one projection, we compute at most four possible rectangles which could have produced this projection. Knowledge of the distance of a rectangle from the s-axis is lost during projection, so for simplicity we set all the possible rectangles "on" the s-axis (so $t_i = 0$ for one vertex i and $t_j >= 0 \ \ \forall$ vertices $j \neq i$). Each possible rectangle thus represents infinitely many solutions of that rectangle shifted up and down the $t-$axis. We outline the algebraic method to compute these possible rectangles below. The specific equations for the vertices of the possible rectangles are derived in Sect.4.1.2.

Algorithm 5 *Algebraic method for computing a rectangle from one projection (Matlab code in Appx.A.6)*

I Let $P_R(s, \alpha)$ be a random projection parametrized by $P_R(s, \alpha) = [s_1, s_2, s_3, h]$, with α the angle of the X-rays.

II Let $(s_1, t_1), (s_2, t_2), (s_3, t_3), (s_4, t_4)$ be the vertices of the rectangle(s) we wish to find in the projection domain. The s-coordinates of these vertices match with the breakpoints s_1, s_2, and s_3 of the projection.

III Due to the symmetry of a projection, we calculate s_4 from the other s-coordinates: $s_4 = s_3 + (s_2 - s_1)$.

IV As earlier, let θ be the angle of rotation between the rectangle and the s-axis.

V We differentiate between three different projection shapes, and compute the $t-$coordinates of the corresponding possible rectangles.

Shape 1 is when the projection itself is rectangular (shown in Fig.5). This is the most trivial case as the angle of projection is equal to the angle of rotation of the rectangle, so the rectangle which produced this projection is the same shape as the projection and the angle $\theta = 0$.

For the two remaining projection shapes we solve for angle θ using the following equation (which will be derived below in Sect.4.1.2):

$$r(\theta) = \csc(\theta)\sec(\theta) \tag{16}$$

and setting $r = \frac{h}{s_2 - s_1}$. This yields one or usually two solutions for θ.

Shape 2 is when the projection function has a trapezoidal shape (shown in Fig.6). We first assume that vertex 2 (s_2, t_2) is the lowest vertex (so has the lowest $t-$coordinate) and all other $t-$coordinates are derived from this; this yields two possible rectangles (one for each value of θ). Next we assume vertex 3 is the lowest vertex, also yielding two possible rectangles.

Shape 3 is when the projection is triangular (shown in Fig.7). Then $s_2 = s_3$ and only two different rectangles are possible, one for each angle θ.

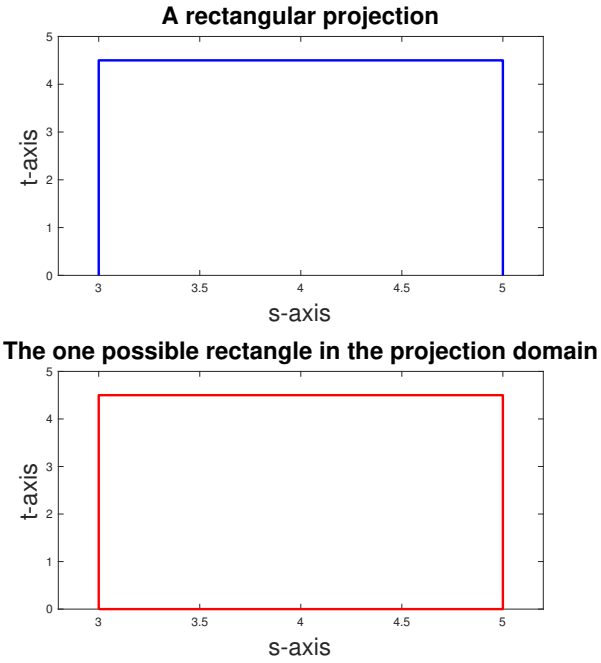


Figure 5: A rectangular projection yields one possible rectangle, representing infinitely many solutions

VI For each calculated possible rectangle, its vertices in the projection domain are put in the following matrix:

$$R_{s,t} = \begin{bmatrix} s_1 & s_2 & s_3 & s_4 \\ t_1 & t_2 & t_3 & t_4 \end{bmatrix}$$

VII Rotate these four vertices clockwise by angle θ, by computing $R_{\text{normal}} = Tc_\theta \cdot R_{s,t}$, where clockwise rotation matrix Tc_θ is defined as in Eq.6. This yields the vertices of the possible rectangle in its normalized pose.

VIII Extract the parameter values $x_{\min}, y_{\min}, x_{\max}, y_{\max}$ for the possible rectangle by minimizing and maximizing the first (x-coordinates) and second (y-coordinates) rows of the matrix R_{normal} respectively.

IX The angle of rotation between the rectangle in its original pose in the image domain and the x-axis is then calculated by $\phi = \alpha + \theta$.

X Thus all the parameters of each possible rectangle are computed:

$$R_{\text{possible, normalized}} = [x_{\min}, y_{\min}, x_{\max}, y_{\max}, \phi]$$

4.1.2 Derivation of the specific equations

Now we will derive the specific equations for step V of Algorithm 5, calculating the $t-$coordinates per projection shape.

Shape 1: A rectangular projection, so $s_1 = s_2$

As explained above, $\theta = 0$ so $\phi = \alpha + \theta = \alpha$. The vertices of the possible rectangle in the projection (s, t) domain are:

$$R_{s,t} = \begin{bmatrix} s_1 & s_2 & s_3 & s_4 \\ 0 & h & h & 0 \end{bmatrix}$$

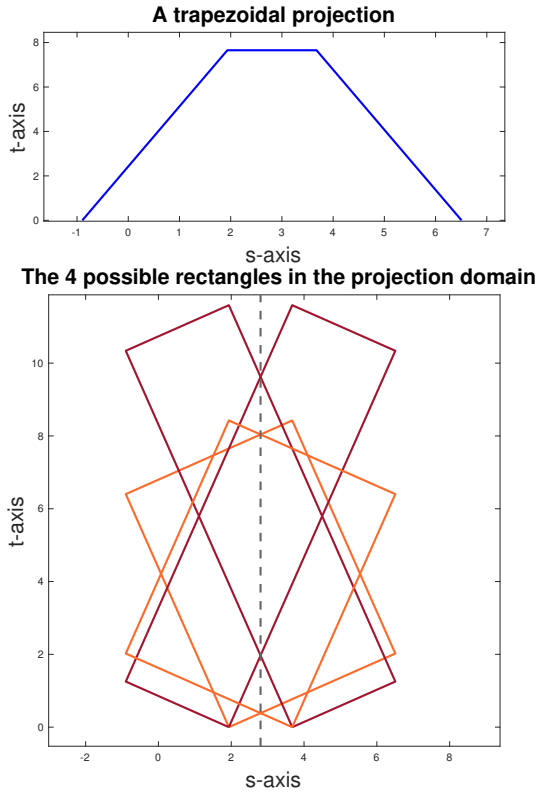


Figure 6: A trapezoidal projection yields 4 possible rectangles

Thus for a rectangular projection, the corresponding possible rectangle has $t-$coordinates:

$$t_1 = 0$$
$$t_2 = h$$
$$t_3 = h$$
$$t_4 = 0$$

Shape 2: A trapezoidal projection, so $s_1 \neq s_2 \neq s_3$

A rectangle with the second vertex being positioned "on" the s-axis and a rectangle with the third vertex on the s-axis both could have produced this projection, so we derive both cases **2a** and **2b** separately.

Case 2a: A trapezoidal projection, setting $t_2 = 0$

We set the second vertex of the possible rectangle "on" the $s-$axis, as shown in Fig.8.

Using basic geometry, we know

$$\tan(\theta) = \frac{t_4 - t_2}{s_4 - s_2} = \frac{t_4}{s_4 - s_2}$$

since $t_2 = 0$. Thus

$$t_4 = (s_4 - s_2)\tan(\theta)$$

Similarly,

$$\tan\left(\frac{\pi}{2} - \theta\right) = \cot(\theta) = \frac{t_1 - t_2}{s_2 - s_1} = \frac{t_1}{s_2 - s_1}$$

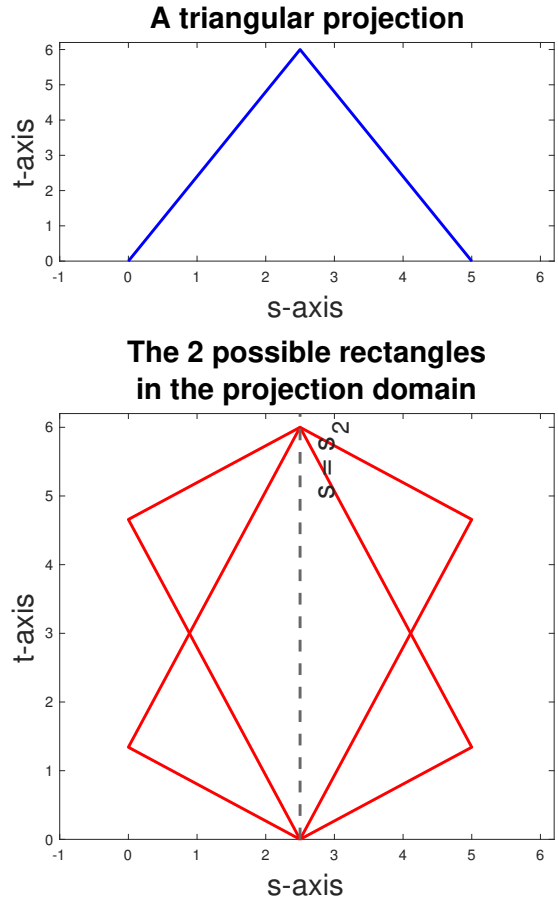


Figure 7: A triangular projection yields 2 possible rectangles

Therefore

$$t_1 = (s_2 - s_1)\cot(\theta)$$

Lastly, $t_3 = t_1 + t_4$. Now we have expressed all the vertices of the possible rectangle in terms of θ, but what is the value of θ?

Solving for θ

In order to derive Eq.16 which is used to solve for θ, we express t_3 in terms of h. Let T be the t-coordinate of the point (s_3, T) on the edge of the rectangle between vertices 2 and 4 (see Fig.8). In other words, as $t_2 = 0$, T is the height between the lowest edge of the rectangle and the s-axis at $s = s_3$. Now

$$\tan(\theta) = \frac{T}{s_3 - s_2}$$

so

$$T = (s_3 - s_2)\tan(\theta)$$

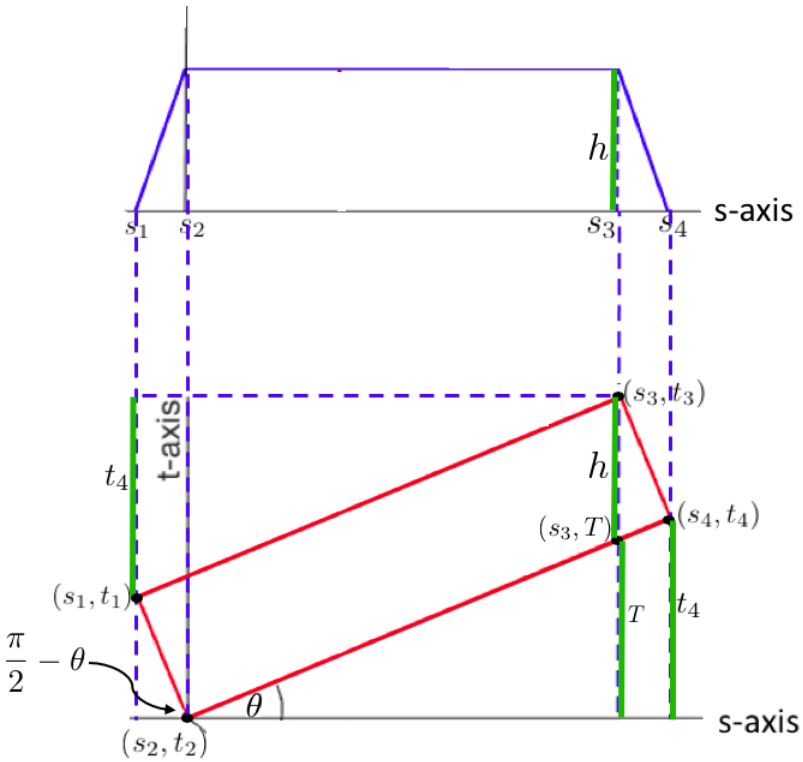


Figure 8: Algebraically computing the rectangle from a trapezoidal projection

Now $t_3 = h + T$ and $t_3 = t_1 + t_4$ must both hold. So:

$$\begin{aligned} t_3 &= h + T = t_1 + t_4 \\ &= h + (s_3 - s_2)\tan(\theta) = (s_2 - s_1)\cot(\theta) + (s_4 - s_2)\tan(\theta) \\ h &= (s_2 - s_1)\cot(\theta) + (s_4 - s_2 - s_3 + s_2)\tan(\theta) \\ &= (s_2 - s_1)\cot(\theta) + (s_4 - s_3)\tan(\theta) \\ &= (s_2 - s_1)(\cot(\theta) + \tan(\theta)) \end{aligned}$$

Therefore

$$\cot(\theta) + \tan(\theta) = \frac{h}{s_2 - s_1}$$

But

$$\cot(\theta) + \tan(\theta) = \csc(\theta)\sec(\theta)$$

Thus

$$\csc(\theta)\sec(\theta) = \frac{h}{s_2 - s_1}$$

we define the function $r(\theta) = \csc(\theta)\sec(\theta)$ as in Eq.16 and solve for θ by computing the inverse of this function.

The function $r(\theta)$ has a minimum at $\left(\frac{\pi}{4}, 2\right)$, as can be seen in Fig.9. For $r > 2$ this equation always has two solutions θ_1 and θ_2 between 0 and $\frac{\pi}{2}$. For $r < 2$ it has zero solutions, so if $r < 2$ we do not have a valid projection and thus cannot compute a rectangle from this projection.

Proof of minimum:

$$r(\theta) = \csc(\theta)\sec(\theta) = \tan(\theta) + \cot(\theta)$$

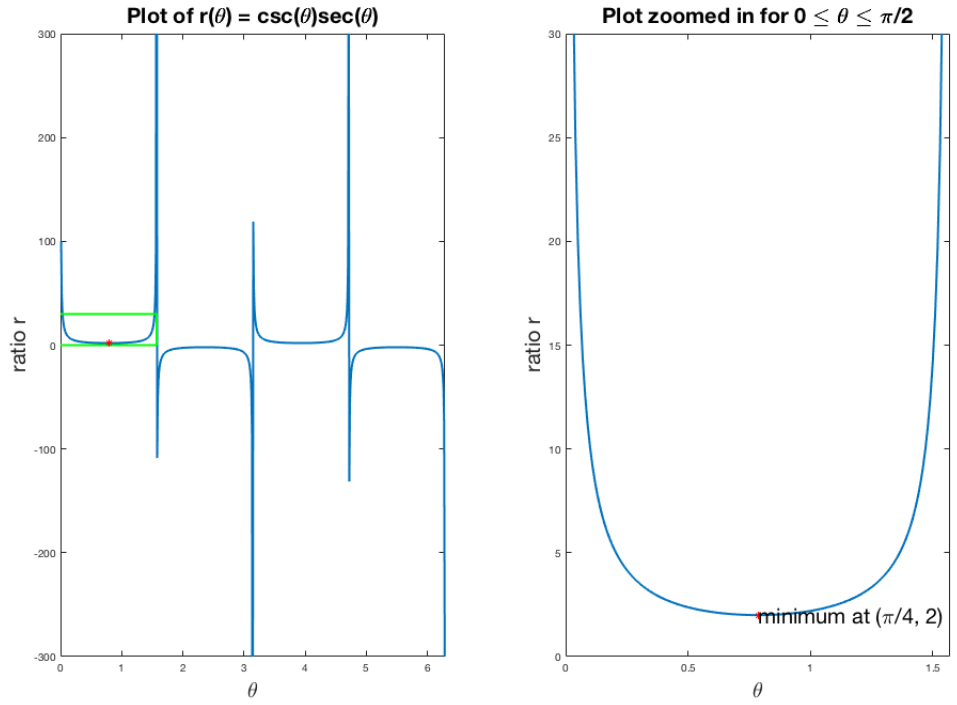


Figure 9: The function $r(\theta)$ from Eq.16

Thus

$$\frac{\mathrm{d}}{\mathrm{d}\theta} r(\theta) = \sec^2(\theta) - \csc^2(\theta) = 0$$

Thus

$$\theta_1 = \pi\left(k - \frac{7}{4}\right)$$
$$\theta_2 = \pi\left(k - \frac{1}{4}\right) \text{ for } k \in \mathbb{Z}$$

So for $0 \leq \theta < \frac{\pi}{2}$ we have $k = 2$ giving us one extremum at $\theta = \frac{\pi}{4}$.

$$r\left(\frac{\pi}{4}\right) = \tan\left(\frac{\pi}{4}\right) + \cot\left(\frac{\pi}{4}\right) = 1 + 1 = 2$$
$$\frac{\mathrm{d}^2}{\mathrm{d}\theta^2} r(\theta) = 2\sec(\theta)(\sec(\theta)\tan(\theta) - 2\csc(\theta)(-\csc(\theta)\cot(\theta))$$
$$= 2\left(\frac{\sin(\theta)}{\cos^3(\theta)} + \frac{\cos(\theta)}{\sin^3(\theta)}\right)$$
$$\cos\left(\frac{\pi}{4}\right) = \frac{1}{\sqrt{2}} \text{ and } \sin\left(\frac{\pi}{4}\right) = \frac{1}{\sqrt{2}}$$

Thus

$$\frac{\mathrm{d}^2}{\mathrm{d}\theta^2} r\left(\frac{\pi}{4}\right) > 0$$

Thus the extremum $\left(\frac{\pi}{4}, 2\right)$ is a minimum ■

So for case 2a we now calculate the angle(s) θ_1 and θ_2 from Eq.16 and setting $r = \frac{h}{s_2 - s_1}$, where s_1, s_2, and h are the given parameter values. Given these one or two values of θ, we

calculate the t-coordinates of the two possible rectangles with vertex 2 being the lowest from the equations derived above:

$$\begin{aligned} t_2 &= 0 \\ t_4 &= (s_4 - s_2)\tan(\theta) \\ t_1 &= (s_2 - s_1)\cot(\theta) \\ t_3 &= t_1 + t_4 \end{aligned}$$

Case 2b: A trapezoidal projection, setting $t_3 = 0$

In this case vertex 3 of the possible rectangle is the lowest vertex, since we set $t_3 = 0$. The equations for the other 3 t-coordinates are derived in a similar manner as in case 2a.

We now have

$$\tan(\theta) = \frac{t_4 - t_3}{s_4 - s_3} = \frac{t_4}{s_4 - s_3}$$

so $t_4 = (s_4 - s_3)\tan(\theta)$.
Likewise

$$\tan(\frac{\pi}{2} - \theta) = \cot(\theta) = \frac{t_1 - t_3}{s_3 - s_1} = \frac{t_1}{s_3 - s_1}$$

thus $t_1 = (s_3 - s_2)\cot(\theta)$ and $t_2 = t_1 + t_4$.

So the $t-$coordinates of the possible rectangle with its lowest vertex being vertex 3 are:

$$\begin{aligned} t_3 &= 0 \\ t_4 &= (s_4 - s_3)\tan(\theta) \\ t_1 &= (s_3 - s_2)\cot(\theta) \\ t_2 &= t_1 + t_4 \end{aligned}$$

But is this angle θ the same as derived for case 2a?

Solving for θ

In order to solve for θ, we now let (s_2, T) be the point on the edge of the rectangle between vertex 1 and 3 at $s = s_2$. Then both $t_2 = h + T$ and $t_2 = t_1 + t_4$ must hold. We have

$$\cot(\theta) = \frac{T - t_3}{s_3 - s_2} = \frac{T}{s_3 - s_2}$$

so $T = (s_3 - s_2)\cot(\theta)$. Thus

$$\begin{aligned} t_3 &= h + T = t_1 + t_4 \\ &= h + (s_3 - s_2)\cot(\theta) = (s_3 - s_2)\cot(\theta) + (s_4 - s_3)\tan(\theta) \end{aligned}$$

So

$$\begin{aligned} h &= (s_2 - s_1)\cot(\theta) + (s_4 - s_3)\tan(\theta) \\ &= (s_2 - s_1)(\cot(\theta) + \tan(\theta)) \text{ due to the symmetry of projections} \\ &= (s_2 - s_1)(\csc(\theta)\sec(\theta)) \end{aligned}$$

Thus we once again have

$$\csc(\theta)\sec(\theta) = \frac{h}{s_2 - s_1}$$

We see that Eq.16 does in fact yield the two different possible angles θ, which in combination with positioning the lowest vertex either at $(s_2, 0)$ or at $(s_3, 0)$ gives us four different

possible rectangles from a trapezoidal projection shape. Unless, as mentioned above, we have $r = 2$ exactly, in which case we have only two possible rectangles (calculated from one angle θ).

Note the symmetry of the possible rectangles, shown in Fig.6: for both angles θ, the rectangle with $t_3 = 0$ (from case 2b) is equal to the one with $t_2 = 0$ (case 2a) reflected in the vertical line $s = \frac{s_2+s_3}{2}$, just as the projection is also symmetrical in this vertical line. Furthermore, $\theta_1 + \theta_2 = \frac{\pi}{2}$ holds.

Shape 3: A triangular projection, so $s_2 = s_3$

This is in fact a special version of case 2a because vertex 2 is the one situated on the s-axis and vertex 3 is situated vertically above it. Thus $t_2 = 0$ and $t_3 = h$ and t_1 and t_4 are as in case 2a:

$$\begin{aligned} t_2 &= 0 \\ t_4 &= (s_4 - s_2)\tan(\theta) \\ t_1 &= (s_2 - s_1)\cot(\theta) \\ t_3 &= h \end{aligned}$$

Once again we have the symmetry between the possible rectangles: they are reflected in the vertical line $s = s_2$ as shown in Fig.7.

4.1.3 Existence of the solution

Given a random projection $P_R(s, \alpha) = [s_1, s_2, s_3, h]$, we have seen above that Eq.16 is crucial in the existence of a solution when computing the possible rectangle(s) from a projection. By constructive proof we have shown that if ratio

$$r = \frac{h}{s_2 - s_1} \geq 2$$

then there exists one or two angles θ which satisfy Eq.16:

$$r = \csc(\theta)\sec(\theta)$$

and possible rectangles (solutions) can be computed from this projection.

4.1.4 Uniqueness of the solution

We have also shown by a constructive proof that when this ratio r is greater or equal to two we can compute one to four possible rectangles, depending on the shape of the projection.

Recall however, that when projecting a rectangle along a certain angle α, knowledge of the distance between the rectangle and the $s-$axis in the projection domain is lost. In our calculations we therefore set the possible rectangle "on" the $s-$ axis and each possible rectangle that is derived thus in fact represents an infinite number of rectangles shifted vertically parallel to the $t-$axis. Therefore, even for projection shape 1 where we have a rectangular projection and compute only one possible rectangle, we still have infinitely many solutions as that rectangle can be shifted up and down the t-axis by any arbitrary amount.

Therefore, when solving the inverse problem of calculating which rectangle produced a certain projection, we always find infinitely many solutions.

4.2 Computing a rectangle from two projections

4.2.1 The algorithm

There are many methods that could be used to find which rectangle produced two projections. After some trials, the following algorithm was developed to compute the original rectangle $R = [x_{\min}, y_{\min}, x_{\max}, y_{\max}, \phi]$, from two different projections $P_R(s, \alpha_1)$ and $P_R(s, \alpha_2)$. The specific equations to implement the algorithm will be derived below in Sect.4.2.2.

Algorithm 6 *Algebraic method for computing a rectangle from two projections (Matlab code in Appx.A.9)*

I From the first projection $P_R(s, \alpha_1)$, compute the at most four different possible rectangles which could have created this projection using Algorithm 5.

II For each possible rectangle, calculate its projection along angle α_2 using Algorithm 1.

III For each of these possible projections from the possible rectangles, compare them to the second projection $P_R(s, \alpha_2)$ and find the possible projection that matches best with projection 2 (using Algorithm 7 below).

IV Thus we know which possible rectangle created projection 1 and 2, let this be

$$R_{\text{found}} = [x_{\min}, y_{\min}, x_{\max}, y_{\max}, \phi]$$

V Find the exact position of the rectangle from the s_1 values of both projections (using Algorithm 8 below).

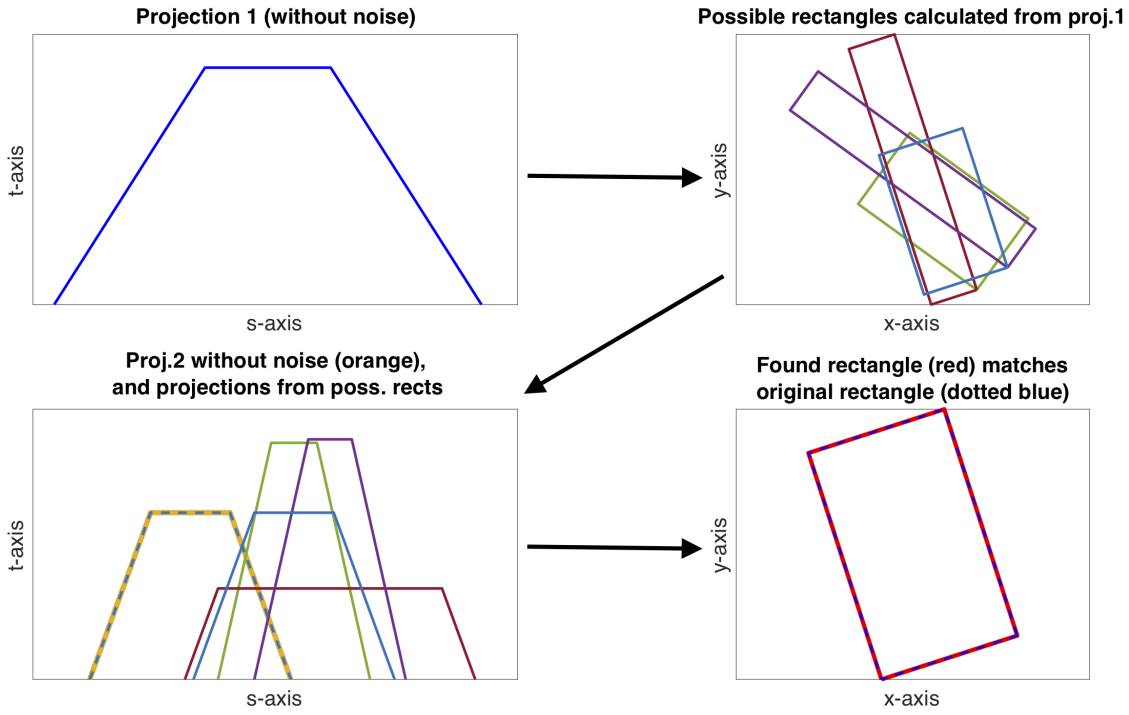


Figure 10: Algebraic method for computing one solution R from two projections

Fig.10 shows projection 1 in the top left corner, and the four possible rectangles which could have produced this projection (in their original poses) in the top right corner. The bottom left corner shows projection 2 in orange, and the projections along angle α_2 of the four possible rectangles in the same colors as their corresponding rectangles. Indeed one of these four projections is identical to the projection 2: this projection is shifted to the left such that the s_1 values are equal and indeed it overlaps perfectly with projection 2 (the shifted projection from one of the possible rectangles is the blue dotted line). Lastly, the bottom right corner shows the shifted found rectangle (in red), overlaid by the original rectangle we wish to find in dotted blue: indeed these rectangles are identical and we have thus by this algorithm found exactly which rectangle R produced the two given projections.

4.2.2 Derivation of the specific equations

The implementation of step III of Algorithm 6 to find which of the at most four possible projections best match the second projection is outlined below.

Algorithm 7 *Matching possible projections to a given projection (Matlab code in Appx.A.7)*

I For each possible projection $P_{\text{poss.proj}} = [s_{1,\text{poss.proj}}, s_{2,\text{poss.proj}}, s_{3,\text{poss.proj}}, h_{\text{poss.proj}}]$ do the following:

- i Calculate the difference in height Δ_h between the possible projection and projection 2: $\Delta_h = h_{\text{proj.2}} - h_{\text{poss.proj}}$
- ii Calculate the shift Δ_s between the projections, which is the difference between the s_1 parameter of the possible projection and that of projection 2. So $\Delta_s = s_{1,\text{proj.2}} - s_{1,\text{poss.proj}}$
- iii Increase the other parameter values s_2, and s_3 by that shift Δ_s: $s_{2,\text{shifted}} = s_{2,\text{poss.proj}} + \Delta_s$ and $s_{3,\text{shifted}} = s_{3,\text{poss.proj}} + \Delta_s$.
- iv Calculate the difference between the shifted s_2 and s_3 parameter values of the possible projection and projection 2: $\Delta_{s_2} = s_{2,\text{proj.2}} - s_{2,\text{shifted}}$ and $\Delta_{s_3} = s_{3,\text{proj.2}} - s_{3,\text{shifted}}$.
- v Compute the match value m of that possible projection with projection 2, which is defined as the sum of the square of the height difference, the s_2 difference and the s_3 difference: $m = \Delta_h^2 + \Delta_{s_2}^2 + \Delta_{s_3}^2$

II The possible projection with the lowest match value m best matches projection 2.

Now the implementation of step V of Algorithm 6 to find the exact position of R_{found} using the s_1 values of projections 1 and 2 follows.

Algorithm 8 *Computing the exact position of a found rectangle (Matlab code in Appx.A.8)*

I Denote Δ_x and Δ_y as the shift in the x and y direction in the image domain respectively such that when the found rectangle is shifted thus and projected along angles α_1 and α_2, the projection parameter s_1 is equal to that of projection 1 and 2 respectively. The aim is thus to find Δ_x and Δ_y.

II Let

$$R_{\text{found, shifted}} = [x_{\min} - \Delta_x, y_{\min} - \Delta_y, x_{\max} - \Delta_x, y_{\max} - \Delta_y, \phi]$$

III Now examine the projection of this rectangle along angle α_1 and α_2. However, only the s-coordinates are needed from this calculation; let those be vectors $\mathbf{s}_{\text{coordinats,proj.1}}$ and $\mathbf{s}_{\text{coordinates,proj.2}}$ respectively. These vectors are of the form

$$\mathbf{s}_{\text{coordinates,proj.}i} = \begin{bmatrix} a + b \cdot \Delta_x + c \cdot \Delta_y \\ d + b \cdot \Delta_x + c \cdot \Delta_y \\ e + b \cdot \Delta_x + c \cdot \Delta_y \\ f + b \cdot \Delta_x + c \cdot \Delta_y \end{bmatrix}$$

for certain constants $a, b, c, d, e, f \in \mathbb{R}$ and $i = 1$ and 2.

IV Due to the form of these vectors $\mathbf{s}_{\text{coordinates,proj.}i}$, in order to find the minimum value of this vector, find the minimum value of the constants of this vector, defined as:

$$\mathbf{s}_{\text{constants,proj.}i} = [a, d, e, f]$$

V The minimum of these constants yields the index or position of the minimum of the vector $\mathbf{s}_{\text{coordinates,proj.}i}$.

VI This minimum should be equal to the first parameter (s_1) of projection i:

$$\min[\mathbf{s}_{\text{coordinates,proj.}i}] = s_{1,\text{proj.}i}$$

for i = 1 and 2.

VII This yields two equations from which the values for the shifts Δ_x and Δ_y are then found.

VIII Substitute these values for Δ_x and Δ_y into $R_{\text{found, shifted}}$ to find the exact position of the rectangle.

4.2.3 Existence and uniqueness of the solution

By constructive proof we have demonstrated above that given two projections of the same rectangle from two non equal angles α_1 and α_2, the exact rectangle which produced these projections can be calculated.

Thus the existence and uniqueness of the solution depends on the precision of numerical computations of Matlab. Let $\alpha_{\text{diff}} = \alpha_2 - \alpha_2$, then the more precise the computations, the smaller the value of α_{diff} can be and still yield the correct solution. When using Matlab's default precision of 16 digits, it was found that the smallest difference α_{diff} from which the unique original rectangle can still be found is $\alpha_{\text{diff}} = 1 \times 10^{-14}$.

4.2.4 Stability of the solution

In reality, the CT scan data of course does not yield perfect piecewise linear projection functions, as the data will contain noise. Given this noisy CT scan data, one would first interpolate it to produce the piecewise linear functions we have worked with so far.

We simulate this noise in the data by adding some random noise $-\epsilon_{\text{noise}} \leq \text{noise} \leq +\epsilon_{\text{noise}}$ to each parameter s_1, s_2, s_3 and h of both projections $P_R(s, \alpha_1)$ and $P_R(s, \alpha_2)$, as shown in Fig.11. We define the difference between the projection angles as $\alpha_{\text{diff}} = \alpha_2 - \alpha_1$. Now we analyze the stability of the solution found using the Algebraic method of Algorithm 6 as follows, for different values of ϵ_{noise} and α_{diff}.

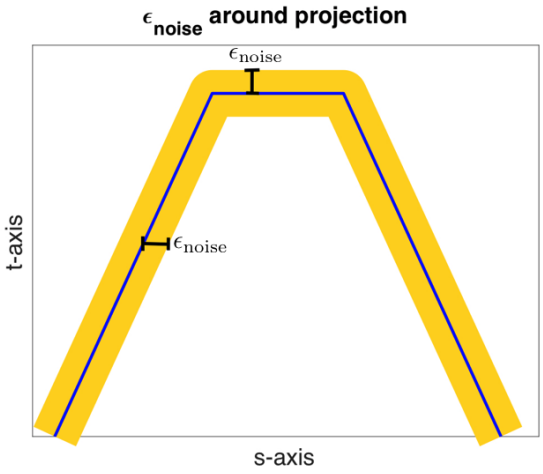


Figure 11: Add ϵ_{noise} to the projection function to simulate noise in the data

Algorithm 9 *Stability analysis of the Algebraic method of Algorithm 6 (Matlab code in Appx.A.10)*

For each value of ϵ_{noise} and α_{diff} :

I Generate a random rectangle R_{original}

II Generate a random projection angle α_1 and compute $P_R(s, \alpha_1)$ using Algorithm 1.

III For $\alpha_2 = \alpha_1 + \alpha_{\text{diff}}$, compute $P_R(s, \alpha_2)$ using Algorithm 1.

IV For each parameter s_1, s_2, s_3 and h of both projections, generate a random noise value $-\epsilon_{\text{noise}} \leq \text{noise} \leq +\epsilon_{\text{noise}}$ and add it to the parameter.

V Compute the rectangle R_{found} from the two distorted projections using Algorithm 6.

VI Compute the error between R_{found} and R_{original}. The error is defined as the Euclidean distance between the four vertices of both rectangles (see Appx.A.11).

VII Repeat steps I - VI 20 times for each ϵ_{noise} and α_{diff} value and calculate the mean error and standard deviation.

Fig.12 shows an example of the stability analysis. The top right corner shows the same projection of Fig.10 in dotted blue, and this projection distorted by noise in blue. The bottom left corner also shows the original second projection in dotted orange, and the noisy projection in orange. We see that due to the noise in both projections, there is no projection from one of the possible rectangles from projection 1 (top right corner) which is identical to projection 2 (in orange). Instead, the possible projection which best matches projection 2 (shown in dotted blue line) is selected, yielding a found rectangle (shown in red in the bottom right corner) which is similar, but not identical, to the original rectangle we wish to find (shown with the dotted red line). The error between the original and found rectangle is shown in yellow.

Fig.13 shows the mean error values for different values of ϵ_{noise}, plotted for different values of α_{diff}. Fig.14 shows the same data, but with the ϵ_{noise} values plotted on a logarithmic scale to better see the data for the smaller ϵ_{noise} values. One can see that the increase in mean error is pretty linear for most values of α_{diff}, as would be expected.

For α_{diff} close to $\frac{\pi}{2}$, the mean error is greater and the solution more unstable. This is corroborated further when looking at Fig.15, which shows the standard deviation bars for the

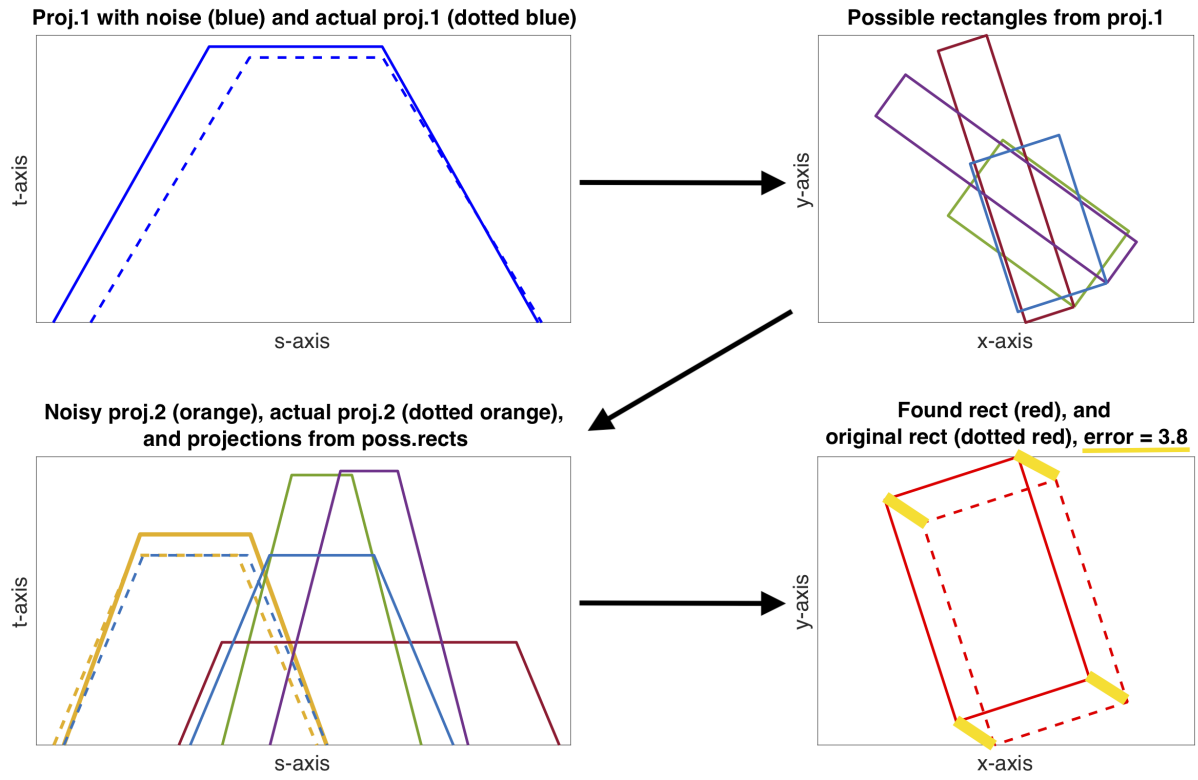


Figure 12: Algebraic method for data with $\epsilon_{\text{noise}} = 0.5$, for $\alpha_{\text{diff}} = \frac{\pi}{5}$

mean values for each value of α_{diff}. The standard deviation for the α_{diff} values close to $\frac{\pi}{2}$ is large for all values of ϵ_{noise}, not just for the larger ϵ_{noise} values as with the other plots.

The instability of the solution for (almost) orthogonal projections, so α_{diff} values close to $\frac{\pi}{2}$, was to be expected as the algorithm which finds the rectangle first computes the four (or less) possible rectangles from projection 1, then projects those possible rectangles along angle α_2. However, the possible rectangles come in symmetric pairs: they are reflected in the line $s = s_0$ for a certain value s_0. So projecting the possible rectangle along angle $\alpha_2 = \alpha_1 + \frac{\pi}{2}$, almost perpendicular to the original projection angle, would yield identical projections for each pair of symmetrical possible rectangles. The algorithm then compares which of the four projections from the possible rectangles best match projection 2. However, since α_{diff} is close to $\frac{\pi}{2}$, the projections of the four possible rectangles have merged into two possible projections, and thus the algorithm cannot successfully distinguish which of the two rectangles from a symmetric pair is the best match. This accounts for the large error (almost) orthogonal projections.

With similar reasoning, one would also expect that α_{diff} values close to $\frac{\pi}{4}$ would also yield higher errors. This is because if a possible rectangle is close to square in shape, there is even more symmetry as the projections of a symmetric pair of possible rectangles along $\alpha_2 = \alpha_1 + \frac{\pi}{4}$ would also yield identical projections. Although one can see a slight increase in mean error for $\alpha_{\text{diff}} = \frac{\pi}{4}$ and slightly higher standard deviation values, the effect is by far not as dramatic as for α_{diff} values close to $\frac{\pi}{2}$ because this effect is only when dealing with (almost) square rectangles.

Lastly, one can also see in Fig.15 that for all the non-orthogonal α_{diff} values the standard deviation increases dramatically for $\epsilon_{\text{noise}} \geq 0.5$. So for $\epsilon_{\text{noise}} < 0.5$ the solution is reasonably stable for non-orthogonal projections, and for $\epsilon_{\text{noise}} \geq 0.5$ the solution is definitely unstable and thus unreliable, regardless of α_{diff} value.

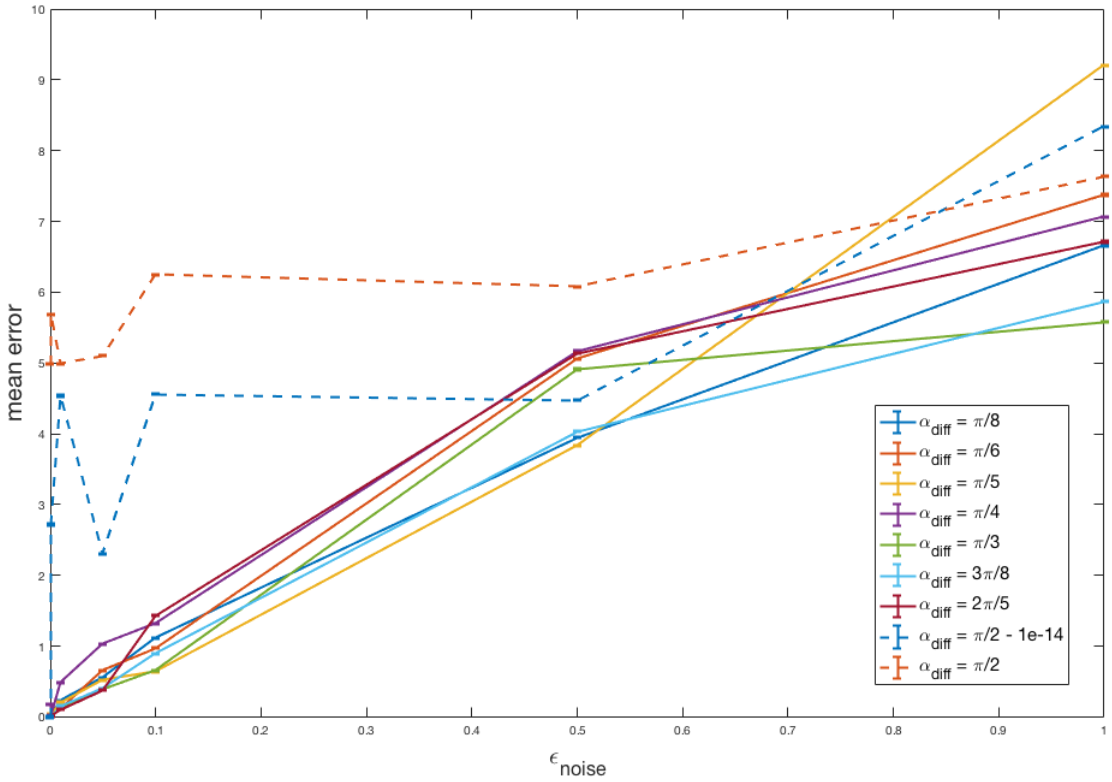


Figure 13: Mean error from 20 trials of the Algebraic method, for different values of α_{diff}

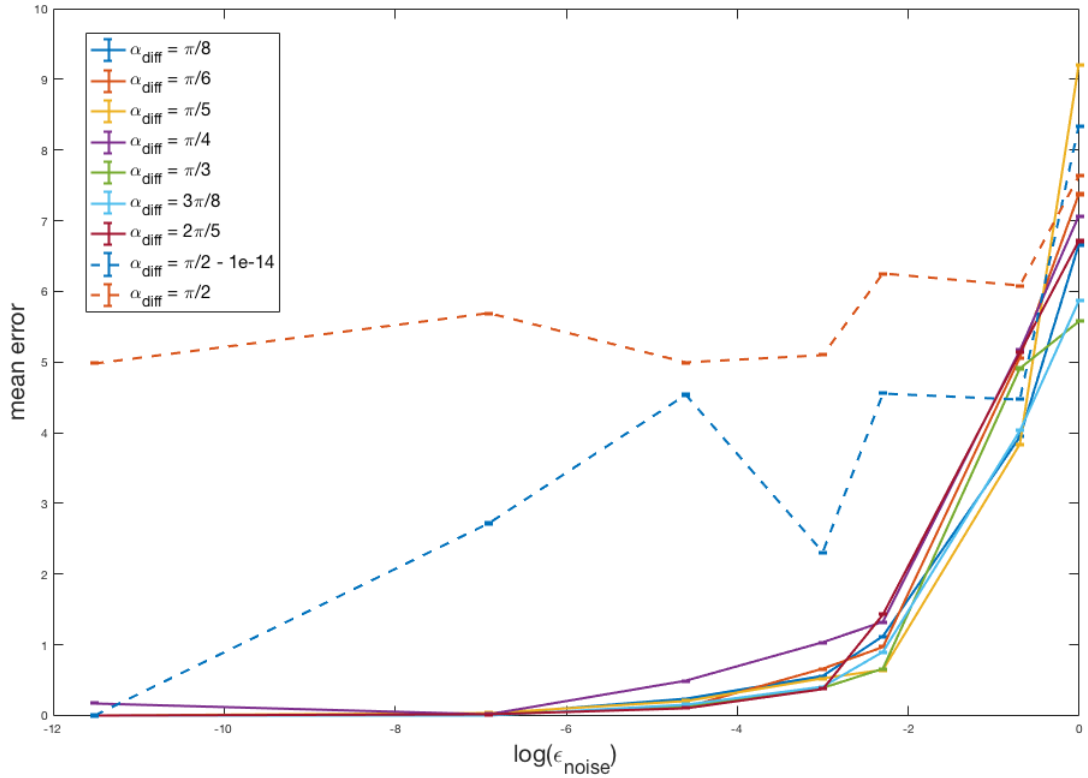


Figure 14: Mean error from 20 trials of the Algebraic method, for different values of α_{diff}, on a logarithmic scale

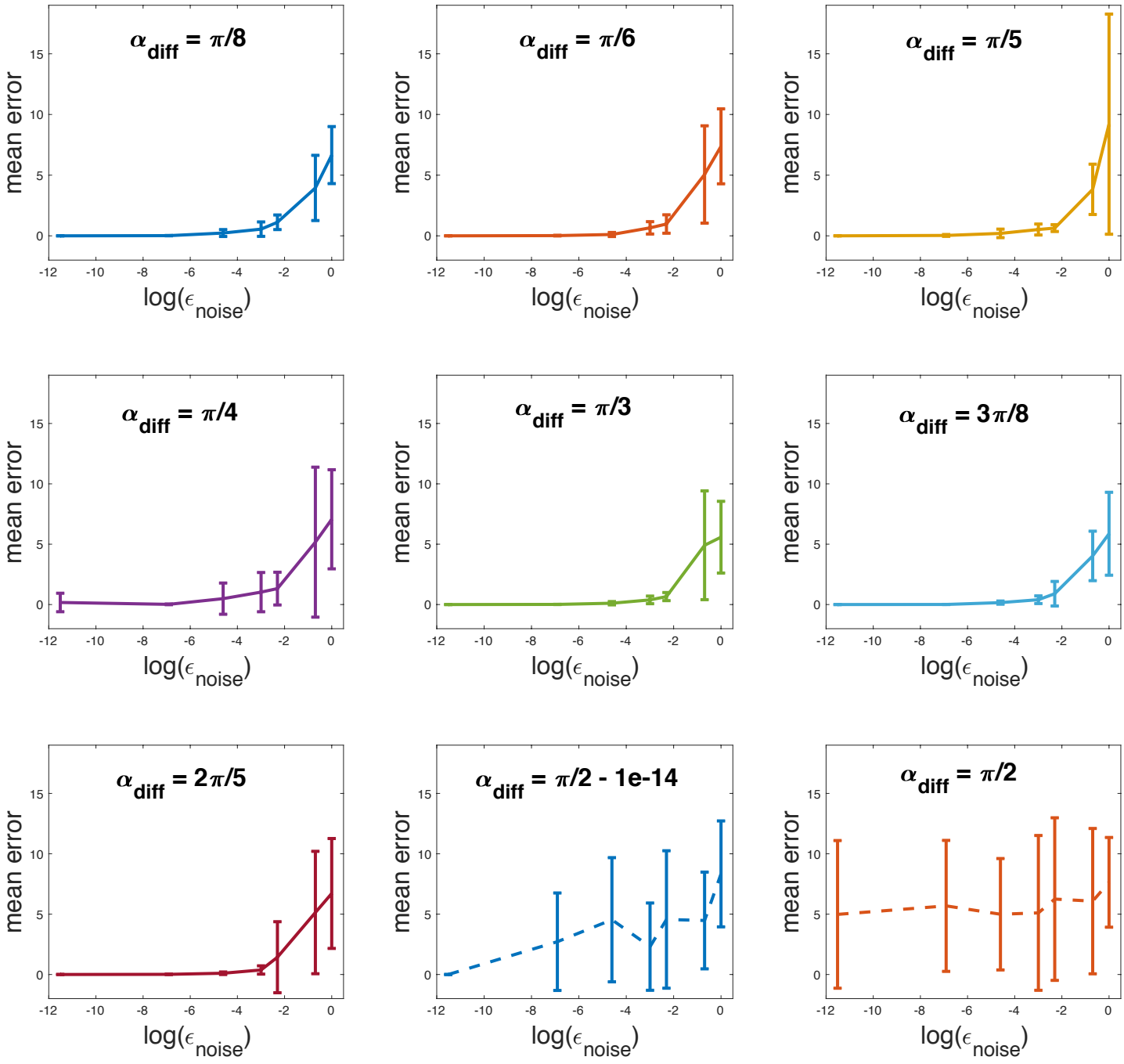


Figure 15: Mean from 20 trials of the Algebraic method with standard deviation bars, for different values of α_{diff}

5 BLP method

When expanding the image reconstruction problem from two projections of one rectangle to several projections of several rectangles, the computational complexity of solving this problem algebraically increases dramatically and therefore an alternative, more efficient method is necessary. In this section, we transform the problem into a binary linear programming problem: given the projection data $P_{\mathbf{R}}(s, \alpha_l)$ for $l = 1, ...p$ and $s = 1, ...q$, find the optimal configuration of rectangles which best fit this projection data. The method for the formulation of this problem is inspired by Jiang and Xiao who fit cuboids to a RGBD image in [12] and is outlined below.

Algorithm 10 *BLP method for computing an image consisting of rectangles from several projections*

I Construct a set of candidate rectangles $\mathbf{R} = \{R_i\}_{i=1}^{n}$ from the back projection image using Alg.11 described below. Candidate rectangle i is denoted as $R_i = [x_{\min}, y_{\min}, x_{\max}, y_{\max}, \phi]$.

II Let binary variable x_i be defined as follows:

$$x_i = \begin{cases} 1 & \text{if candidate rectangle } i \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$

III The binary linear programming problem is:

$$\begin{aligned} &\min_{\mathbf{x}} \{\mathbf{C}(\mathbf{x}) - \nu\mathbf{A}(\mathbf{x}) + \mu\mathbf{N}(\mathbf{x}) + \lambda\mathbf{P}(\mathbf{x}) - \gamma\mathbf{B}(\mathbf{x})\} \\ &\text{s.t. rectangle configuration } \mathbf{x} \text{ satisfies given constraints} \end{aligned} \tag{17}$$

The $\mathbf{C}(\mathbf{x})$ term quantifies the matching costs of the candidate rectangles to all the measured projections, $\mathbf{A}(\mathbf{x})$ reflects the area covered by the selected rectangles, $\mathbf{N}(\mathbf{x})$ is the number of selected candidates rectangles, $\mathbf{P}(\mathbf{x})$ is the pairwise term quantifying the intersection between pairs of candidate rectangles, and $\mathbf{B}(\mathbf{x})$ quantifies the matching of the rectangles to the back projection image. The parameter values $\nu, \mu, \lambda, \gamma \in \mathbb{R}_{>0}$ adjust the weight of each term in the optimization problem.

IV Minimize the BLP in Eq.17 using the simplex method in combination with branch and bound (outlined in Sect.5.4). Thus we find a solution to the image reconstruction problem by using linear programming to select an optimal subset of the set of candidate rectangles $\mathbf{R}$.

In Sect.5.1 we elaborate on the back projection image, and in Sect.5.2 we describe the algorithm used to construct candidate rectangles. In Sect.5.3 we derive the separate terms of the Binary Linear Programming problem of Eq.17. Lastly, in Sect.5.5 we simulate the BLP method by implementing a simplified example, finding an approximation of an image consisting of three rectangles from five projections.

5.1 The back projection image

As in [12], first candidate rectangles are needed. These candidate rectangles are constructed from the back projection image.

As mentioned in Sect.2.2, a back projection image is essentially all the projections "smeared" back over the image domain. Given p projections, the angles of projection α are evenly spaced between 0 and π radians. We discretize the image domain into a grid, and for each point (x_0, y_0) on the grid, we calculate the sum of the intensity or height of each projection from that point as in Eq.7. See Appx.A.12 for this function.

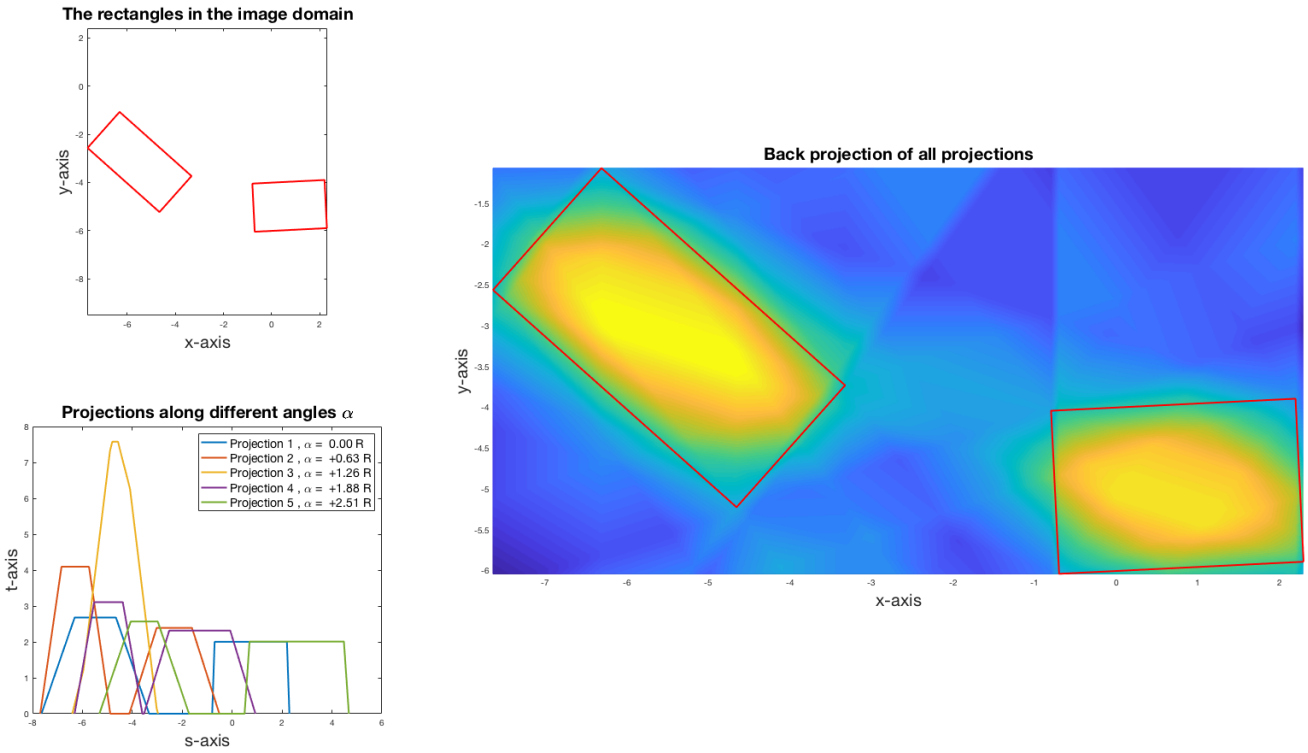


Figure 16: Back projection image from 5 projections of 2 rectangles, with grid size 0.01

Fig.16 shows an example of an image consisting of random rectangles in the top left corner, the Radon Transform of that image along five different angles α in the bottom left corner, and the back projection image on the right. For this figure, the image domain was discretized into a grid with interval width 0.01. Decreasing the interval width to 0.001 produces an even more detailed back projection image, shown in Fig.17: one can see more clearly the rectangular nature of the original image, but at the cost of 100 times more computation time. Likewise, as would be expected, doubling the number of projections from 5 to 10 also improves the quality of the back projection image, shown in Fig.18.

5.2 Constructing candidate rectangles

The candidate rectangles can now be constructed from the back projection image as follows.

Algorithm 11 *Constructing candidate rectangles from the back projection image*

I Find **superpixels**. These are groups of connected pixels with a similar intensity, so $\mathcal{B}(f)(x, y)$ value. Image segmentation tools can be used for this, such as the graph method in [7], but this is beyond the scope of this paper.

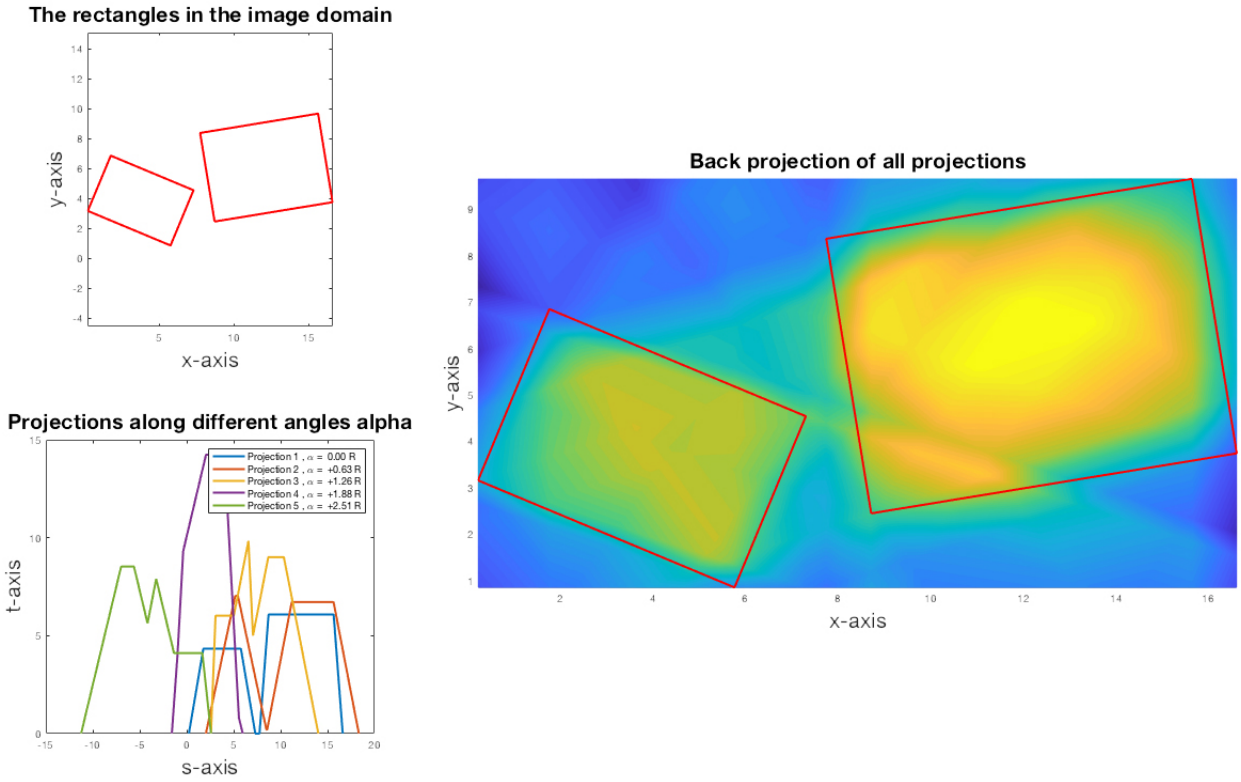


Figure 17: Back projection image from 5 projections of 2 rectangles, with grid size 0.001

II For decreasing threshold values ξ, find **planar patches**. A planar patch is defined as all connected superpixels with an intensity $\mathcal{B}(f)(x, y) > \xi$. So starting with a high threshold value ξ and decreasing this value at regular intervals (down to a certain minimum intensity ξ_0), planar patches of increasing size are formed, each enveloping the previous patch. The resulting effect resembles contour lines on a hill, as in Fig.19

III For each planar patch, construct a candidate rectangle around it: the candidate rectangle is the smallest rectangle that fully encloses that planar patch

5.2.1 Hard constraints

Given this set $\{R_i\}_{i=1}^n$ of candidate rectangles, those that do not comply with the hard constraints are eliminated. Namely:

a Only construct planar patches with an intensity greater than a certain threshold value: $\mathcal{B}(f)(x, y) > \xi_0$, thus ensuring that the candidate rectangles only enclose planar patches with intensities above that threshold.

b Let the **bounding box** be the polygon bounded by the outer edges of the projections. All candidate rectangles must lie for at least 80% of their area within this bounding box.

c For each candidate rectangle, their projection along the different projection angles is calculated. At least 80% of this projection must fall within each measured projection.

These hard constraints ensure that we have a viable set of candidate rectangles $\{R_i\}_{i=1}^n$.

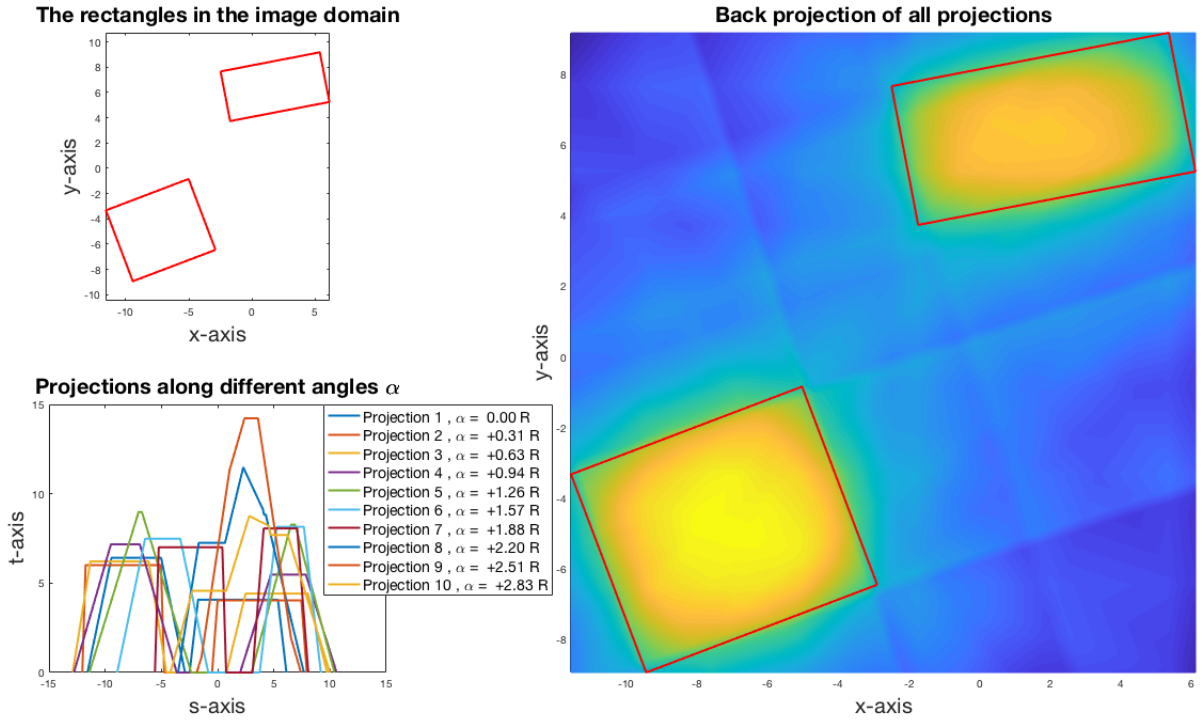


Figure 18: Back projection image from 10 projections of 2 rectangles, with grid size 0.01

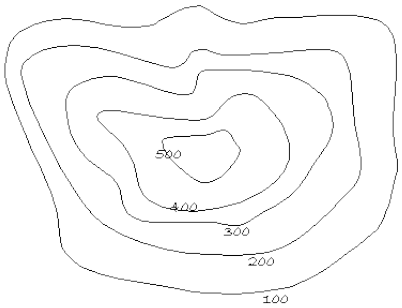


Figure 19: Planar patches on the back projection image resemble contour lines on a hill [25]

5.3 Formulating the BLP

Given this set of candidate rectangles $\{R_i\}_{i=1}^n$, selection criteria are now formulated which must either be minimized or maximized in order to find the candidate rectangles which best fit the given projection data. The separate terms which form the BLP in Eq.17 are explained.

5.3.1 Minimize the local matching cost

For each candidate rectangle $i \in \{1, ...n\}$ and for each projection $l \in \{1, ...p\}$, we calculate the local matching cost. We define the **local matching cost** $c_{i,l}$ as the area of candidate rectangle i's projection along angle α_l which does not overlap with projection $f^{(\alpha_l)}$.

The local matching cost was already subject to hard constraint [c] in the previous section so at least 80% of the rectangle's projection will fall within the projection $f^{(\alpha_l)}$. However, the more a candidate rectangle's projection "sticks out" above the projection data, the worse the fit of this candidate to the data. Fig.20 shows some of the fit of a candidate rectangle's projection with a certain projection l. The total of these matching costs $\mathbf{C}(\mathbf{x})$ is thus the soft constraint

term in the objective function and needs to be minimized.

The total matching cost term is thus:

$$\text{Minimize } \mathbf{C}(\mathbf{x}) = \sum_{i=1}^{n} \sum_{l=1}^{p} c_{i,l} x_i \tag{18}$$

Simply minimizing the term $\mathbf{C}(\mathbf{x})$ would yield the trivial all zero solution, so we need more terms in the objective function.

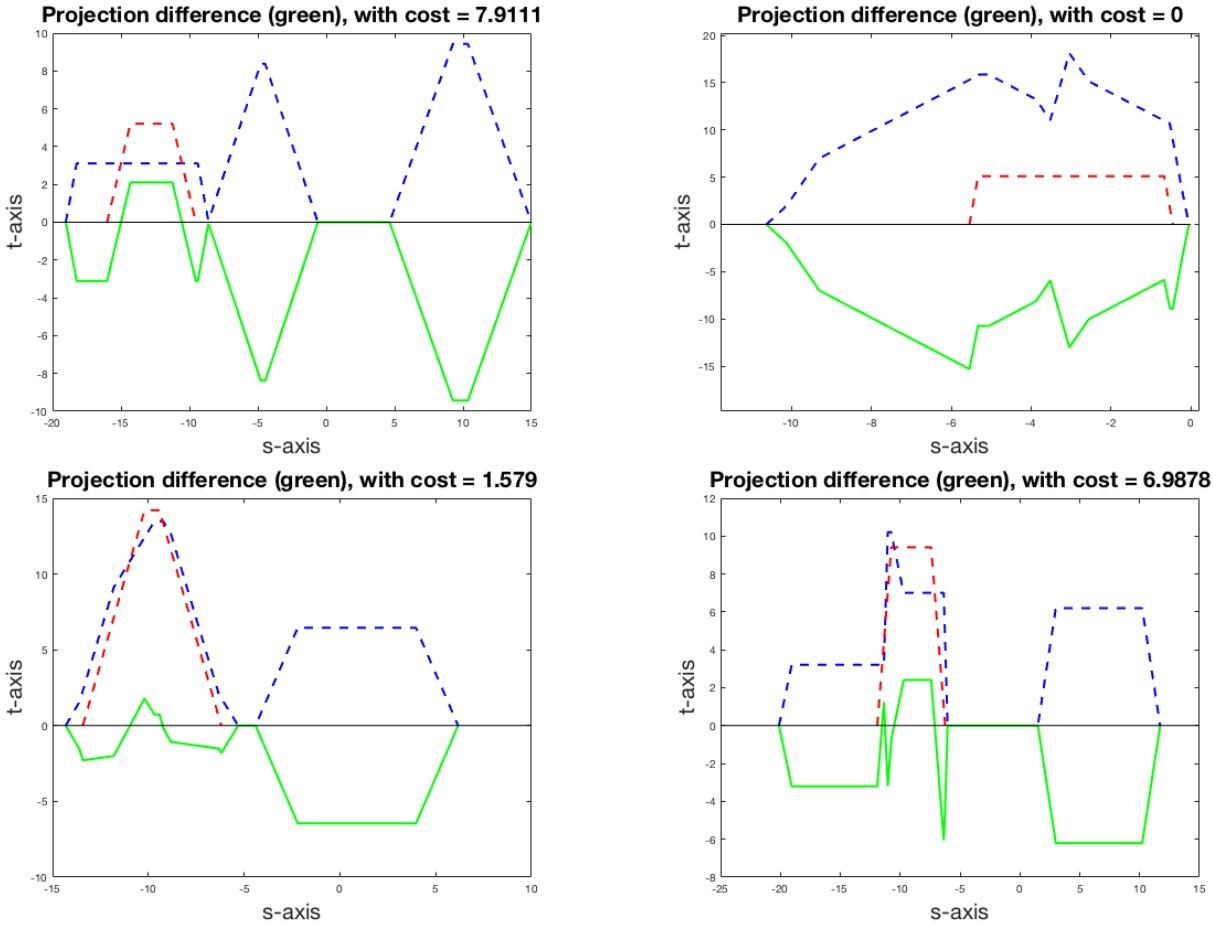


Figure 20: A candidate rectangle's projection (in red), original projection l (in blue), and the projection difference (in green). The local matching cost is the area of the green curve above the s-axis.

5.3.2 Maximize the area of the rectangles

When minimizing the local matching cost, selecting a small rectangle which lies within the original rectangle (for example which only encloses the brightest yellow patch in the previous back projection figures) would be preferred to selecting a larger (but actually better fitting) rectangle. Thus we add a soft constraint term which prefers larger rectangles to smaller rectangles.

We define a_i as the area of candidate rectangle i. Then the total area term in the objective function is:

$$\text{Maximize } \mathbf{A}(\mathbf{x}) = \sum_{i=1}^{n} a_i x_i \tag{19}$$

5.3.3 Minimize the number of rectangles

To avoid getting unnecessarily large, complicated solutions with many small rectangles covering a large area, we also have a soft constraint term which minimizes the number of rectangles. This in combination with maximizing the area of the rectangles ensures that finding a small number of large rectangles which fit the projection data is favoured to finding many small rectangles.

However, given a projection $f^{(\alpha_l)} = [S^l; F^l]$ we can deduce the minimum number of rectangles which produced this projection by looking at the number of breakpoints of the projection, which is the cardinality of set S^l: $|S^l|$. To wit, one rectangle produces at most four breakpoints since its breakpoints could overlap with that of another rectangle. Therefore we define n_l as the minimum number of rectangles which formed projection l:

$$n_l = \left\lceil \frac{|S^l|}{4} \right\rceil$$

Now we define the minimum number of rectangles needed to form all projections as:

$$n_{\min} = \max_{l=1,...p} n_l \tag{20}$$

Thus we add the soft constraint number term to the objective function:

$$\begin{aligned} &\text{Minimize } \mathbf{N}(\mathbf{x}) = \sum_{i=1}^{n} x_i \\ &\text{subject to } \sum_{i=1}^{n} x_i \geq n_{\min} \end{aligned} \tag{21}$$

This constraint ensures that a non-trivial solution to the BLP is always found.

5.3.4 Minimize the intersection between pairs of rectangles

As we have assumed that the rectangles in the original image do not overlap, we wish to minimize the overlap between the selected candidate rectangles. Let $e_{i,j}$ be the **intersection ratio** between candidate rectangles i and j defined as:

$$e_{i,j} = \frac{\text{intersection area}}{\text{area of smallest rectangle}} \quad \forall i, j \in \{1, ..., n\} \text{ with } i \neq j$$

We compute this intersection ratio for all pairs of candidate rectangles $i \neq j$. The soft constraint in the objective function is thus the intersection ratio term:

$$\text{Minimize } \mathbf{P}(\mathbf{x}) = \sum_{\{\{i,j\}:0<e_{i,j}<t\}} e_{i,j}\, x_i\, x_j$$

We also want the hard constraint that the intersection ratio never exceeds a certain threshold value t. This is ensured by adding the constraint $x_i + x_j \leq 1 \;\; \forall$ pairs with $e_{i,j} \geq t$. The term $e_{i,j} x_i x_j$ then becomes zero for those sets of candidate rectangles.

However, $\mathbf{P}$ in the current form is quadratic, so in order to linearize it we introduce the auxiliary variable $w_{i,j}$ and let $w_{i,j} \leq x_i$, $w_{i,j} \leq x_j$, $w_{i,j} \geq 0$ and $w_{i,j} \geq x_i + x_j - 1$. By thus defining $w_{i,j}$, $w_{i,j} = x_i \cdot x_j$ holds. The proof is left up to the reader as one simply has to verify that for all cases $((x_i = 0 \wedge x_j = 0), (x_i = 1 \wedge x_j = 0), (x_i = 1 \wedge x_j = 1))$ we get $w_{i,j} = x_i \cdot x_j$. (The case $(x_i = 0 \wedge x_j = 1)$ is excluded due to the symmetry of the problem.)

Thus the complete intersection term containing the soft and hard constraints is as follows:

$$\begin{aligned} \text{Minimize} \quad & \mathbf{P}(\mathbf{x}) = \sum_{\{i,j\}} e_{i,j}\, w_{i,j} \\ \text{subject to} \quad & x_i + x_j \leq 1 \quad \forall \{i,j\} : e_{i,j} \geq t \\ & w_{i,j} \leq x_i, \;\; w_{i,j} \leq x_j, \;\; w_{i,j} \geq x_i + x_j - 1 \quad \forall \{i,j\} : 0 < e_{i,j} < t \end{aligned} \tag{22}$$

5.3.5 Maximize matching to the back projection image

We have constructed the candidate rectangles around areas or planar patches of higher intensity pixels. However, the higher the intensity $\mathcal{B}(f)(x_0, y_0)$ at point (x_0, y_0) in the back projection image, the bigger the probability that there is a rectangle which covers that point. Recall that a superpixel was a group of connected pixels with almost the same intensity. Let g_k be the average intensity of superpixel k and let s_k be the area on the back projection image covered by superpixel k. We define auxiliary variable v_k as:

$$v_k = \begin{cases} 1 & \text{if superpixel } k \text{ is covered by at least one selected rectangle} \\ 0 & \text{otherwise} \end{cases}$$

Thus the total match to the back projection image is defined as:

$$\mathbf{B}(\mathbf{x}) = \sum_{\text{superpixel } k} s_k\, g_k\, v_k$$

We implement the definition of v_k by adding the constraints $v_k \leq \sum_{\{\text{cand.rect. } i \text{ covers superpixel } k\}} x_i$ and $v_k \leq 1 \;\; \forall$ superpixels k. This constraint forces $v_k = 0$ if no selected candidate rectangle covers superpixel k, but if superpixel k is covered by one (or more) selected candidate rectangle, $v_k = 1$ as desired, so $\mathbf{B}$ is maximized.

We maximize the matching to the back projection image by the soft constraint term in the objective function:

$$\begin{aligned} \text{Maximize} \quad & \mathbf{B}(\mathbf{x}) = \sum_{k} s_k\, g_k\, v_k \\ \text{subject to} \quad & v_k \leq \sum_{\text{rect. } i \text{ covers superpixel } k} x_i, \;\; v_k \leq 1, \quad \forall \text{ superpixel } k \end{aligned} \tag{23}$$

Maximizing $\mathbf{B}$ thus defined favors selection of candidate rectangles which cover large superpixels with high intensities over small superpixels with low intensities.

5.3.6 The entire BLP

Now that we have derived all the soft and hard constraints, we combine all the terms in Eq.18, Eq.19, Eq.21, Eq.22 and Eq.23. These form the objective function and their corresponding

constraints for the BLP in Eq.17. The overal problem thus becomes:

$$\min_{\mathbf{x}} \left\{ \sum_{i=1}^{n} \sum_{l=1}^{p} c_{i,l}\, x_i - \nu \sum_{i=1}^{n} a_i\, x_i + \mu \sum_{i=1}^{n} x_i + \lambda \sum_{\{i,j\}} e_{i,j}\, w_{i,j} - \gamma \sum_{k} s_k\, g_k\, v_k \right\}$$

$$\begin{aligned} \text{s.t.} \quad & \sum_{i=1}^{n} x_i \geq n_{\min} \\ & w_{i,j} \leq x_i, \;\; w_{i,j} \leq x_j, \;\; w_{i,j} \geq x_i + x_j - 1 \quad \forall \{i,j\} : 0 < e_{i,j} < t \\ & x_i + x_j \leq 1 \;\; \forall \{i,j\} : e_{i,j} \geq t \\ & v_k \leq \sum_{\text{rect. } i \text{ covers superpixel } k} x_i, \;\; v_k \leq 1, \quad \forall \text{ superpixel } k \\ & x_i = 0 \text{ or } 1. \text{ All variables are non-negative.} \end{aligned} \tag{24}$$

We see that it is indeed a Binary Linear programming problem because the objective function and constraints are a linear function of the binary decision variables $\mathbf{x}$.

5.4 Solving the BLP

Given this BLP, one method to solve it is using the **Implicit Enumeration Method**. As all decision variables can either be 0 or 1, the implicit enumeration method systematically eliminates obviously infeasible solutions, and then evaluates all the remaining solutions to find the optimum [23, p.C-11]. This is a useful method for small BLP problems. However, in practice the number of binary variables n (so number of candidate rectangles) is usually quite large, and then the exhaustive approach of evaluating all 2^n combinations quickly becomes very inefficient.

A better way to solve the formulated BLP is using the **Branch and Bound Method** in conjunction with the **Simplex Method**. First, some definitions are reviewed. Then we outline the basic concept of the algorithm (in Sect.5.4.2), and finally look at its complexity (Sect.5.4.3).

5.4.1 Some definitions

The BLP can be written into the following canonical form [21, p.304]:

$$\begin{aligned} \text{Minimize} \quad & \mathbf{cx} \\ \text{subject to} \quad & \mathbf{Ax} \leq \mathbf{b} \\ & 0 \leq \mathbf{x} \leq 1 \\ & \mathbf{x} \text{ integer} \end{aligned} \tag{25}$$

Where $\mathbf{cx}$ is the linear objective function, with $\mathbf{c}$ being the $n \times 1$ cost coefficient vector and $\mathbf{x}$ the n binary decision variables denoting the rectangle configuration. The linear constraints are expressed in $\mathbf{Ax} \leq \mathbf{b}$, with $\mathbf{A}$ a $m \times n$ matrix and $\mathbf{b}$ a $m \times 1$ vector. Note that any inequality constraint $i \in \{1, ..., m\}$ can simply be converted to an equation by adding the slack variable x_{n+i} [21, p.4]:

$$\sum_{j=1}^{n} a_{i,j} x_j \leq b_i \;\; \text{is equivalent to} \;\; \sum_{j=1}^{n} a_{i,j} x_j + x_{n+i} = b_i, x_{n+i} \geq 0$$

Also note that any maximization problem can be transformed into a minimization problem [21, p.5] as:

$$\max_{\mathbf{x}} \sum_{i=1}^{n} c_i x_i = -\min_{\mathbf{x}} \sum_{i=1}^{n} -c_i x_i$$

This is how for example the rectangle area term in Eq.19 which needed to be maximized was added to the objective function which is minimized.

We define a **feasible solution** as a set of values $x_1, x_2, ...x_n$ which satisfy all given constraints. We define the **feasible region X** as the set of all feasible points:

$$\mathbf{X} = \{\mathbf{x} \in \mathbb{R}^n | \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \in \{0, 1\}$$

Thus defined, $\mathbf{X}$ is a **convex** region, meaning that for all points $\mathbf{y_1}, \mathbf{y_2} \in \mathbf{X}$, $\mathbf{y} = \zeta\mathbf{y_1} + (1 - \zeta)\mathbf{y_2} \in \mathbf{X}$ also holds, for all $\zeta \in \mathbb{R}$ with $0 \leq \zeta \leq 1$ [10].

We define $\mathbf{y} \in \mathbf{X}$ as an **extreme point** of $\mathbf{X}$ if "it cannot be represented as a strict convex combination of two distinct points in $\mathbf{X}$" [21, p.65]. So $\mathbf{y} = \zeta\mathbf{y_1} + (1 - \zeta)\mathbf{y_2}$ with $\zeta \in \mathbb{R}, 0 \leq \zeta \leq 1$, implies $\mathbf{y} = \mathbf{y_1} = \mathbf{y_2}$.

A **basic solution x** has at most m non-zero values and can be rearranged such that $\mathbf{x} = \begin{bmatrix} \mathbf{x}_B \\ \mathbf{x}_N \end{bmatrix}$ with $\mathbf{x}_B$ the part of the solution corresponding with the basic variables, and $\mathbf{x}_N = \mathbf{0}$ [21, p.95]. Finally, we can now define a **basic feasible solution** (BFS) as a basic solution with $\mathbf{x}_B \geq 0$; this is an extreme point of the convex feasible region $\mathbf{X}$ [21, p.99].

5.4.2 Simplex Method with Branch and Bound

A general outline for the method to solve the BLP now follows. First, an optimal solution of the Linear Programming-Relaxation is found using the **Simplex Method**:

Algorithm 12 *Outline of the Simplex Method*

I Remove the integrality constraints on all the decision variables

II Find an initial Basic Feasible Solution with the **Two-Phase Method**:

- i Choose basic variables $\mathbf{x_B}$
- ii Set all the other nonbasic variables $\mathbf{x_N}$ on either their lower or upper bound
- iii Determine the value of the basic variables based on those set non-basic variables
- iv Where necessary, add artificial variables
- v Solve **Phase 1:** minimize the sum of the artificial variables
- vi When a BFS is found without artificial variables in the solution, go to phase 2. Otherwise, no feasible solution exists.

III Solve **Phase 2:** minimize the objective function $\mathbf{z} = \mathbf{cx}$:

- i Determine whether the found BFS is optimal. If so, stop. Otherwise, continue to the next step
- ii Move to a BFS at an adjacent vertex with a value $\mathbf{z}$ less than or equal to that of the previous BFS, by walking around the edge of the feasible region $\mathbf{X}$. This is done by removing one basic variable from the basis and replacing it by a nonbasic variable by pivoting in the tableau. Go back to the previous step.

The found relaxed solution of the problem (meaning with no integer restrictions) will be the lower bound (abbreviated LB) of the first node in the Branch and Bound Method. Next, all non-integer decision variables x_i of this solution are rounded up. In other words, all decision variables which are not exactly zero are set to one, and this forms the upper bound (abbreviated UB) of the first node. We know that the optimal binary solution will always be greater than or equal to the found relaxed optimal solution (the LB), and less than or equal to the rounded up binary solution (the UB) [23, p.C-3].

The general concept of the Branch and Bound method is now to partition the feasible region into smaller, more manageable subregions, and if necessary, partition those subregions again. A tree diagram with nodes and branches is used to organize this partitioning.

Algorithm 13 *Outline of the Branch and Bound Method [23]*

I As mentioned, the LB of the first node is the found relaxed solution, and the UB is the rounded up binary solution

II We branch from the decision variable x_i with the lowest fractional part, meaning the index i with lowest value $x_i > 0$. Naturally, the two branches are $x_i = 0$ and $x_i = 1$; these constraints thus partition the feasible region in two.

III We re-calculate the upper and lower bounds by once again solving the linear programming relaxation with the Simplex Method, with added constraints $x_i = 0$ or 1 for each branch. These form the lower bounds of the new nodes, and their rounded up binary solutions once again form the upper bounds. We keep track of the best binary solution $\mathbf{z}$ found so far at any node, which is the minimum UB of all nodes.

IV We stop branching from a node if either:

- The found relaxed solution is binary, so UB = LB
- Or we have an infeasible solution, so this node does not have a basic feasible solution
- Or the LB $\geq \mathbf{z}$, so we cannot find a better solution in that branch than we have already found

V The optimal binary solution is the UB solution of the node with the smallest LB value of any ending node.

VI Repeat steps II - V, continuing to branch from the node with the minimum lower bound, until the optimal binary solution is found

With this method an optimal solution to the BLP is found. A finite set of candidate rectangles is always constructed and due to the constraint $\sum_{i=1}^{n} x_i \geq n_{\min}$ a solution to the BLP is always found.

5.4.3 Complexity of the algorithm

When looking at the worst-case computational performance, the simplex algorithm in combination with branch and bound is exponential in problem size (so the number of candidate rectangles n). However, in practice it is actually an extremely efficient way to solve linear problems [21, p.393]. Empirically, the simplex algorithm takes on average about m to at most $3m$ iterations [21, p.206], where m is the total number of constraints, so rows of matrix $\mathbf{A}$.

There do exist some algorithms which are theoretically more efficient as they are polynomial-time (in terms of problem size n) [21, p.393]. For example Khachian's Ellipsoid Algorithm has

a computational complexity of $O[(m+n)^6 L]$, where L is the number of binary bits necessary for all the data of the problem, which does indeed make it a polynomial-time algorithm, though of a higher order. Furthermore, in practice, its performance is very close to the worst-case bound, whereas in the simplex method this is not the case [21, p.402].

A more feasible alternative is Karmarkar's Projective Algorithm, which has a polynomial lower bound for the worst-case computation time of $O(n^{3.5}L)$ [21, p.422]. Where the Simplex algorithm walks along the edges of the polytopal feasible region $\mathbf{X}$, going from BFS to BFS [21, p.397], Karmarkar's algorithm traverses a trajectory through the interior of the feasible region, seeking a pathway toward optimality [21, p.428]. This method is still a long way from being fully developed, but the underlying concept is very promising [21, p.393], especially for large sparse systems [22].

5.5 Simplified implementation of the BLP

We will now demonstrate a simplified example of how the BLP method in Algorithm 10 can be implemented to numerically solve the inverse problem. Implementing the entire problem into Matlab is beyond the scope of this paper. This is because first of all, the image segmentation tools to construct the superpixels and planar patches are quite complex in implementation. And second, deriving algorithms to compute all the terms of the BLP is also very time-consuming. Therefore it was decided to drop the last two terms of the objective function in the following example.

Say we have three random rectangles we wish to find, and five projections of these rectangles forming the projection data. See Fig.21 for the rectangles and projections in this example.

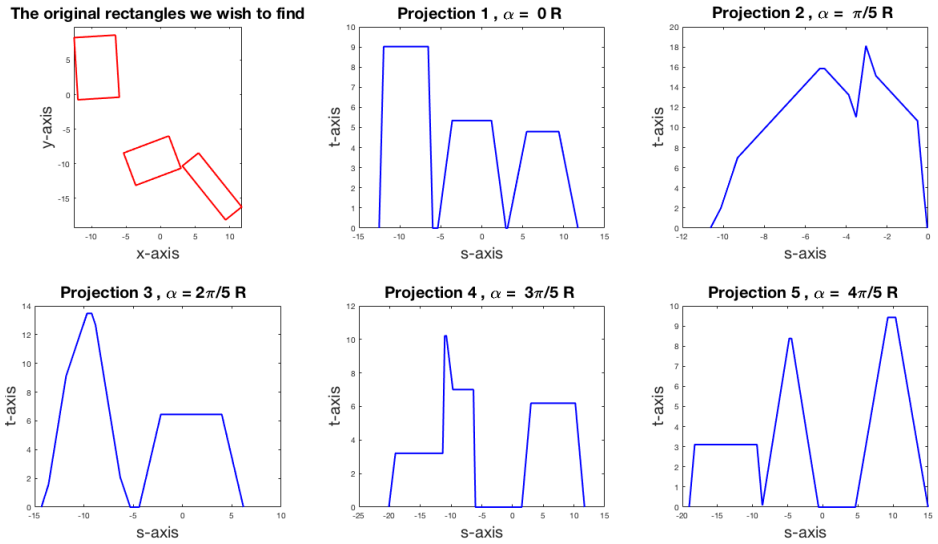


Figure 21: Three random rectangles measured from five projection angles

The back projection image is computed as in Sect.5.1. Normally image segmentation tools would now be used to make the superpixels, then the planar patches and the candidate rectangles would be constructed around these planar patches. However, for this simplified example, the candidate rectangles are drawn on the back projection image by sight (see Appx.A.13). Fig.22 shows the back projection image overlaid by a set of 12 candidate rectangles.

Given this set of candidate rectangles $\mathbf{R}$, we now compute the local matching cost $c_{i,l}$ for each candidate rectangle $i \in \{1, ...n\}$ with each projection $l \in \{1, ...p\}$. This is done by calculating the projection $P_{R_i}(s, \alpha_l)$ of candidate rectangle i along angle α_l, and computing the projection difference function between $P_{R_i}(s, \alpha_l)$ and projection l, which is also a piecewise linear function.

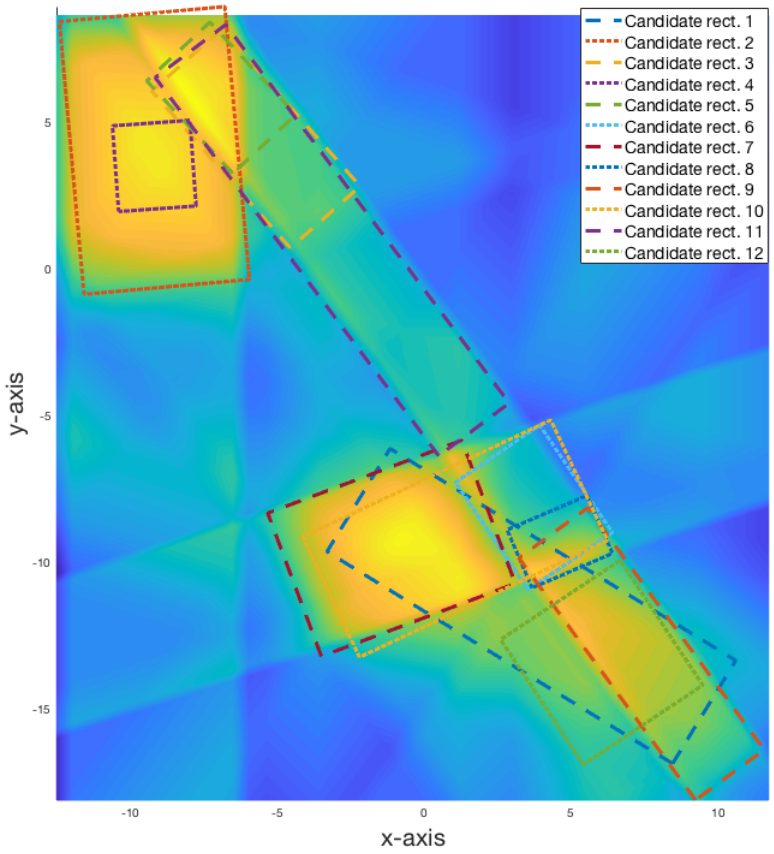


Figure 22: The back projection image of the five projections, overlaid by the set of candidate rectangles $\mathbf{R}$

The local matching cost is the sum of the areas of this function above the $s-$axis. Appx.A.14 contains the function which calculates the local matching cost for a given candidate rectangle and projection, some examples of which were shown in Fig.20.

Given this $n \times p$ matrix of matching costs, the total matching cost of each candidate rectangle is calculated by summing up all the matching costs of rectangle i for each projection, so summing over each row of the matching cost matrix. This produces a $n \times 1$ vector $\mathbf{c}_{\text{total}}$. Multiplying the transposed of this vector with our rectangle configuration vector $\mathbf{x}$ yields the matching cost term: $\mathbf{C}(\mathbf{x}) = \mathbf{c}_{\text{total}}^T \mathbf{x}$.

The area of each candidate rectangle i is simply calculated by multiplying the base by the height: $a_i = (x_{\max} - x_{\min})(y_{\max} - y_{\min})$. Then the area term of the objective function is: $\mathbf{A}(\mathbf{x}) = \mathbf{a}^T \mathbf{x}$.

Finally, the number of rectangles is calculated by multiplying a $1 \times n$ vector of ones denoted $\mathbf{n}$ by $\mathbf{x}$: $\mathbf{N}(\mathbf{x}) = \mathbf{n}^T \mathbf{x}$. The minimum number of rectangles $n_{\min}$ needed to produce these projections defined as in Eq.20 is clearly 3 based on the projection functions in Fig.21.

Combining these three terms yields the simplified optimization problem:

$$\begin{aligned} &\min_{\mathbf{x}} (\mathbf{c}_{\text{total}} - \nu \mathbf{c} + \mu \mathbf{n})^T \mathbf{x} \\ &\text{s.t. } \mathbf{n}^T \mathbf{x} \geq n_{\min} \\ &\quad x_i = 0 \text{ or } 1 \end{aligned} \tag{26}$$

These equations are then implemented into Matlab's linear programming solver `intlinprog`, see Appx.A.15.

When implementing and testing the entire BLP model, finding the best suitable parameter values for ν, μ, λ and γ could be approached as a machine learning problem. However, in this

simplified example, setting $\mu = 5$ and $\nu = 0.5$ yielded the desired result: $x_2 = 1, x_7 = 1, x_9 = 1$ and $x_i = 0$ for all other indexes of the candidate rectangles. Comparing the candidate rectangles $2, 7$ and 9 with the original rectangles in Fig.23, one sees that indeed the candidates which closest resemble the original rectangles were selected. Of course, this was a very biased simulation of the process as the original rectangles were known when drawing the candidates. But this example simply illustrates the process of how the BLP method can be used to approximate the original rectangle composition.

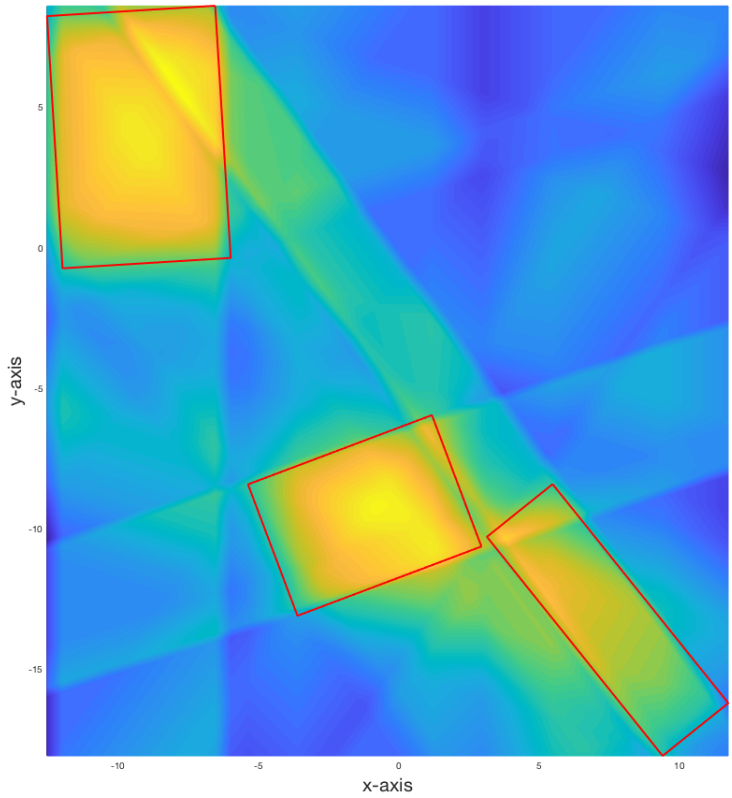


Figure 23: The original rectangles on the back projection image

Matlab actually has built-in `radon` and `iradon` functions which compute the Radon Transform and Inverse Radon Transform respectively, thus which should obtain similar results as our numerical approach. However, the `radon` and `iradon` computations are black boxes which determines the back projection by approximation, as shown in Fig.24. Comparing the projection functions in the bottom left corner to the exact projections shown in Fig.21, one sees a great differences in accuracy of the Radon Transform (projection functions). This difference is further illustrated when comparing the back projection image in Fig.23 computed from the exact Radon Transform functions with the approximation in Fig.24.

Though they are black boxes, the complexity of the `radon` and `iradon` computations probably depend on the number of pixels which in the given example might be small, but in practice, real CT scans are usually about 512 by 512 pixels in size [18], so this method quickly becomes very inefficient. As explained above, the computational complexity of the method described here depends on the number of rectangle candidates and constraints, which is always much less than the number of pixels. Thus in practice this algorithm would be much more efficient.

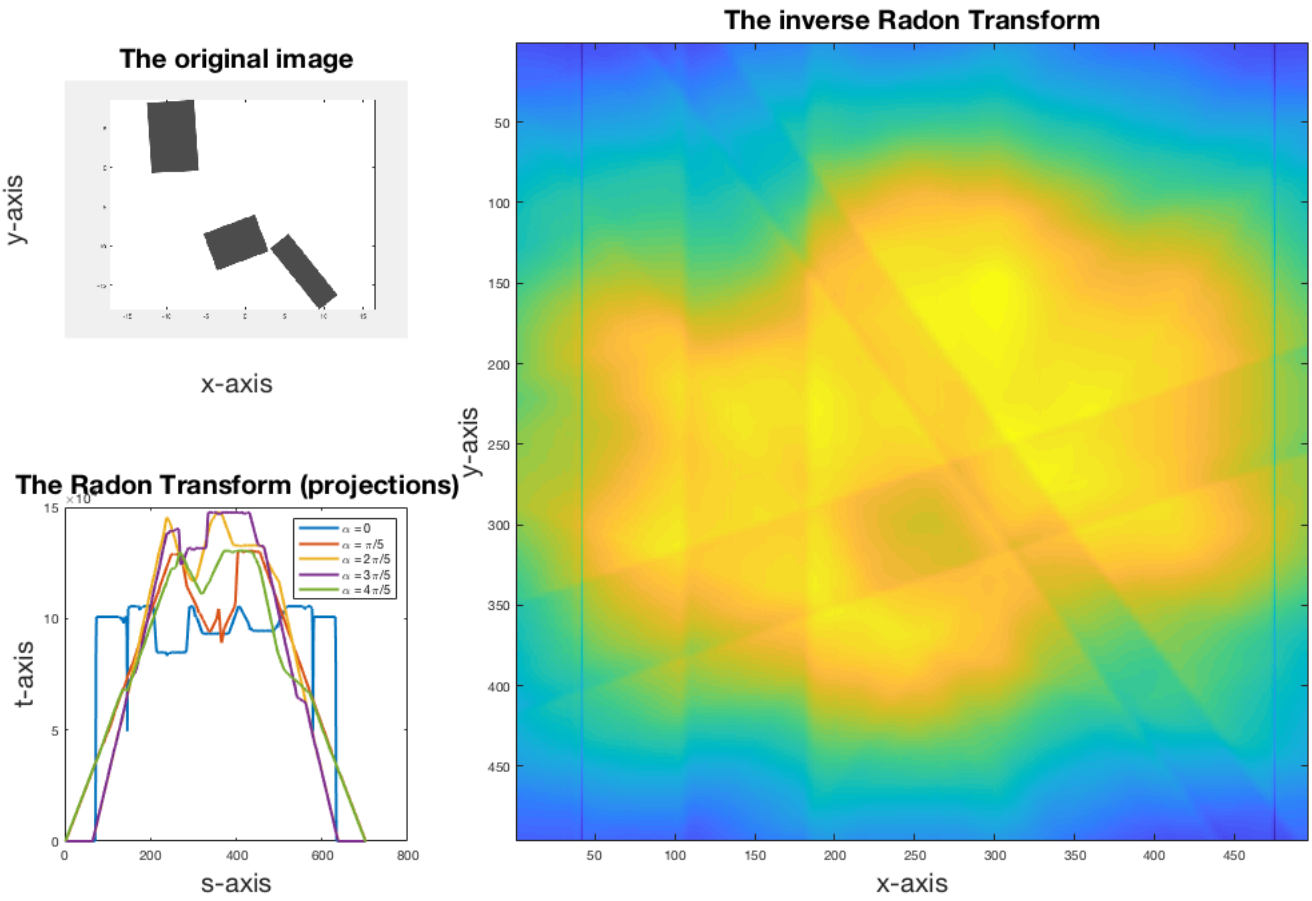


Figure 24: The Radon Transform (projection functions) of the rectangles and the Inverse Radon Transform

6 Conclusion

The concepts and mathematics of the existing algorithms used to reconstruct an image from CT scan data were examined, including the Radon Transform, which is critical to understand the process of computed tomography. When restricting the image reconstruction problem with the assumption of the rectangular nature of the image, a new method to algebraically compute the exact Radon Transform of an image was developed.

Furthermore, an algebraic method to compute the original rectangle (the image), from one or two projections was developed. For one projection of the rectangle, it was found that as long as the ratio $r = \frac{h}{s_2 - s_1}$ is greater or equal to two, a solution to this inverse problem exists. However, for just one projection, it will never be unique. In fact, we will always have infinitely many solutions.

The algebraic method which computes a rectangle from two projections, does yield a unique solution, as long the two projections are not orthogonal or almost equal. When adding noise to the projection data, the solutions were reasonably stable, regardless of the angle of difference α_{diff} between the two projection angles. The error in the solution increases linearly with the level of noise added to the projection data, as would be expected. Only when α_{diff} is close to 0 or $\frac{\pi}{2}$, so when the projections are either almost parallel or almost orthogonal, then the solution from the algebraic method is very unstable.

When expanding the inverse problem by increasing the number of rectangles in the image and increasing the number of projections, the algebraic method becomes computationally inefficient. Instead, the BLP method was developed, where the image reconstruction problem was formulated as a Binary Linear Programming problem (BLP). A method for constructing a set of candidate rectangles from the back projection image was described. Next, the BLP was constructed from different terms for the matching cost of the rectangles to the projections, the areas of the rectangles, the number of selected rectangles, the intersection between pairs of rectangles and the matching of the rectangles to the back projection image. We outlined how the BLP could be solved using Branch and Bound in conjunction with the Simplex Method and explored the algorithm's complexity.

A simplified version of the BLP was implemented to demonstrate how it could be used to approximate the original image. Implementing the entire BLP would require further research, such as the image segmentation tools needed to construct the candidate rectangles from the back projection image. The model would need to be tested on a very large data set, and finding the best suited parameter values in the objective function could be approached as a machine learning problem. Further research could also be done to expand both the algebraic and BLP methods to the 3-dimensional case, working with rectangular cuboids instead of rectangles.

In conclusion, the premise of the image consisting of rectangular shapes enables one to develop new alternative algorithms to the existing methods (such as Filtered Back Projection and algebraic reconstruction techniques) to solve the image reconstruction problem of CT scans. These existing methods require the scan to be performed around the full 180°, and approximate the solution using no prior knowledge of the shape of the image. They are thus less accurate than the new algebraic method which yields a specific rectangle or the BLP method which yields a set of selected rectangles from only a few projections. Furthermore, the complexity of the usual methods depends on the pixel grid size, whereas the complexity of the two methods developed in this paper depend on the number of projections and rectangles. Therefore, these methods are a promising start to approaching the image reconstruction problem from a slightly different angle when performing CT scans of rectangular or cuboidal shapes and future research could further develop these algorithms.

References

[1] Uri M. Ascher and Chen Greif. *A First Course in Numerical Methods*. Society for Industrial and Applied Mathematics, Philadelphia, 2011.

[2] Oren Barkan, Jonathan Weill, Amir Averbuch, and Shai Dekel. Adaptive compressed tomography sensing. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2195–2202, 2013.

[3] Julia F Barrett and Nicholas Keat. Artifacts in CT: Recognition and Avoidance. *RadioGraphics*, 24:1679–1691, 2004.

[4] Ronald Newbold. Bracewell. *Two dimensional imaging*, chapter 8: n-dimensional Fourier Transform. Prentice Hall, 1995.

[5] Stanley Roderick Deans. *The Radon Transform and Some of Its Applications*. Dover Publications, 2007.

[6] Timothy G. Feeman. *The Mathematics of Medical Imaging: a beginners guide*. Springer, Place of publication not identified, 2 edition, 2015.

[7] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2):167–181, Sep 2004.

[8] Yanan Fu, Honglan Xie, Biao Deng, Guohao Du, Rongchang Chen, and Tiqiao Xiao. Three-dimensional structure of polystyrene colloidal crystal by synchrotron radiation x-ray phase-contrast computed tomography. *Appl. Phys. A*, 115:781–790, 06 2014.

[9] Gabor T. Herman. *Fundamentals of Computerized Tomography: Image Reconstruction from Projections*. Springer, New York, NY, 2009.

[10] Han Hoogeveen. Samenvatting college 5. http://www.cs.uu.nl/docs/vakken/opt/, Dec 2018. Lecture notes from the UU course Optimalisering en Complexiteit.

[11] Kirsten M. Ø. Jensen, Xiaohao Yang, Josefa Vidal Laveda, Wolfgang G. Zeier, Kimberly A. See, Marco Di Michiel, Brent C. Melot, Serena A. Corr, and Simon J. L. Billinge. X-ray diffraction computed tomography for structural analysis of electrode materials in batteries. *Journal of The Electrochemical Society*, 162(7):A1310–A1314, 2015.

[12] Hao Jiang and Jianxiong Xiao. A linear approach to matching cuboids in rgbd images. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2171–2178, June 2013.

[13] Toshiyuki Kawamura. Observations of the internal structure of sea ice by x ray computed tomography. *Journal of Geophysical Research: Oceans*, 93(C3):2343–2350.

[14] Lene Hundborg Koss. For the first time scientists can observe the nano structure of food in 3d. https://www.science.ku.dk/english/press/news/2016/for-the-first-time-scientists-can-observe-the-nano-structure-of-food-in-3d/, Mar 2016. [Online; accessed 27 Nov 2018].

[15] Cristina Luiggi. Unveiling animal mummies. https://www.the-scientist.com/news-opinion/unveiling-animal-mummies-42257, Jul 2011. [Online; accessed 27 Nov 2018].

[16] Andrew J. McElrone, Brendan Choat, Dilworth Y. Parkinson, Alastair A. MacDowell, and Craig R. Brodersen. Using High Resolution Computed Tomography to Visualize the Three Dimensional Structure and Function of Plant Vasculature. *Journal of Visualized Experiments*, (74):1–11, 2013.

[17] NU.nl. Stralingsdosis ct-scan kan omlaag zonder kwaliteitsverlies. https://www.nu.nl/gezondheid/4067171/stralingsdosis-ct-scan-kan-omlaag-zonder-kwaliteitsverlies.html, Jun 2015. [Online; accessed 27 Nov 2018].

[18] Oleg S. Pianykh. *Digital imaging and communications in medicine: a practical introduction and survival guide*, chapter 6.3 Working with digital medical images, page 97–102. Springer Science & Business Media, 2 edition, 2009.

[19] Thammanit Pipatsrisawat, Aca Gacic, Franz Franchetti, M Puschel, and Jose Moura. Performance analysis of the filtered backprojection image reconstruction algorithms. volume 5, pages v/153 – v/156, Apr 2005.

[20] Alexander D. Poularikas and Stanley R. Deans. *Transforms and applications handbook*, chapter 8: Radon and Abel Transforms. CRC Press, Boca Raton, FL, 2 edition, 2010.

[21] Hanif D. Sherali, John J. Jarvis, and Mokhtar S. Bazaraa. *Linear Programming And Network Flows*. John Wiley & Sons Incorporated, 4 edition, 2010.

[22] Gilbert Strang. Karmarkar's algorithm and its place in applied mathematics. *The Mathematical Intelligencer*, 9(4):4–10, 1987.

[23] Bernard W Taylor. *Introduction to Management Science*, chapter Module C: Integer Programming: The Branch and Bound Method, pages C2–C11. Pearson, 13 edition, 2018.

[24] Amanda Verdonk. Met minder metingen meer laten zien. *NWO Hypothese*, 2:24–25, 2018.

[25] Jennifer Wadsworth. Swale: significance of "on contour"? https://permies.com/t/36143/swale-significance-contour, 2014.

A Appendix: Matlab code

A.1 Computing the Radon Transform of a rectangle

```
function projection = calculate_projection(xmin, ymin, xmax, ymax, theta)

% projection = calculate_projection(xmin, ymin, xmax, ymax, theta):
%takes rectangle = [xmin, ymin, xmax, ymax]  and rotation angle theta = ...
    phi-alpha,
%and returns parametrized projection = [s1 s2 s3 h]

%Define counter-clockwise rotation matrix T_theta:
T_theta = [
    cos(theta)    -sin(theta);
    sin(theta)    cos(theta)];

% 4 vertices of the original rectangle in normalized pose in the image ...
    domain:
rect_normal = [
    xmin xmin xmax xmax;
    ymin ymax ymax ymin];

% calculate the 4 vertices, transformed to the projection domain:
rectangle_st = T_theta * rect_normal;

% convert matrix elements to double type to sort them
for i = 1 : 4
   rectangle_st(1,i) = double(rectangle_st(1,i));
   rectangle_st(2,i) = double(rectangle_st(2,i));
end

% sort the columns of rectangle_st according to increasing s-coordinate ...
    values (1st row),
%so transpose matrix to use the sortrows command, then transpose back:
sorted_rect_st = transpose(sortrows(transpose(rectangle_st)));

% now extract parameters:
s1 = sorted_rect_st(1,1);
t1 = sorted_rect_st(2,1);
s2 = sorted_rect_st(1,2);
t2 = sorted_rect_st(2,2);
s3 = sorted_rect_st(1,3);

%we want the angle theta to be 0 ≤ theta < pi/2:
theta_new = mod(theta, pi/2);
% w = rectangle width (between corners 1 and 2):
w = sqrt( (s2-s1)^2 + (t2-t1)^2);

%now calculate the height h of the projection:
%check whether corner1 is "lower" or "higher" than corner2 (relative to ...
    t-axis):
if t1 > t2
    h = w*sec(theta_new);
%else t1 < t2
elseif s1 == s2     %so we have a rectangular projection
    h = w;
else
```

```
    h = w*csc(theta_new);
end

projection = [s1 s2 s3 h];
```

A.2 Computing the Lagrange polynomial hat function

```
function L = hat_function(s,s1,s2,s3)

% L = hat_function(s,s1,s2): given breakpoints s1,s2,s3,
% returns the value of the Lagrange polyniam L(s) at point s,
% a hat function defined on these breakpoints

if s < s1
    % so s is outside the domain of the hat function
    L = 0;
elseif s ≥ s1 && s < s2
    L = (s-s1)/(s2-s1);
elseif s ≥ s2 && s < s3
    L = (s-s3)/(s2-s3);
else    % s ≥ s3 so s is outside the domain of the hat function
    L = 0;
end
```

A.3 Evaluating the projection function

```
function f = evaluate_projection_function(s,P)

% f = evaluate_projection_function (s,P): given projection P =[S;F] with S
% = set of breakpoints, F = set of corresponding function values, this
% function returns the value of the projection function at point s.

S = P(1,:);          % set of breakpoints
F = P(2,:);          % set of function values
n = size(S,2);       % number of breakpoints

%first check whether point s even falls within the projection region:
if s ≤ S(1) || s ≥ S(n)
    f = 0;
else %s does fall within S(1) < s < S(n)

    % find the index ind of the breakpoint in list S which has closest
    % value to s (and their difference d):
    [d, ind] = min(abs(S-s));

    if d == 0
        %then s is a breakpoint of set S, therefore f = value at that
        %breakpoint:
        f = F(ind);
    else % s is not a breakpoint:

        % find between which breakpoints (and their indexes) point s is
        % exactly located:
```

```
        if S(ind) < s
            left = ind;
            right = ind + 1;
        else % s < S(ind)
            left = ind - 1;
            right = ind;
        end

        % now f = linear Lagrange interpolation of projection function:
        % f = f_left * L_left(s) + f_right * L_right(s)

        %check for extremities in L functions:
        if left == 1
            % breakpoint to the left of s is the first breakpoint, so the
            % hat function is 0:
            L_left = 0;
        else
            L_left = hat_function(s,S(left-1),S(left),S(left+1));
        end
        if right == n
            % breakpoint to the right of s is the last breakpoint, so again
            % the hat function is 0:
            L_right = 0;
        else
            L_right = hat_function(s, S(right-1),S(right),S(right+1));
        end

        f = F(left) * L_left + F(right) * L_right;

    end
end
```

A.4 Combining two projections

```
function P = combine_projections(P1,P2, operator)

% P = combine_projections(P1,P2, operator): given projection1 in the form
% P1 = [S1; F1] with breakpoints S1 = [s1, s2, s3, s4,...], and
% corresponding function values F1 = [f1, f2, f3, f4,...] and projection2 =
% [S2; F2], function combines both projections according to operator ('add'
% or 'sub') into one total projection P = [S,F].

S1 = P1(1,:);   %breakpoints of projection1
S2 = P2(1,:);   %breakpoints of projection2

% new set of breakpoints = union of both sets of breakpoints (sorted in
% ascending order, double values removed):
S = union(S1, S2);
% total number of breakpoints of combined projection:
n = size(S,2);

% F will contain the new projection function values:
F = zeros(1,n);

% calculate new function values f for each breakpoint:
% (skip function values of first and last points because they are always 0)
```

```
for i = 2 : n-1
    s = S(i);    %value of breakpoint i
    % f1 = function value of projection1 at point s
    f1 = evaluate_projection_function(s, P1);
    % f2 = function value of projection2 at point s
    f2 = evaluate_projection_function(s, P2);
    % function value of total projection at point s
    % = sum of projection1 and 2 at point s:
    if operator == 'add'
        F(i) = f1 + f2;
    elseif operator == 'sub' %subtract
        F(i) = f1 - f2;
    end
end

P = [S; F];
```

A.5 Calculating the total projection

```
function total_projection = calculate_total_projection(rectangles, alpha)

% total_projection = calculate_total_projection(rectangles, alpha): given a
% certain number of parametrized rectangles in the list 'rectangles', this
% function calculates their total_projection = [S; F] where S =
% s-coordinates of breakpoints and F = their corresponding heights or
% function values

n = size(rectangles , 1);   %number of rectangles

for i = 1 : n
    xmin = rectangles(i,1);
    ymin = rectangles(i,2);
    xmax = rectangles(i,3);
    ymax = rectangles(i,4);
    phi = rectangles(i,5);
    theta = phi - alpha;

    %calculate the projection of rectangle i along angle alpha:
    proj = calculate_projection(xmin, ymin, xmax, ymax, theta);

    s1 = proj(1);
    s2 = proj(2);
    s3 = proj(3);
    h = proj(4);
    projection = [
        s1  s2  s3  (s3+(s2-s1));
        0   h   h       0];

    %total_projection = sum of the total proj thus far and new projection:
    if i == 1
        total_projection = projection;
    else
        P = combine_projections(projection, total_projection, 'add');
        total_projection = P;
    end
end
```

A.6 Calculating the possible rectangles from one projection

```
function poss_rect = calculate_rects_from_proj(s1,s2,s3,h,alpha)

% poss_rect = function_proj_to_rect(s1,s2,s3,h,alpha): given projection
% parametrization (s1,s2,s3,h) and X-ray angle alpha, function returns the
% max 4 possible parametrized rectangles in the x,y-domain that could have
% generated this projection

s4 = s3 + (s2-s1);

%Case rectangular projection:
if s1 == s2
    %then rectangle is exactly perpendicular/parallel to X-ray direction:
    theta = 0;

    %rectangle is identical to the projection:
    rectangle_st = [
        s1  s2 s3  s4;
        0   h  h   0];
    %since theta = 0, the rectangle in the normal position = rectangle in
    %s,t-domain:
    rectangle_normal = rectangle_st;

    % xmin = minimum of the first row = x-coordinates:
    xmin = min(rectangle_normal(1,:));
    % ymin = minimum of the second row = y-coordinates:
    ymin = min(rectangle_normal(2,:));
    xmax = max(rectangle_normal(1,:));
    ymax = max(rectangle_normal(2,:));
    phi = alpha + theta;
    poss_rect = [xmin ymin xmax ymax phi];

% Case triangular projection so vertices 2 and 3 align perfectly in the
% X-ray direction
elseif s2 == s3
    % s1 ≠ s2 so r = h / (s2-s1) ≠ infinity
    %so calculate rotation angle theta:
    theta = solve_for_theta (h/(s2-s1));

    %verify whether we found 1 or 2 different angles theta:
    if theta(1) ≠ theta(2)
         % two different angles theta give two different poss rects:
        n = 2;
    else
        n = 1;
    end

    for i = 1 : n
        %Tc =  clockwise rotation matrix:
        Tc_theta = [
            cos(theta(i))   sin(theta(i));
            -sin(theta(i))  cos(theta(i))];

        %vertices of the possible rectangle in the projection domain:
        t2 = 0;
        t1 = (s2-s1)*cot(theta(i));
        t4 = (s4-s2)*tan(theta(i));
```

```
57          t3 = h;
58          rectangle1_st = [
59              s2  s1  s3  s4  s2;
60              t2  t1  t3  t4  t2];
61          % rotate poss rect theta radians clockwise to its normalized pose:
62          rectangle1_normal = Tc_theta * rectangle1_st;
63          % extra parameters:
64          xmin = min(rectangle1_normal(1,:));
65          ymin = min(rectangle1_normal(2,:));
66          xmax = max(rectangle1_normal(1,:));
67          ymax = max(rectangle1_normal(2,:));
68          phi = alpha + theta(i);
69          poss_rect(i,:) = [xmin ymin xmax ymax phi];
70      end
71
72  % Case trapezoidal or triangular projection
73  else
74      % s1 ≠ s2 so r = h / (s2-s1) ≠ infinity
75      %so calculate rotation angle theta:
76      theta = solve_for_theta (h/(s2-s1));
77
78      %verify whether we found 1 or 2 angles theta:
79      if theta(1) ≠ theta(2)
80          % then we will have 4 different possible rectangles
81          n = 2;
82      else
83          %then we we have 2 different possible rectangles
84          n = 1;
85      end
86
87      for i = 1 : n
88          Tc_theta = [
89              cos(theta(i))   sin(theta(i));
90              -sin(theta(i))  cos(theta(i))];
91
92          % either t2 = 0 or t3 = 0, yielding 2 different possible rectangles
93          % for each angle theta:
94
95          %Case t2 = 0: rectangle with lowest vertex = (s2,t2), so for each
96          %theta:
97          t2 = 0;
98          t1 = (s2-s1)*cot(theta(i));
99          t4 = (s4-s2)*tan(theta(i));
100         t3 = t1 + t4;
101         rectangle1_st = [
102             s2  s1  s3  s4  s2;
103             t2  t1  t3  t4  t2];
104         rectangle1_normal = Tc_theta * rectangle1_st;
105         xmin = min(rectangle1_normal(1,:));  %minimum of the first row = ...
                x-coordinates
106         ymin = min(rectangle1_normal(2,:));  %minimum of the second row = ...
                y-coordinates
107         xmax = max(rectangle1_normal(1,:));
108         ymax = max(rectangle1_normal(2,:));
109         phi = alpha + theta(i);
110         poss_rect1(i,:) = [xmin ymin xmax ymax phi];
111
112         %Case t3 = 0: rectangles with lowest vertex = (s3, t3):
113         t3 = 0;
```

```
        t1 = (s3 - s1)*cot(theta(i));
        t4 = (s4 - s3)*tan(theta(i));
        t2 = t1 + t4;
        rectangle2_st = [
           s3   s1  s2  s4  s3;
           t3   t1  t2  t4  t3];
        rectangle2_normal = Tc_theta * rectangle2_st;
        xmin = min(rectangle2_normal(1,:));  %minimum of the first row = ...
           x-coordinates
        ymin = min(rectangle2_normal(2,:));  %minimum of the second row = ...
           y-coordinates
        xmax = max(rectangle2_normal(1,:));
        ymax = max(rectangle2_normal(2,:));
        poss_rect2(i,:) = [xmin ymin xmax ymax phi];
    end
    poss_rect = [poss_rect1; poss_rect2];
end
```

```
function theta = solve_for_theta(r)

% theta = solve_for_theta(r): given a random projection with ratio r = h /
% (s2-s1) = csc(theta)sec(theta), function returns max 2 values for theta
% between 0 and pi/2

if r ≤ 2
    theta(1) = pi/4;
    theta(2) = pi/4;
else
    syms x;          % declare x as the variable you want to solve for

    eqn = csc(x)*sec(x) == r;       %equation we want to solve

    % solve above equation for x, keeping track of parameters and
    % conditions of solution:
    [solx, param, cond] = solve(eqn,x,'ReturnConditions', true);

    assume(cond); % assume condition cond holds ("param k is an int")
    interval = [solx ≥ 0, solx ≤ pi/2];
    %find parameter value k s.t. solx lies in the interval:
    solk = solve(interval, param);

    %in solution solx: substitute parameter param for found parameter value
    %solk:
    valx = subs(solx,param,solk);
    theta = valx;
end

%disp(['r = ', num2str(double(r))]);
%disp('theta values are:');
%fprintf('%0.5e \n', theta);
```

A.7 Matching a possible projection to a given projection

```
function m = match_projections(poss_projection, projection2)

```

```
% m = [index, shift] = match_projections(poss_projection, projection2):
% finds which projection in the poss_projections list is closest in shape
% to projection 2 and returns its index number (and the shift such that
% both s1 values of the projections overlap)

n = size(poss_projection, 1); %number of possible projections

% extract parameters of projection 2:
s1_proj2 = projection2(1);
s2_proj2 = projection2(2);
s3_proj2 = projection2(3);
h_proj2 = projection2(4);

match = zeros(1,n);    %preallocate this vector
shift = zeros(1,n);

% for each possible projection:
for i = 1 : n
    % extract its parameters:
    s1_poss_proj = poss_projection(i,1);
    s2_poss_proj = poss_projection(i,2);
    s3_poss_proj = poss_projection(i,3);
    h_poss_proj = poss_projection(i,4);

    %calculate the difference in heights of proj2 and possible projection:
    height_diff = h_proj2 - h_poss_proj;

    %calculate shift s.t. s1 of poss proj + shift = s1 of proj 2
    shift(i) = s1_proj2 - s1_poss_proj;

    % now calc new parameter values of the shifted possible projection:
    s2_shifted = s2_poss_proj + shift(i);
    s3_shifted = s3_poss_proj + shift(i);

    %calc difference in s2 and s3 values between shifted poss proj and proj2:
    s2_diff = s2_proj2 - s2_shifted;
    s3_diff = s3_proj2 - s3_shifted;

    %calc the match value between the poss proj and proj2:
    match(i) = height_diff^2 + s2_diff^2 + s3_diff^2;
end

%find index of poss projection with minimum matchings cost:
[val,index] = min(match);

m = [index, shift(index)];
```

A.8 Finding the position of a rectangle from two projections

```
function shifted_found_rect = find_rect_position (found_rect, ...
    projection1, alpha1, projection2, alpha2)

% shifted_found_rect = find_rect_position(found_rect, projection1, alpha1,
% projection2, alpha2): given found_rect = [xmin, ymin, xmax, ymax, phi] in
% its normalized pose, this function finds the shift dx and dy necessary
% s.t. s1 of the projection of shifted_found_rect along alpha1 and 2  =
```

```
% s1 of projection 1 and 2

% extract parameters of the (unshifted) found rectangle:
xmin = found_rect(1);
ymin = found_rect(2);
xmax = found_rect(3);
ymax = found_rect(4);
phi = found_rect(5);

s1_proj1 = projection1(1);      % s1 of projection1
theta1 = phi - alpha1;

s1_proj2 = projection2(1);      % s1 of projection2
theta2 = phi - alpha2;

%two shift values we wish to find:
syms dx;
syms dy;
% s.t. shifted_found_rect = [xmin-dx, ymin-dy, xmax-dx, ymax-dy, phi] = ... original
% rectangle we wish to find, based on projection1 and projection2

%vector s_coord_1 = s-coordinates of shifted_found_rect projected along
%angle alpha1:
s_coord_1 = [ cos(theta1)*xmin - sin(theta1)*ymin - cos(theta1)*dx + ... sin(theta1)*dy,
        cos(theta1)*xmin - sin(theta1)*ymax - cos(theta1)*dx + ... sin(theta1)*dy,
        cos(theta1)*xmax - sin(theta1)*ymax - cos(theta1)*dx + ... sin(theta1)*dy,
        cos(theta1)*xmax - sin(theta1)*ymin - cos(theta1)*dx + ... sin(theta1)*dy ];
%same for vector s_coord_2 for angle alpha2:
s_coord_2 = [ cos(theta2)*xmin - sin(theta2)*ymin - cos(theta2)*dx + ... sin(theta2)*dy,
        cos(theta2)*xmin - sin(theta2)*ymax - cos(theta2)*dx + ... sin(theta2)*dy,
        cos(theta2)*xmax - sin(theta2)*ymax - cos(theta2)*dx + ... sin(theta2)*dy,
        cos(theta2)*xmax - sin(theta2)*ymin - cos(theta2)*dx + ... sin(theta2)*dy ];

%GOAL: solve s1_proj1 = min(s_coord_1), s1_proj2 = min(s_coord_2)

% make vectors containing only the constants of s_coord_1 and s_coord_2 to
% find their minimum elements:
s_const_1 = [   cos(theta1)*xmin - sin(theta1)*ymin,
                cos(theta1)*xmin - sin(theta1)*ymax,
                cos(theta1)*xmax - sin(theta1)*ymax,
                cos(theta1)*xmax - sin(theta1)*ymin];
s_const_2 = [   cos(theta2)*xmin - sin(theta2)*ymin,
                cos(theta2)*xmin - sin(theta2)*ymax,
                cos(theta2)*xmax - sin(theta2)*ymax,
                cos(theta2)*xmax - sin(theta2)*ymin];

%find the index of the minimum value in vector s_const_1 = index of minimum
%value in vector s_coord_1
[val,ind_proj1] = min(s_const_1);
%and do the same for vector s_const_2:
```

```
57 [val,ind_proj2] = min(s_const_2);
58
59 % Now solve equations: min(s_coord_1) == s1_proj1, min(s_coord_2) ==
60 % s1_proj2:
61 eqns = [ s_coord_1(ind_proj1) == s1_proj1, s_coord_2(ind_proj2) == ...
       s1_proj2 ];
62 vars = [dx dy];
63 [soldx, soldy] = solve(eqns, vars);
64
65 shifted_found_rect = [(xmin - soldx), (ymin - soldy), (xmax - soldx), ...
       (ymax - soldy), phi];
```

A.9 Finding a rectangle from two projections

```
1  function R = find_rect_from_projs(projection1, alpha1, projection2, ...
       alpha2, plot_on, rectangle_original)
2
3  % R = find_rect_from_projs(projection1, alpha1, projection2, alpha2,
4  % plot_on, rectangle_original): returns rectangle R = [xmin, ymin, xmax,
5  % ymax, phi] which best matches to projection 1 and 2. If plot_on == True
6  % => plots projections, and original and found rectangles, else not
7
8  % extract parameters:
9  s1_1 = projection1(1);
10 s2_1 = projection1(2);
11 s3_1 = projection1(3);
12 h_1 = projection1(4);
13
14 s1_2 = projection2(1);
15 s2_2 = projection2(2);
16 s3_2 = projection2(3);
17 h_2 = projection2(4);
18
19 if plot_on
20     %% Make plots
21
22     figure;
23
24     %plot projection1
25     subplot(2,2,1);
26     proj1 = [
27         s1_1  s2_1   s3_1  (s3_1+(s2_1-s1_1));
28         0       h_1   h_1       0];
29     plot(proj1(1,:), proj1(2,:), 'b-', 'linewidth', 2);
30     xlabel('s-axis', 'fontsize', 20);
31     ylabel('t-axis', 'fontsize', 20);
32     heading = ['Projection 1 with \alpha_1 = ', angl2str(alpha1,'pm', ...
           'radians',-5)];
33     %heading = ['Projection 1'];
34     title(heading, 'fontsize', 20);
35
36     %plot projection2
37     subplot(2,2,3);
38     proj2 = [
39         s1_2    s2_2  s3_2  (s3_2+(s2_2-s1_2));
40         0       h_2   h_2       0];
```

```
41      plot(proj2(1,:), proj2(2,:), 'g-', 'linewidth', 5);
42      hold on;
43      xlabel('s-axis', 'fontsize', 20);
44      ylabel('t-axis', 'fontsize', 20);
45      heading = ['Proj.2 (green) with \alpha_2 = ', angl2str(alpha2,'pm', ...
            'radians',-5), ' and those from Proj.1'];
46      %heading = ['Proj.2 (green) with \alpha_2 = \alpha_1 + \pi/2 and ...
            those from Proj.1'];
47      %heading = ['Proj.2 (green) and projections from poss. rects from ...
            Proj.1'];
48      title(heading, 'fontsize', 20);
49
50      %plot original rectangle we wish to find
51      subplot(2,2,4);
52      plot(rectangle_original(1,:), rectangle_original(2,:), ...
            'r-','linewidth', 4);
53      hold on;
54      %title('The original rectangle, overlaid by the found rectangle ...
            (striped lines)', 'fontsize', 20);
55      xlabel('x-axis', 'fontsize', 20);
56      ylabel('y-axis', 'fontsize', 20);
57      axis equal;
58  end
59
60  %% Calculate the possible rectangles from projection 1
61
62  %given projection1 (and angle alpha1), calculate the possible rectangles:
63  poss_rect = calculate_rects_from_proj(s1_1, s2_1, s3_1, h_1, alpha1);
64  nr_rects = size(poss_rect,1);
65
66  %for each possible rectangle calculated from projection1,
67  % plot it (if plot_on == true) and
68  % calculate its projection at angle alpha2:
69  poss_projection = zeros(nr_rects, 4);      %to initiate the lists
70  poss_original_rect = zeros(nr_rects, 2, 5);    %to inititate the lists
71  for j = 1 : nr_rects
72      %for poss_rect(j):
73      xmin = poss_rect(j,1);
74      ymin = poss_rect(j,2);
75      xmax = poss_rect(j,3);
76      ymax = poss_rect(j,4);
77      phi = poss_rect(j,5);
78      %calc angle theta betw projection2-domain and possible rectangle j:
79      theta = phi - alpha2;
80
81      %calc projection of poss rect j along angle alpha2:
82      poss_projection(j,:) = calculate_projection(xmin, ymin, xmax, ymax, ...
            theta);
83
84      rect_normal = [
85          xmin    xmin    xmax    xmax    xmin;
86          ymin    ymax    ymax    ymin    ymin];
87      T_phi = [
88          cos(phi)    -sin(phi);
89          sin(phi)    cos(phi)];
90      rect_original = T_phi * rect_normal;
91      %save this "possible original rectangle" to the list
92      poss_original_rect(j,:,:) = rect_original;
93
```

```
94      if plot_on
95          %plot poss_rect(j)
96          subplot(2,2,2);
97          plot(rect_original(1,:),rect_original(2,:), 'r-', 'linewidth', 2);
98          %plot(poss_original_rects(j,1,:),poss_original_rects(j,2,:), ...
                'r-', 'linewidth', 2);
99          axis equal;
100         hold on;
101
102         %plot poss_proj(j) besides projection2:
103         subplot(2,2,3);
104         s1 = poss_projection(j,1);
105         s2 = poss_projection(j,2);
106         s3 = poss_projection(j,3);
107         h = poss_projection(j,4);
108         proj = [
109             s1  s2  s3  (s3+(s2-s1));
110             0   h   h       0];
111         plot(proj(1,:), proj(2,:), 'b-', 'linewidth', 2);
112         hold on;
113     end
114 end
115
116
117 %% Find the rectangle and its position using projection 1 and 2
118
119 %given 4 (or less) possible projections (calculated from proj1), find the ...
       one that is
120 %closest in shape and height to projection2:
121 m = match_projections(poss_projection, projection2);
122 index = m(1);
123 proj_shift = m(2);
124 %so the rectangle corresponding to this projection is located at the found
125 %index:
126 found_rect = poss_rect(index,:);
127 %given found_rect, find its position:
128 shifted_found_rect = find_rect_position(poss_rect(index,:), projection1, ...
       alpha1, projection2, alpha2);
129 R = shifted_found_rect;
130
131 if plot_on
132     %% Make plots
133
134     %add title and labels to subplot 2:
135     subplot(2,2,2);
136     xlabel('x-axis', 'fontsize', 20);
137     ylabel('y-axis', 'fontsize', 20);
138     title('Possible rectangles calculated from Proj.1', 'fontsize', 20);
139
140     %plot found shifted_proj "onto" projection2 (to see how much they ...
           differ):
141     subplot(2,2,3);
142     s1 = poss_projection(index,1) + proj_shift;
143     s2 = poss_projection(index,2) + proj_shift;
144     s3 = poss_projection(index,3) + proj_shift;
145     h = poss_projection(index,4);
146     identical_proj = [
147             s1  s2  s3  (s3+(s2-s1));
148             0   h   h       0];
```

```
149     plot(identical_proj(1,:), identical_proj(2,:), 'b--', 'linewidth', 2);
150
151     %plot shifted_found_rect:
152     xmin = shifted_found_rect(1);
153     ymin = shifted_found_rect(2);
154     xmax = shifted_found_rect(3);
155     ymax = shifted_found_rect(4);
156     phi = shifted_found_rect(5);
157     T_phi = [
158         cos(phi)    -sin(phi);
159         sin(phi)    cos(phi)];
160     rect_normal = [
161         xmin xmin xmax xmax xmin;
162         ymin ymax ymax ymin ymin];
163     subplot(2,2,4);
164     found_rect_original = T_phi*rect_normal;
165     plot(found_rect_original(1,:),found_rect_original(2,:), 'b--', ...
            'linewidth', 2);
166     title('The original rect (red), overlaid by the found rect', ...
            'fontsize', 20);
167     xlabel('x-axis', 'fontsize', 20);
168     ylabel('y-axis', 'fontsize', 20);
169     axis equal;
170 end
```

A.10 Analyzing the stability of the solution found with Appx.A.9

```
1  function E = analyze_stability(n, alpha_diff, eps_noise)
2
3  % function E = analyze_stability(n, alpha1, alpha2, noise_eps): repeats n
4  % trials of: generating a random rectangle, calculating the projection
5  % along random alpha1 and alpha2 = alpha1 + alpha_diff, adding random
6  % epsilon noise to them, and computing the rectangle from those projections
7  % using the find_rect_from_projs function. Returns vector E containing the
8  % error between the actual and found rectangle for each trial
9
10 close all;
11
12 epsilon = eps_noise;
13
14 E = zeros(n,1);
15 for i = 1 : n
16     %% Generate a rectangle and two projections
17
18     % generate a random rectangle rect
19     rect = generate_rectangle();
20     xmin = rect(1);
21     ymin = rect(2);
22     xmax = rect(3);
23     ymax = rect(4);
24     phi = rect(5);
25     %define the transformation matrix
26     T_phi = [
27         cos(phi)    -sin(phi);
28         sin(phi)    cos(phi)];
29     rect_normal = [
```

```
30          xmin xmin xmax xmax xmin;
31          ymin ymax ymax ymin ymin];
32      rectangle_original = T_phi*rect_normal;
33
34      % calculate projections 1 and 2:
35      alpha1 = 2*pi*rand();
36      theta1 = phi - alpha1;
37      projection1 = calculate_projection(xmin, ymin, xmax, ymax, theta1);
38
39      alpha2 = alpha1 + alpha_diff;
40      theta2 = phi - alpha2;
41      projection2 = calculate_projection(xmin, ymin, xmax, ymax, theta2);
42
43      %% Add noise to the projections
44
45      %add epsilon noise to all parameter values of proj1 and 2:
46      for j = 1 : 4
47          %generate random noise value between -epsilon ≤ noise ≤ + epsilon:
48          noise = epsilon * ( -1 + 2*rand());
49          projection1(j) = projection1(j) + noise;
50
51          noise = epsilon * ( -1 + 2*rand());
52          projection2(j) = projection2(j) + noise;
53      end
54      %adjust to still have a valid projections (s1 ≤ s2 ≤ s3):
55      if projection1(1) > projection1(2)
56          projection1(1) = projection1(2);
57      elseif projection1(2) > projection1(3)
58          projection1(2) = projection1(3);        %make it into a ...
                triangular projection
59      end
60      if projection2(1) > projection2(2)
61          projection2(1) = projection2(2);
62      elseif projection2(2) > projection2(3)
63          projection2(2) = projection2(3);
64      end
65
66      %% Find the rectangle from those projections and compute the error E:
67
68      %choose whether you wish to make plots:
69      plot_on = true;
70      %plot_on = false;
71      R = find_rect_from_projs(projection1, alpha1, projection2, alpha2, ...
            plot_on, rectangle_original);
72
73      xmin = R(1);
74      ymin = R(2);
75      xmax = R(3);
76      ymax = R(4);
77      phi = R(5);
78      T_phi = [
79          cos(phi)    -sin(phi);
80          sin(phi)    cos(phi)];
81      rect_normal = [
82          xmin xmin xmax xmax xmin;
83          ymin ymax ymax ymin ymin];
84      found_rect_original = T_phi*rect_normal;
85
86      %calculate error E = difference between original and found rectangle as
```

```
87      %computed in calculate_rect_error function
88      E(i) = calculate_rect_error(rectangle_original, found_rect_original);
89
90  end
```

A.11 Calculating the error between the found and original rectangle

```
1   function error = calculate_rect_error(rectangle1, rectangle2)
2
3   % function error = calculate_rect_error(rectangle1, rectangle2): calculates
4   % difference or "error" between rectangle1 = [vertex1, v2, v3, v4, v1] and
5   % rectangle2, which is the euclidean distance between the four vertices of
6   % both rectangles
7
8   %r1_vertices = 4 x 2 matrix containing the vertices of rectangle1:
9   for i = 1 : 4
10      r1_vertices(i,:) = rectangle1(:,i);
11      r2_vertices(i,:) = rectangle2(:,i);
12  end
13
14  %find index of vertex of r2 which is closest to 1st vertex of r1:
15  for i = 1 : 4
16     d(i) = sqrt( (r1_vertices(1,1) - r2_vertices(i,1))^2 + ...
           (r1_vertices(1,2) - r2_vertices(i,2))^2 );
17  end
18  [val, index] = min(d);
19
20  %shift naming of index numbers of rectangle2 s.t. r2_vertex1 = the vertex
21  %closest to vertex1 of rectangle1, etc. going clockwise:
22  if index == 1
23      vertex(1,:) = r2_vertices(1,:);
24      vertex(2,:) = r2_vertices(2,:);
25      vertex(3,:) = r2_vertices(3,:);
26      vertex(4,:) = r2_vertices(4,:);
27  elseif index == 2
28      %new vertex1 = original vertex2
29      vertex(1,:) = r2_vertices(2,:);
30      %new vertex2 = original vertex3:
31      vertex(2,:) = r2_vertices(3,:);
32      %new vertex3 = original vertex4:
33      vertex(3,:) = r2_vertices(4,:);
34      %new vertex 4 = original vertex1:
35      vertex(4,:) = r2_vertices(1,:);
36  elseif index == 3
37      %new vertex1 = original vertex3:
38      vertex(1,:) = r2_vertices(3,:);
39      %new vertex2 = original vertex4:
40      vertex(2,:) = r2_vertices(4,:);
41      %new vertex3 = original vertex1:
42      vertex(3,:) = r2_vertices(1,:);
43      %new vertex4 = original vertex2:
44      vertex(4,:) = r2_vertices(2,:);
45  else % index = 4
46      vertex(1,:) = r2_vertices(4,:);
47      vertex(2,:) = r2_vertices(1,:);
```

```
48      vertex(3,:) = r2_vertices(2,:);
49      vertex(4,:) = r2_vertices(3,:);
50  end
51
52  %error = euclidean distance between all 4 matched vertices
53  error = 0;
54  for i = 1 :4
55      d = sqrt( (r1_vertices(i,1) - vertex(i,1))^2 + (r1_vertices(i,2) - ...
            vertex(i,2))^2 );
56      error = error + d;
57  end
58  error = double(error);      %cast from sym to double type
```

A.12 Computing the back projection image

```
1  % The back projection image
2
3  % Generates r random rectangles, calculates p projections along different
4  % angles alpha and plots the back projection image B(f)(x,y)
5
6  clear all;
7
8  % generate r random rectangles
9  r = randi([2,5],1);
10
11 % generate p random projections:
12 p = 5;
13
14 figure;
15 %% generate and plot the r random rectangles
16 x_smallest = 20;
17 x_largest = -10;
18 y_smallest = 20;
19 y_largest = -10;
20
21 for i = 1 : r
22     rectangle = generate_rectangle();
23     all_rectangles(i,:) = rectangle;
24
25     xmin = rectangle(1);
26     ymin = rectangle(2);
27     xmax = rectangle(3);
28     ymax = rectangle(4);
29     phi = rectangle(5);      % angle of rotation of rectangle
30
31     %draw the rectangle in the x,y-domain (in original pose):
32     subplot(2,3,1);
33     T_phi =[
34         cos(phi) -sin(phi);
35         sin(phi) cos(phi)];
36     rectangle_normal = [
37         xmin xmin xmax xmax xmin;
38         ymin ymax ymax ymin ymin];
39     %rotate all the vertices from the normal to the original position:
40     rectangle_original = T_phi*rectangle_normal;
```

```
41      plot(rectangle_original(1,:), rectangle_original(2, :), 'r-', ...
            'linewidth', 2);
42      hold on;
43      axis equal;
44
45      %add z-value = rectangle "height" to plot the original rectangles on
46      %the back projection:
47      rectangle_original = [rectangle_original(1,:); ...
            rectangle_original(2,:); 100 100 100 100 100];
48      subplot(2,3,[2 3 5 6]);
49      plot3(rectangle_original(1,:), rectangle_original(2,:), ...
            rectangle_original(3,:), 'r-', 'linewidth', 2);
50      axis equal;
51      hold on;
52
53      %keep track of smallest and largest x and y values encountered so far:
54      if x_smallest > min(rectangle_original(1,:))
55          x_smallest = min(rectangle_original(1,:));
56      end
57      if x_largest < max(rectangle_original(1,:))
58          x_largest = max(rectangle_original(1,:));
59      end
60      if y_smallest > min(rectangle_original(2,:))
61          y_smallest = min(rectangle_original(2,:));
62      end
63      if y_largest < max(rectangle_original(2,:))
64          y_largest = max(rectangle_original(2,:));
65      end
66
67  end
68  subplot(2,3,1);
69  title('The rectangles in the image domain', 'fontsize', 20);
70  xlabel('x-axis', 'fontsize', 16);
71  ylabel('y-axis', 'fontsize', 16);
72
73
74  %% calculate and plot the projections for different angles alpha
75
76  alpha = zeros(1,p);     % alpha vector will contain the different angles
77  %calculate and plot the p different projections:
78  for k = 1 : p
79      % evenly space alpha angles between 0 and pi:
80      alpha(k) = (k-1)*pi/p; %want to start at alpha(1)=0
81      projection = calculate_total_projection(all_rectangles, alpha(k));
82      projections{k} = projection;
83
84      subplot(2,3,4);
85      plot(projection(1,:), projection(2,:), 'linewidth', 2);
86      hold on;
87      labels{k} = ['Proj.', num2str(k), ' , \alpha = ', ...
            angl2str(alpha(k),'pm', 'radians',-2)];
88  end
89  subplot(2,3,4);
90  heading = ['Projections along different angles \alpha'];
91  title(heading, 'fontsize', 20);
92  xlabel('s-axis', 'fontsize', 16);
93  ylabel('t-axis', 'fontsize', 16);
94  legend(labels, 'fontsize', 12);
95
```

```
%% Plot the backprojection Z = B(f)(x,y) = f1 + f2 + f3 +...

%define on what grid we will plot function B(f):
x = x_smallest : 0.01 : x_largest;
m = size(x,2);      % = # x-values evaluated

y = y_smallest : 0.01 : y_largest;
n = size(y,2);      % = # y-values

Z = zeros(n,m); % preallocate matrix Z(j,i) = B(f)(x(i),y(j))
for i = 1 : m   %iterate over index of all x values

    for j = 1 : n       %iterate over index of all y values

        % at point (x,y)=(x(i),y(i)),
        %calculate B(f)(x,y) = Z(j,i) = f^(1)_(x,y) + f^(2)(x,y) + ... for
        %all projections
        for k = 1 : p   %iterate over all projections

            %function value of projection k at point (x,y) transformed to
            %projection k domain so at point s = cos(alpha)x + sin(alpha)y
            %f^(k)_(x,y) = f^(k)_(x(i),y(j)) =  f^(k)(s)
            s = cos(alpha(k))*x(i) + sin(alpha(k))*y(j);
            f = evaluate_projection_function(s, projections{k});

            %cumulatively add all those calculated intensities from the
            %projections:
            Z(j,i) = Z(j,i) + f;
        end
    end
end

% X = n x m matrix with each row = x; Y = n x m matrix with each column = y
[X,Y] = meshgrid(x,y);

subplot(2,3,[2 3 5 6]);
mesh(X, Y, Z);
view(2);
title('Back projection image', 'fontsize', 20);
xlabel('x-axis', 'fontsize', 16);
ylabel('y-axis', 'fontsize', 16);
```

A.13 Finding candidate rectangles

Instead of using image segmentation tools to find candidate rectangles using superpixels and planar patches, a set of candidate rectangles constructed by "drawing" rectangles around high intensity regions. First, the coordinates of three of the four vertices of a candidate rectangle were read from the plot. Next, it was verified that those three vertices did indeed form a rectangle (so they were at right angles to each other): if not, the y-coordinate of the last vertex was adjusted. These three vertices were then entered into the function below to obtain the parameters of the candidate rectangle in the format used in this paper.

```
function R = parametrize_rectangle(v1, v2, v3)

% R = parametrize_rectangle(v1, v2, v3): given 3 vertices of a rectangle in
% its original pose, function returns parametrized rectangle R = [xmin,
```

```
% ymin, xmax, ymax, phi]

x1 = v1(1);
y1 = v1(2);
x2 = v2(1);
y2 = v2(2);
x3 = v3(1);
y3 = v3(2);

phi = atan((y2-y1)/(x2-x1));

% 3 of the 4 vertices of the rectangle in its original pose:
R_original = [
    x1  x2  x3;
    y1  y2  y3];

%clockwise transformation matix
Tc_phi = [
    cos(phi)    sin(phi);
    -sin(phi)   cos(phi)];

R_normalized = Tc_phi * R_original;
%extract rectangle parameters:
xmin = min(R_normalized(1,:));  %minimum of the first row = x-coordinates
ymin = min(R_normalized(2,:));  %minimum of the second row = y-coordinates
xmax = max(R_normalized(1,:));
ymax = max(R_normalized(2,:));

R = [xmin ymin xmax ymax phi];
```

A.14 Calculating the local matching cost

```
function c = local_matching_cost(R, P, alpha)

% c = local_matching_cost(R, P, alpha): given parametrized rectangle R and
% total projection P =[S;F] along angle alpha, computes local matching cost
% c = area of R's projection along alpha that does not fall within P

% proj_rect = projection of R along alpha:
xmin = R(1);
ymin = R(2);
xmax = R(3);
ymax = R(4);
phi = R(5);
theta = phi - alpha;
proj_rect = calculate_projection(xmin, ymin, xmax, ymax, theta);
%change proj_rect from parametrized vector [s1 s2 s3 h] to matrix form
%proj_rect = [S;F]:
s1 = proj_rect(1);
s2 = proj_rect(2);
s3 = proj_rect(3);
h = proj_rect(4);
s4 = s3 + (s2-s1);
proj_rect = [
    s1  s2  s3  s4;
    0   h   h   0];
```

```
25
26 %proj_diff = difference between proj_rect - P:
27 proj_diff = combine_projections(proj_rect, P, 'sub');
28 n = size(proj_diff, 2);     % number of breakpoints
29 S = proj_diff(1,:); %set of breakpoints of proj_diff
30 F = proj_diff(2,:); %set of function values of proj_diff
31
32 % A = area of function proj_diff which is above the s-axis:
33 A = 0;      %to initialize
34 a = 0;
35 %for each breakpoint look at the area a under proj_diff between S(i-1) and
36 %S(i): if it is above the s-axis, calculate that area a and add to A:
37 for i = 2 : n
38     a = 0; % reset a to 0
39     if F(i) == 0    % if function value at that breakpoint = 0
40         % check whether projection between S(i-1) and S(i) is positive
41         % (above s-axis):
42         if F(i-1) > 0
43             base = S(i) - S(i-1);
44             height = F(i-1);
45             a = (base * height)/2;
46         end
47
48     elseif F(i) > 0 % so we know there is an area a to calculate
49         if F(i-1) == 0  %we have a triangle again
50             base = S(i) - S(i-1);
51             height = F(i);
52             a = (base * height)/2;
53         elseif F(i-1) > 0
54             base = S(i)-S(i-1);
55             % check which function value is higher:
56             if F(i) < F(i-1)
57                 height1 = F(i);
58                 height2 = F(i-1) - F(i);
59             elseif F(i) == F(i-1)
60                 height1 = F(i);
61                 height2 = 0;
62             else % F(i) > F(i-1)
63                 height1 = F(i-1);
64                 height2 = F(i) - F(i-1);
65             end
66             a = (base * height1) + ((base * height2)/2);
67         else % F(i-1) < 0
68             % find area of function above s-axis between S(i-1) and S(i):
69             height1 = F(i) + abs(F(i-1));
70             base1 = S(i) - S(i-1);
71             theta = atan(height1/base1);
72             height2 = F(i);
73             % we know tan(theta) = height2/base2 so
74             base2 = height2/tan(theta);
75             a = (height2*base2)/2;
76         end
77
78     else   % F(i) < 0
79         % again find area before function crosses s-axis:
80         if F(i-1) > 0
81             height1 = F(i-1) + abs(F(i));
82             base1 = S(i) - S(i-1);
83             theta = atan(height1/base1);
```

```
            height2 = F(i-1);
            % tan(theta) = height2/base2 so
            base2 = height2/tan(theta);
            a = (height2*base2)/2;
        end
    end

    % add a(area above s-axis between S(i-1) and S(i)) to total area A:
    A = A + a;
end

c = A;

%{
figure;
plot(P(1,:), P(2,:), 'b--', 'linewidth', 2);
hold on;
plot(proj_diff(1,:), proj_diff(2,:), 'g-', 'linewidth', 2);
plot(proj_rect(1,:), proj_rect(2,:), 'r--', 'linewidth', 2);

%heading = ['Projection data (blue), proj_{rect}(red), and proj_{diff} ...
    (green), with cost = ', num2str(c)];
heading = ['Projection difference (green), with cost = ', num2str(c)];
title(heading, 'fontsize', 20);
%add axis
xL = xlim;
yL = ylim;
%line([0 0], yL, 'color', 'k', 'linewidth', 1); % y-axis line in black (k)
line(xL, [0 0], 'color', 'k', 'linewidth', 1); % x-axis line
xlabel('s-axis', 'fontsize', 20);
ylabel('t-axis', 'fontsize', 20);
%}
```

A.15 Simplified implementation of the BLP

```
% BLP example

close all;

%% The original rectangles and projection data

% r = 3 rectangles to be found:
original_rectangle(1,:) = [1,10,4,20,3.81769675990223];
original_rectangle(2,:) = [6,-9,12,0,3.20309243373196];
original_rectangle(3,:) = [1,6,8,11,3.50172628564382];

p = 5; % number of projections taken at regular intervals
for l = 1 : p
   alpha(l) = (l-1)*pi/p;
   projection{l} = calculate_total_projection(original_rectangle, alpha(l));
end

%% The rectangle candidates

% create the set of parametrized candidate rectangles R:
R = calc_rect_candidates();
```

```
n = size(R,1);      %number of candidate rectangles

%% Local matching cost

%C(i,l) = local matching cost of candidate rectangle i along alpha_l = area
%of rect i's projection along alpha_l which does not fall within proj_l

C = zeros(n,p);
% C_total(i) = total matching cost of rect i (for all projections)
C_total = zeros(n,1);
for i = 1 : n % for each rect. candidate

    for l = 1 : p   %for each projection

        C(i,l) = local_matching_cost(R(i,:), projection{l}, alpha(l));
        C_total(i) = C_total(i) + C(i,l);

    end
end

%% Area of the rectangles

Area = zeros(n,1);
% calculate area of each candidate rectangle:
for i = 1 : n
    xmin = R(i,1);
    ymin = R(i,2);
    xmax = R(i,3);
    ymax = R(i,4);
    base = xmax - xmin;
    height = ymax - ymin;
    Area(i) = base * height;
end

%% Minimum number of rectangles to be selected

n_min = 3;      %usually computed algorithmically, now by sight

%% Solving the simplified BLP

Ones = ones(n,1);
nu = 0.5;   % adjustable parameter
mu = 5;     % adjustable parameter
f = C_total - nu*Area + mu*Ones;    % objective function to be minimized
intcon = 1 : n;     % set of x variables that are ints, so all
lb = zeros(n,1);    % lb ≤ x_i ≤ ub for all i = 1 : n
ub = ones(n,1);
A = -Ones';
b = -n_min;     % constraints: A*x ≤ b
% solve: min f^T*X, s.t. X(intcon) are integers, A*x ≤ b, lb ≤ X ≤ ub
X = intlinprog(f, intcon, A, b, [], [], lb, ub);
for i = 1 : n
   txt = ['X(', num2str(i), ') = ', num2str(X(i))];
   disp(txt);
end
nr_rects_chosen = sum(X);
disp(['Number of rectangles chosen = ', num2str(nr_rects_chosen)]);
```