Image restoration

Michael Jacobsen Jan Marthedal Rasmussen
Heino Sørensen

October 23, 2000

Preface

This report has been written at the Department of Mathematical Modelling (IMM) at the Technical University of Denmark (DTU). It is a midterm project, usually written half way through the study, required as a part of obtaining a Master's Degree in Engineering at DTU.

The paper focuses on a method specially suited for certain *inverse problems*, mainly the reconstruction of blurred images. That is, given a blurred image, e.g. from a distorting camera lens, reconstruct the original.

The method is called Piecewise Polynomial Truncated SVD, abbreviated PP-TSVD. It has the characteristic feature that the solutions are piecewise polynomials, hence the name. It is relatively new (1996), and has only been used on 1-dimensional problems. Here, we will see how useful the method is for 2-dimensional data, namely images.

The algorithm was developed by Per Christian Hansen from the Technical University of Denmark and Klaus Mosegaard from Copenhagen University. Per Christian Hansen is the originator of this project and has advised us during the making of this paper.

To fully benefit from all chapters, a certain amount of mathematics is required. Especially linear algebra is used throughout the paper. On the other hand, it should be possible to understand the overall conclusions in the result chapter, if the reader is interested only in seeing what the method is capable of.

Michael Jacobsen, c958319

Jan Marthedal Rasmussen, c958548

Heino Sørensen, c958600

Lyngby, June 21, 1998

Contents

Preface **i**

Symbols **1**

1 Introduction **3**
1.1 Background 3
1.2 Report objectives 4

2 The mathematical tools **5**
2.1 Singular value decomposition 5
2.1.1 The continuous case 5
2.1.2 The discrete formulation 6
2.1.3 Properties and problems of the SVD 8
2.1.4 The truncated SVD 9
2.2 Properties of the l_1-problem 11

3 The algorithm **15**
3.1 Traditional regularization 15
3.2 The PP-TSVD 15
3.3 Moving to 2 dimensions 17
3.3.1 Representing images 17
3.3.2 Discrete derivative operators in 2 dimensions 17
3.3.3 Discrete linear blurring models 19

4 Implementation **21**
4.1 The linear constrained discrete l_1 problem 21
4.1.1 Transformation to an LP-problem 21
4.1.2 Optimizations 24
4.1.3 Internal representation 26
4.1.4 Example 26
4.2 The PP-TSVD 27

5 Results **29**
5.1 Results in 1D 29
5.1.1 Varying k 29
5.1.2 Varying the derivative operator 31
5.2 Results in 2D 31
5.2.1 Restoring single point objects 31
5.2.2 Horizontal motion blur 35
5.2.3 Stacking operators 35
5.2.4 Restoring images of stars 39

6 Conclusion **45**

A PP-TSVD toolbox user guide **47**
PPTSVD 47
L1C 49
DISPVEC 51
OPERATOR2D 52
PATBLUR 53

B Source code **55**
pptsvd.m 55
l1c.m 57
dispvec.m 64
operator2d.m 65
patblur.m 66

C Internal tests **67**
l1c.m 67

Bibliography **69**

Symbols

Symbol	Meaning	Page
$\mathbf{A}_k$	A truncated version of the matrix $\mathbf{A}$.	9
$\mathbf{e}$	A vector $[1\,1\,\cdots\,1]^T$ of appropriate length.	12
$\mathbf{I}_{n\times n}$	An $n \times n$ identity matrix.	7
k	The truncation parameter for a TSVD solution.	9
$\mathbf{L}$	A linear operator used to compute seminorms such as $\|\mathbf{Lx}\|$.	15
$\mathbf{L}_p$	The discrete p'th order derivative operator.	16
$\mathcal{N}(\mathbf{A})$	Null-space of the mapping $\mathbf{x} \mapsto \mathbf{Ax}$.	7
$\mathcal{R}(\mathbf{A})$	The range of the mapping $\mathbf{x} \mapsto \mathbf{Ax}$.	7
$\text{rank}(\mathbf{A})$	The rank of matrix $\mathbf{A}$.	7
$\mathbf{x}_k$	The minimum 2-norm solution to $\mathbf{A}_k\mathbf{x} \simeq \mathbf{b}$.	9
$\mathbf{x}_{L,k}$	Solution vector obtained via the PP-TSVD.	16
$\mathbf{x}_k^{TSVD}$	A solution to the TSVD problem.	9
$\mathbf{u}_i$	The i'th left singular vector.	6
$\mathbf{v}_i$	The i'th right singular vector.	6
$\mathbf{V}_k$	A matrix containing $[\mathbf{v}_1\,\mathbf{v}_2\,\cdots\,\mathbf{v}_k]$.	9
$\mathbf{V}_k^0$	A matrix containing $[\mathbf{v}_{k+1}\,\mathbf{v}_{k+2}\,\cdots\,\mathbf{v}_n]$.	9
σ_i	The i'th largest singular value.	6

CHAPTER 1

Introduction

1.1 Background

Many problems in physics, chemistry, astronomy and mathematical physics can be described using linear models, consisting of *system*, *input* and *output*. For instance the input could be a picture, the system could describe the blurring of a picture (e.g. a distorting lens) and the output could be another picture (a blurred picture).

It is easy for such models to compute the output from given input, but the *inverse* problem—reconstructing the input from output data—is quite another task. This is because the problem is *ill-posed*. Noise will often influence the data (e.g. atmospheric noise when photographing stars) and this makes the inversion difficult. On top of this, the computers solving these problems have limited number accuracy. These factors result in, when using simple, traditional methods for solving the problems, that the reconstructed input data is unusable. That is, ill-posed problems have the unfortunate effect of being very sensitive to noise and computer (in)accuracy.

What is needed is called *regularization*, which is a way of controlling the solution. The computed solutions get "nicer" and more useful, but there is a trade-off. Some methods result in solutions that can not contain finer details, which generally makes it impossible to reconstruct the input completely. This is where user insight comes in. If the user knows that the solution should look in a certain way, perhaps a number of straight lines, the finer details can be reconstructed in an artificial way to obtain a hopefully satisfactory solution.

This trade-off concept is characteristic to regularization methods. The method must balance between two factors: Using the data to get a reconstruction as precise as possible, risking that the noise destroys the solution, and regularizing the solution, risking that the solution is nice but has nothing to do with the actual solution.

It is important to note that a best regularization method does not exist. Each

method has its own strengths and it is up to the user to chose a suitable method for the problem.

The PP-TSVD is such a regularization method. The method first generates an approximative solution where all the finer details are lost, but thereby avoiding noise to influence the result. Then fine details are created by making the solution piecewise polynomial, e.g. piecewise linear or second degree polynomials, as specified by the user. Such solution properties can be very useful. For instance, as we shall see in this report, the method can be used to reconstruct images which have constant coloured surfaces.

The method originates from an article by Per Christian Hansen and Klaus Mosegaard from 1996 [6]. In their article miscellaneous applications were shown. Among others, they looked at geological problems, but the problems were all 1-dimensional. In this report, we experiment with 2-dimensional problems, more specifically deblurring of images. This is a big area, many blurring functions and method parameters ought to be explored. Here, we start by looking at simple blurred shapes and a few blurring models.

1.2 Report objectives

In chapter 2 we will start out by explaining some general theory of ill-posed problems and a way to analyze them called the SVD, Singular Value Decomposition. We will then derive solutions via the SVD and explain the problems concerning ill-posed problems in terms of the SVD. Important theorems and mathematical tools needed for the PP-TSVD will also be treated.

Traditional regularization methods are mentioned in chapter 3 as an introduction to the PP-TSVD algorithm. We then show what properties the solutions of the PP-TSVD have, namely that they are piecewise polynomials.

In chapter 4 follows details of the implementation of the PP-TSVD algorithm. This includes a thorough explanation of an important algorithm used by the PP-TSVD.

Several results are shown in chapter 5, both in one and two dimensions. We will use the 1-dimensional data to illustrate some important properties of the PP-TSVD. For the 2-dimensional data (images) we will look at various image restoration applications: Images containing point like objects, horizontal blur, stacked operators and star photos.

In chapter 6 we sum up what we have learned working with the PP-TSVD. We conclude on which type of applications the method is suited or unsuited for. Suggestions for future projects and experiments are also mentioned.

In addition we have developed a MATLAB toolbox consisting of functions needed to experiment with the PP-TSVD method. A user guide and source code for this toolbox is provided in the appendix.

Chapter 2

The mathematical tools

2.1 Singular value decomposition

The problems we will explore in this paper are all *discretizations* of a continuous problem. But in order to understand some of the properties of the discretized model, it is important to understand some basic properties of the underlying, continuous model.

2.1.1 The continuous case

The following short review is based on the article *Numerisk behandling af Fredholm-integralligninger af første art* [4]. Many problems (including image blurring) can be described by a Fredholm integral equation of the first order. It has the following general form:

$$\int_0^1 K(s,t)\, f(t)\, dt = g(s), \qquad 0 \le s \le 1 \tag{2.1}$$

where the K (the kernel) and g (the right-hand side) are known functions and f is the unknown. K can be considered the system model, f the input to the system and g the output from the system.

The kernel K can be expressed as an infinite sum via the SVE (Singular Value Expansion):

$$K(s,t) = \sum_{i=1}^{\infty} \mu_i u_i(s)\, v_i(t) \tag{2.2}$$

The numbers μ_i are called the singular values of K and are ordered in the following fashion:

$$\mu_1 \ge \mu_2 \ge \cdots \ge \mu_n \ge \cdots \ge 0$$

The functions u_i and v_i are the singular functions of K and are mutually orthonormal, that is:

$$\langle u_i, u_j \rangle = \langle v_i, v_j \rangle = \begin{cases} 1 & \text{if} \quad i = j \\ 0 & \text{if} \quad i \neq j \end{cases}$$

where $\langle \cdot, \cdot \rangle$ is defined as

$$\langle f, g \rangle = \int_0^1 f(t)\, g(t)\, dt$$

Ill-posed problems can be analyzed using (2.1) and the SVE. One important property of ill-posed problems is that the kernel will *smoothen* the input. This means that oscillations and discontinuities in f are "flattened out" and g will seem smooth. Consequently when solving the inverse problem, the data has to be "blown up" in order to restore oscillations and discontinuities.

Ill-posed problems also have certain characteristics in terms of its singular value expansion:

- The singular values decay to zero.
- The smaller the σ_i, the more oscillations (zero-crossings) there will be in the singular functions u_i and v_i.

These properties are difficult to prove in general (see [7, p. 8]), but are essential in order to understand the regularization methods in this report.

2.1.2 The discrete formulation

A discretization of equation (2.1) can lead to a linear system of equations (we will assume $m \geq n$[1]):

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}\,, \qquad \mathbf{A} \in \mathbb{R}^{m \times n}$$

How to set up a linear system for image blurring can be seen in section 3.3.3. A powerful tool, similar to the SVE, can be applied to such discrete problems. The following description is based on [9]. The tool is called *singular value decomposition*, SVD, and is defined as:

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T = \sum_{i=1}^{n} \sigma_i \mathbf{u}_i \mathbf{v}_i^T \tag{2.3}$$

[1]The SVD also covers the case $m < n$, but assuming $m \geq n$ will make the following description more readable and the problems described in this report will all have $m \geq n$.

with $\mathbf{U} = [\mathbf{u}_1\ \mathbf{u}_2\ \cdots\ \mathbf{u}_m] \in \mathbb{R}^{m\times m}$, $\mathbf{V} = [\mathbf{v}_1\ \mathbf{v}_2\ \cdots\ \mathbf{v}_n] \in \mathbb{R}^{n\times n}$ and

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \\ & & & \end{bmatrix} \in \mathbb{R}^{m\times n}.$$

The vectors $\mathbf{u}_i$ represent an orthogonal basis for $\mathbb{R}^m$ and $\mathbf{v}_i$ an orthogonal basis for $\mathbb{R}^n$, hence $\mathbf{U}\mathbf{U}^T = \mathbf{I}_{m\times m}$ and $\mathbf{V}\mathbf{V}^T = \mathbf{I}_{n\times n}$. Like the SVE, the singular values are arranged in the following way: $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$.

We will also define a number p that is equal to the number of non-zero singular values, i.e.:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p > 0, \quad \sigma_{p+1} = \cdots = \sigma_n = 0 \tag{2.4}$$

From this definition follows:

$$\begin{aligned} \mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \ \Leftrightarrow\ \mathbf{A}\mathbf{V} = \mathbf{U}\mathbf{\Sigma} \quad &\Leftrightarrow \quad \mathbf{A}\mathbf{v}_i = \mathbf{u}_i\sigma_i,\ i = 1,\ldots,n \\ &\Leftrightarrow \quad \begin{cases} \mathbf{A}\mathbf{v}_i = \mathbf{u}_i\sigma_i & \text{for } i = 1,\ldots,p \\ \mathbf{A}\mathbf{v}_i = \mathbf{0} & \text{for } i = p+1,\ldots,n \end{cases} \end{aligned}$$

The above equations show that $(\mathbf{v}_{p+1},\ldots,\mathbf{v}_n)$ is a basis for the null-space $\mathcal{N}(\mathbf{A})$ and that $(\mathbf{u}_1,\ldots,\mathbf{u}_p)$ is a basis for the range $\mathcal{R}(\mathbf{A})$. Consequently, $\operatorname{rank}(\mathbf{A}) = p$.

We will now derive the solution(s) to the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ using the SVD. Let $\boldsymbol{\xi}$ denote the coordinates of $\mathbf{x}$ with respect to basis $(\mathbf{v}_i)$ and $\boldsymbol{\beta}$ the coordinates of $\mathbf{b}$ with respect to $(\mathbf{u}_i)$, i.e.

$$\mathbf{x} = \mathbf{V}\boldsymbol{\xi} = \xi_1\mathbf{v}_1 + \xi_2\mathbf{v}_2 + \cdots + \xi_n\mathbf{v}_n \quad \Leftrightarrow \quad \boldsymbol{\xi} = \mathbf{V}^T\mathbf{x} \tag{2.5}$$

$$\mathbf{b} = \mathbf{U}\boldsymbol{\beta} = \beta_1\mathbf{u}_1 + \beta_2\mathbf{u}_2 + \cdots + \beta_n\mathbf{u}_n \quad \Leftrightarrow \quad \boldsymbol{\beta} = \mathbf{U}^T\mathbf{b} \tag{2.6}$$

We now get:

$$\begin{gathered} \mathbf{A}\mathbf{x} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{x} = \mathbf{U}\mathbf{\Sigma}\boldsymbol{\xi} = \mathbf{b} \ \Leftrightarrow\ \mathbf{\Sigma}\boldsymbol{\xi} = \mathbf{U}^T\mathbf{b} = \boldsymbol{\beta} \ \Leftrightarrow \\ \sigma_i\xi_i = \beta_i \quad \text{for } i = 1,\ldots,n \end{gathered}$$

The left-hand side of the last equation will be zero for $i = p+1,\ldots,n$ (cf. (2.4)) and therefore $\beta_i = 0,\ i = p+1,\ldots,n$ if a solution shall exist (the system will then be *consistent*). If this is the case we have $\xi_i = \frac{\beta_i}{\sigma_i}$ for $i = 1,\ldots,p$ and ξ_i can be arbitrarily chosen for $i = p+1,\ldots,n$. Now the solution can be written as:

$$\mathbf{x} = \sum_{i=1}^{p} \frac{\mathbf{u}_i^T\mathbf{b}}{\sigma_i}\mathbf{v}_i + \mathbf{x}_{\mathcal{N}}, \quad \mathbf{x}_{\mathcal{N}} \in \mathcal{N}(\mathbf{A}) \tag{2.7}$$

If the system is not consistent, an exact solution satisfying $\mathbf{Ax} = \mathbf{b}$ cannot be found. Instead, a least squares solution, $\min \|\mathbf{b} - \mathbf{Ax}\|_2$ must be found. Here the solution is in fact the same as to a consistent system, i.e. equation 2.7 (see [9, p. 225]).

When $p < n$, an $(n-p)$-infinity of solutions exist, but the solution that minimizes $\|\mathbf{x}\|_2$ is of special interest. By inspecting $\|\mathbf{x}\|_2^2$ we get:

$$\|\mathbf{x}\|_2^2 = \mathbf{x}^T\mathbf{x} = (\mathbf{V}\boldsymbol{\xi})^T(\mathbf{V}\boldsymbol{\xi}) = \boldsymbol{\xi}^T\mathbf{V}^T\mathbf{V}\boldsymbol{\xi} = \boldsymbol{\xi}^T\boldsymbol{\xi} = \xi_1^2 + \xi_2^2 + \cdots + \xi_n^2$$

This shows, that the *least norm solution* is obtained when $\xi_{p+1} = \cdots = \xi_n = 0$, that is, when $\mathbf{x}_{\mathcal{N}} = \mathbf{0}$.

2.1.3 Properties and problems of the SVD

The SVD inherits the characteristics of the SVE when dealing with ill-posed problems, that is (see [7, p. 20]):

- The singular values σ_i decay gradually to zero with no particular gap in the spectrum. An increase of the dimension of $\mathbf{A}$ will increase the number of small singular values.

- The left and right singular vectors $\mathbf{u}_i$ and $\mathbf{v}_i$ tend to have more oscillations[2] as the index i increases, i.e., as σ_i decreases.

We will now provide a concrete example in order to illustrate these properties and to explain discrete ill-posed problems in terms of the SVD. The example was created using the function blur from REGULARIZATION TOOLS [5]. The singular values are shown in figure (2.1). It clearly shows that the singular values decay smoothly to zero—the problem is ill-posed.

The solution written in (2.7) can be seen as weights, $\frac{\mathbf{u}_i^T\mathbf{b}}{\sigma_i}$, to each of the basis vectors $\mathbf{v}_i$. These weights are examined in figure (2.2), where plots are showed for blurred pictures both with and without noise.

In almost all practical situations the data will be noisy. A problem can now occur. For data without noise, the factors $\mathbf{u}_i^T\mathbf{b}$ decay just as the singular values do (see figure 2.2(a)), which result in weights to $\mathbf{v}_i$ that are fairly constant (see 2.2(c)). But when dealing with noisy data, the factors $\mathbf{u}_i^T\mathbf{b}$ do *not* decay to zero, in fact they stabilize around the noise level which in this case is 10^{-1} (see figure 2.2(b)). As figure 2.2(d) illustrates, the weights "explode" as the singular values decay. This way, the high frequency components dominate the reconstructed picture which in turn becomes useless.

[2]We will also use the term *high/low frequency components* for singular vectors with many/few oscillations respectively.

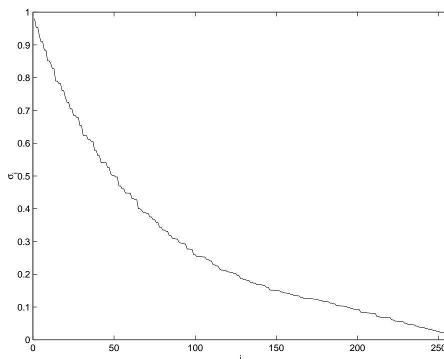


Figure 2.1: The singular values of the test problem blur.

2.1.4 The truncated SVD

A popular method used to prevent the noise from dominating the solution is the TSVD, the *truncated* SVD. This method approximates the original matrix $\mathbf{A}$ by a matrix $\mathbf{A}_k$ where only the first k singular values and functions are used (hence truncated):

$$\mathbf{A}_k = \sum_{i=1}^{k} \mathbf{u}_i \sigma_i \mathbf{v}_i^T \tag{2.8}$$

We will assume that $\sigma_i > 0$ for $i = 1, \ldots, k$. This will always be the case when handling ill-posed problems. Consequently, $\operatorname{rank}(\mathbf{A}_k) = k$. For later use, we split $\mathbf{V}$ into two the following way:

$$\mathbf{V}_k = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_k] \qquad \mathbf{V}_k^0 = [\mathbf{v}_{k+1} \ \mathbf{v}_{k+2} \ \cdots \ \mathbf{v}_n] \tag{2.9}$$

The null-space of $\mathbf{A}_k$ can now be expressed as:

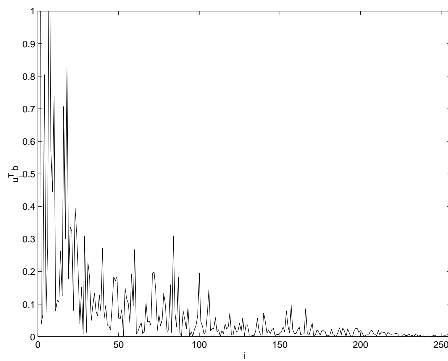
$$\mathcal{N}(\mathbf{A}_k) = \mathcal{R}\left(\mathbf{V}_k^0\right) = \left\{\mathbf{V}_k^0 \boldsymbol{\omega} \mid \boldsymbol{\omega} \in \mathbb{R}^{n \times 1}\right\} \tag{2.10}$$

The solution via the TSVD can simply be written as:

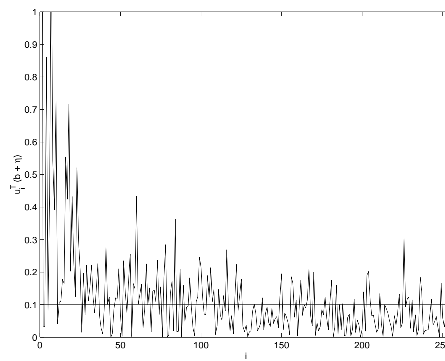
$$\mathbf{x}_k^{TSVD} = \mathbf{x}_k + \mathbf{x}_{\mathcal{N}} \quad \begin{array}{rl} \text{where} & \mathbf{x}_k = \sum_{i=1}^{k} \frac{\mathbf{u}_i^T \mathbf{b}}{\sigma_i} \mathbf{v}_i \\ \text{and} & \mathbf{x}_{\mathcal{N}} \in \mathcal{N}(\mathbf{A}_k) \end{array} \tag{2.11}$$

Similar to the SVD solution, this solution has minimum 2-norm when $\mathbf{x}_{\mathcal{N}} = \mathbf{0}$, that is, when $\mathbf{x}_k^{TSVD} = \mathbf{x}_k$.

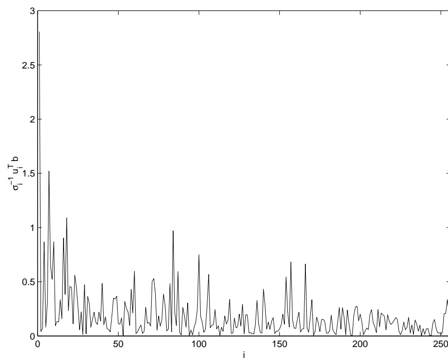
The important aspects of the SVD—at least the ones we will be needing—have now been introduced. In the next section, an important property of minimizing the 1-norm will be proven.



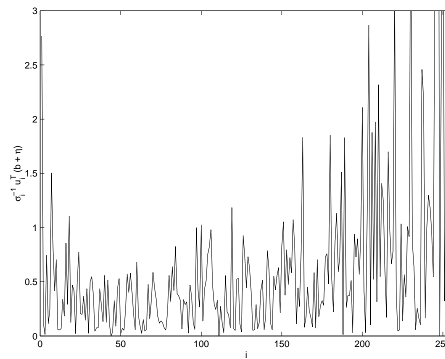
(a) $\mathbf{u}_i^T\mathbf{b}$ for a blurred picture without noise.



(b) $\mathbf{u}_i^T\mathbf{b}$ for a blurred picture with a 10^{-1} noise level.



(c) $\frac{\mathbf{u}_i^T\mathbf{b}}{\sigma_i}$ for a blurred picture without noise.



(d) $\frac{\mathbf{u}_i^T\mathbf{b}}{\sigma_i}$ for a blurred picture with a 10^{-1} noise level.

Figure 2.2: Examining the factors in the SVD-solution for data with and without noise.

2.2 Properties of the l_1-problem

The l_1-problem can be described in the following way:

$$\min_{\mathbf{x}} \|\mathbf{r}(\mathbf{x})\|_1 \ , \quad \mathbf{r}(\mathbf{x}) = \mathbf{b} - \mathbf{A}\mathbf{x} \tag{2.12}$$

where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{A} \in \mathbb{R}^{m \times n}$. The solutions of the PP-TSVD method we shall present later depends on an important property of the l_1-problem. We will show that a solution to the l_1-problem exists which has at least rank($\mathbf{A}$) zeros in the residual. The proof is a rework of the proof in [10], but using a new notation and more steps in order to improve the readability.

We have to do some preliminary work. First we transform the problem into an equivalent problem which makes the proof simpler. By exchanging rows (in both $\mathbf{A}$ and $\mathbf{b}$) and multiplying rows with -1 (in $\mathbf{A}$ and $\mathbf{b}$) it is possible to achieve a residual of the form

$$\mathbf{r}(\mathbf{x}) = \begin{bmatrix} \mathbf{b}_{(0)} \\ \mathbf{b}_{(1)} \end{bmatrix} - \begin{bmatrix} \mathbf{A}_{(0)} \\ \mathbf{A}_{(1)} \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{0} \\ \mathbf{r}_{(1)} \end{bmatrix} \begin{matrix} \}s \\ \}m-s \end{matrix}$$

where s is the number of zeros in the residual and all elements in $\mathbf{r}_{(1)}$ are positive.

Now follows a number of definitions and lemmas which will be used in the main theorem.

Definition 1 *A subset S of a linear vector space U is* **convex** *if $\mathbf{x}, \mathbf{y} \in S$ implies $\gamma\mathbf{x} + (1-\gamma)\mathbf{y} \in S$ for all $\gamma \in [0;1]$*

Lemma 1 *Let D be a compact and convex subset of $\mathbb{R}^m$ where $\mathbf{0} \notin D$. Then a $\mathbf{z} \in \mathbb{R}^m$ exists such that $\mathbf{d}^T\mathbf{z} > 0$ for all $\mathbf{d} \in D$*

Proof: We chose a vector $\mathbf{m} \in D$ that makes $||\mathbf{m}||_2$ minimal. The set D is compact thus the minimum exists[3]. If we take an arbitrary $\mathbf{d} \in D$ then any vector on the line between $\mathbf{m}$ and $\mathbf{d}$ must have a norm equal to or greater than $||\mathbf{m}||_2$. This line exists because of the convexity. Hence we have

$$\begin{aligned} 0 &\leq \|\gamma\mathbf{d} + (1-\gamma)\mathbf{m}\|_2^2 - \|\mathbf{m}\|_2^2 \\ &= (\gamma(\mathbf{d}-\mathbf{m}) + \mathbf{m})^T(\gamma(\mathbf{d}-\mathbf{m}) + \mathbf{m}) - \mathbf{m}^T\mathbf{m} \\ &= \gamma^2 \|\mathbf{d}-\mathbf{m}\|_2^2 + 2\gamma(\mathbf{d}-\mathbf{m})^T\mathbf{m} \\ &= \gamma(\gamma \|\mathbf{d}-\mathbf{m}\|_2^2 + 2(\mathbf{d}-\mathbf{m})^T\mathbf{m}) \ , \ \gamma \in [0;1] \end{aligned}$$

If we let $\gamma > 0$ decay towards zero the above is only true if

$$(\mathbf{d}-\mathbf{m})^T\mathbf{m} \geq 0 \ \Rightarrow \ \mathbf{d}^T\mathbf{m} \geq \mathbf{m}^T\mathbf{m} > 0$$

[3] To be more precise, the minimum exists because D is compact and $||\cdot||_2 : D \mapsto \mathbb{R}$ is continuous [8].

Thus we have that $\mathbf{z}$ exists ($\mathbf{z} = \mathbf{m}$). □

We now introduce a vector $\mathbf{v} \in \mathbb{R}^m$ which is defined by:

$$\mathbf{v} = \begin{bmatrix} \mathbf{d} \\ \mathbf{e} \end{bmatrix} \begin{matrix} \}s \\ \}m-s \end{matrix} \quad \|\mathbf{d}\|_\infty \leq 1$$

where $\mathbf{e} = [1\,1\,\cdots\,1]^T$. This structure implies that $\|\mathbf{r}(\mathbf{x})\|_1 = \mathbf{v}^T\mathbf{r}(\mathbf{x})$, *independently* of $\mathbf{d}$. The elements of $\mathbf{d}$ can be chosen arbitrarily as long as they lie in the interval $[-1;1]$. Hence all possible $\mathbf{v}$'s provide us with a set V which closed and bounded, i.e. compact. To proof V is convex we choose $\mathbf{v}_1, \mathbf{v}_2 \in V$ and use the definition of convexity:

$$\begin{aligned} \gamma\mathbf{v}_1 + (1-\gamma)\mathbf{v}_2 &= \gamma \begin{bmatrix} \mathbf{d}_1 \\ \mathbf{e} \end{bmatrix} + (1-\gamma) \begin{bmatrix} \mathbf{d}_2 \\ \mathbf{e} \end{bmatrix} \\ &= \begin{bmatrix} \gamma\mathbf{d}_1 + (1-\gamma)\mathbf{d}_2 \\ \mathbf{e} \end{bmatrix} \in V \end{aligned}$$

due to $\gamma\mathbf{d}_1 + (1-\gamma)\mathbf{d}_2$ all lying in the interval $[-1;1]$. Because V is compact and convex the set $U = \{\mathbf{A}^T\mathbf{v} | \mathbf{v} \in V\}$ will be compact and convex as well—a property used later. This is a consequence of $\mathbf{x} \mapsto \mathbf{A}^T\mathbf{x}$ being a continuous mapping.

Lemma 2 *Having a solution* $\mathbf{x} \in \mathbb{R}^n$ *to (2.12) then a* $\mathbf{v} \in V$ *exist such that*

$$\mathbf{A}^T\mathbf{v} = \mathbf{0} \tag{2.13}$$

Proof: We will prove this by contradiction. Assume that the condition $\mathbf{A}^T\mathbf{v} = \mathbf{0}$ is not satisfied for any $\mathbf{v}$. $\mathbf{A}^T\mathbf{v}$ forms a compact and convex set and by lemma 1 it is then possible to find a $\mathbf{c} \in \mathbb{R}^n$ which gives $\mathbf{v}^T\mathbf{A}\mathbf{c} > 0$ for all $\mathbf{v} \in V$. By choosing $\mathbf{d} = -\text{sign}\,(\mathbf{A}\mathbf{c})$ and a $\gamma > 0$ for which $\mathbf{b}_{(1)} - \mathbf{A}_{(1)}(\mathbf{x} + \gamma\mathbf{c}) > \mathbf{0}$ (that is, γ small enough) we get

$$\begin{aligned} \|\mathbf{r}(\mathbf{x} + \gamma\mathbf{c})\|_1 &= \|\mathbf{r}(\mathbf{x}) - \gamma\mathbf{A}\mathbf{c}\|_1 \\ &= \mathbf{v}^T(\mathbf{r}(\mathbf{x}) - \gamma\mathbf{A}\mathbf{c}) \\ &= \|\mathbf{r}(\mathbf{x})\|_1 - \gamma\mathbf{v}^T\mathbf{A}\mathbf{c} \\ &< \|\mathbf{r}(\mathbf{x})\|_1 \end{aligned}$$

This is a contradiction because $\mathbf{x}$ was assumed to be a solution. Hence a $\mathbf{v}$ satisfying (2.13) must exist. □

We now have the tools needed to prove the following theorem.

Theorem 1 *If* $\mathbf{A}$ *has rank* t *then a solution to (2.12) exists with the residual vector* $\mathbf{r}$ *containing at least* t *zeros.*

Proof: Assume we have a solution $\mathbf{x}$ which results in a residual vector $\mathbf{r}$ containing $s < t$ zeros. From the residual vector we get a $\mathbf{v} = [\mathbf{d}^T \ \mathbf{e}^T]^T$ satisfying $\mathbf{A}^T\mathbf{v} = \mathbf{0}$ (lemma 2). It is possible to find a $\mathbf{g} \neq \mathbf{0}$ which solves $\mathbf{A}_{(0)}\mathbf{g} = \mathbf{0}$, because $\mathbf{A}_{(0)} \in \mathbb{R}^{s\times n}$ and $s < t \leq n$. Observe that if we add $\mathbf{g}$ to the solution $\mathbf{x}$ we do not disturb the rows having zero residual, that is $\mathbf{b}_{(0)} - \mathbf{A}_{(0)}(\mathbf{x}+\gamma\mathbf{g}) = \mathbf{0}$. This gives us for γ sufficiently small (that is, $\mathbf{r}_{(1)} - \gamma\mathbf{A}_{(1)}\mathbf{g} > \mathbf{0}$):

$$
\begin{aligned}
\|\mathbf{r}(\mathbf{x}+\gamma\mathbf{g})\|_1 &= \|\mathbf{r}(\mathbf{x}) - \gamma\mathbf{A}\mathbf{g}\|_1 \\
&= \left\| \begin{bmatrix} \mathbf{0} \\ \mathbf{r}_{(1)} \end{bmatrix} - \gamma \begin{bmatrix} \mathbf{A}_{(0)} \\ \mathbf{A}_{(1)} \end{bmatrix} \mathbf{g} \right\|_1 \\
&= \left\| \begin{bmatrix} \mathbf{0} \\ \mathbf{r}_{(1)} \end{bmatrix} - \begin{bmatrix} \mathbf{0} \\ \gamma\mathbf{A}_{(1)}\mathbf{g} \end{bmatrix} \right\|_1 \\
&= \left\| \mathbf{r}_{(1)} - \gamma\mathbf{A}_{(1)}\mathbf{g} \right\|_1 \\
&= \mathbf{e}^T \left(\mathbf{r}_{(1)} - \gamma\mathbf{A}_{(1)}\mathbf{g} \right) \\
&= \|\mathbf{r}(\mathbf{x})\|_1 - \gamma\mathbf{e}^T\mathbf{A}_{(1)}\mathbf{g} \\
&= \|\mathbf{r}(\mathbf{x})\|_1 - \gamma\mathbf{e}^T\mathbf{A}_{(1)}\mathbf{g} - \gamma\mathbf{d}^T\mathbf{A}_{(0)}\mathbf{g} \\
&= \|\mathbf{r}(\mathbf{x})\|_1 - \gamma\mathbf{v}^T\mathbf{A}\mathbf{g} \\
&= \|\mathbf{r}(\mathbf{x})\|_1
\end{aligned}
$$

Hence it is possible two increase γ away from zero to find other solutions to (2.12). Because $\mathbf{A}$ has rank t we can find a $\mathbf{g}$ which gives us $\mathbf{A}_{(1)}\mathbf{g} \neq \mathbf{0}$. This enables us to adjust γ so we get a residual vector containing one more zero, *without* loosing those we have. This procedure can be applied until we are unable to solve $\mathbf{A}_{(0)}\mathbf{g} = \mathbf{0}$, which happens when we have at least t zeros in the residual vector. □

The just proven theorem is crucial in connection with the PP-TSVD method. This method *requires* the solution to have the property proven to exist by theorem 1.

When dealing with an l_1-problem, the set of vectors resulting in the minimum, $S = \{\mathbf{x} \mid \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_1 = \text{minimum}\}$, will be convex. For instance, when $\mathbf{x} \in \mathbb{R}^2$, S will be bounded by straight, connected lines. The points where the residual contains two zero elements will be the *corners* of this set.

Many methods used to solve the l_1-problem are based on the simplex algorithm. This algorithm always finds solutions exactly on the corners of this convex set. Other methods do not guarantee a solution to have this property. However, by using a single simplex iteration it will be assured that we end up in a corner. Then the solution is ready for use for the PP-TSVD algorithm.

CHAPTER 3

The algorithm

3.1 Traditional regularization

As said in the introduction, regularization must be applied to an ill-posed problem in order to get a solution that is as useful as possible. The regularization methods mentioned in the following are all based on the TSVD. The TSVD solution has an $(n-p)$-infinity of solutions (see (2.11)), but which one of these should be chosen? An obvious choice could be the solution that has the smallest 2-norm:

$$\min \|\mathbf{x}\|_2 \qquad \text{subject to} \qquad \|\mathbf{b} - \mathbf{A}_k\mathbf{x}\|_2 = \text{minimum} \tag{3.1}$$

- or expressed in words: Among all the $\mathbf{x}$'s, that minimizes $\|\mathbf{b} - \mathbf{A}_k\mathbf{x}\|_2$, choose the $\mathbf{x}$ that has minimum 2-norm. But the solution to this problem is already known as $\mathbf{x}_k$ from (2.11).

Often, it is more useful to minimize a seminorm instead, that is:

$$\min \|\mathbf{L}\mathbf{x}\|_2 \qquad \text{subject to} \qquad \|\mathbf{b} - \mathbf{A}_k\mathbf{x}\|_2 = \text{minimum} \tag{3.2}$$

Here the $\mathbf{L}$ is normally a discrete derivative operator. So instead of minimizing the "size" of $\mathbf{x}$, we can minimize the p'th derivative of $\mathbf{x}$. This approach is known as the *modified* TSVD, the MTSVD, see [7].

3.2 The PP-TSVD

The PP-TSVD changes the MTSVD slightly and simply uses the 1-norm instead of the 2-norm:

$$\min \|\mathbf{L}\mathbf{x}\|_1 \qquad \text{subject to} \qquad \|\mathbf{b} - \mathbf{A}_k\mathbf{x}\|_2 = \text{minimum} \tag{3.3}$$

This is known as the PP-TSVD, the Piecewise Polynomial TSVD. The reason for this name will be explained later, but first we will show how a solution can be computed.

The solutions to $\|\mathbf{b} - \mathbf{A}_k\mathbf{x}\|_2$ are $\mathbf{x}_k^{TSVD} = \mathbf{x}_k + \mathbf{x}_{\mathcal{N}}$, see (2.11). For later convenience we will instead write the TSVD solutions as $\mathbf{x}_k^{TSVD} = \mathbf{x}_k - \mathbf{x}_{\mathcal{N}}$ which leads to the exact same solutions, because if $\mathbf{x} \in \mathcal{N}(\mathbf{A}_k)$ then $(-\mathbf{x}) \in \mathcal{N}(\mathbf{A}_k)$.

The PP-TSVD can now be written as $\mathbf{x}_{L,k} = \mathbf{x}_k - \mathbf{x}_{\mathcal{N}}$ where $\mathbf{x}_{\mathcal{N}}$ is found as

$$\min_{\mathbf{x}_{\mathcal{N}} \in \mathcal{N}(\mathbf{A}_k)} \|\mathbf{L}(\mathbf{x}_k - \mathbf{x}_{\mathcal{N}})\|_1$$

But as stated in equation (2.10), the null-space vectors can all be written as $\mathbf{V}_k^0\boldsymbol{\omega}$, thereby obtaining the following formulation:

$$\begin{array}{c} \mathbf{x}_{L,k} = \mathbf{x}_k - \mathbf{V}_k^0\boldsymbol{\omega}_k \\ \text{where } \boldsymbol{\omega}_k \text{ is solution to} \\ \min_{\boldsymbol{\omega}} \left\|\mathbf{L}\mathbf{x}_k - \left(\mathbf{L}\mathbf{V}_k^0\right)\boldsymbol{\omega}\right\|_1 \end{array} \tag{3.4}$$

Software exists that can solve problems in the generic form of $\min_{\mathbf{x}} \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_1$, an l_1-problem. This is the method proposed in [6].

This method has the disadvantage of needing to find all the singular values and vectors. The TSVD only uses the k largest singular values/vectors, and because in most practical situations we have $k \ll n$, there would be a computational advantage in finding and using only the k largest singular values/vectors.

The problem is making sure that $\mathbf{x}_{\mathcal{N}}$ lies in $\mathbf{A}_k$'s null-space. As stated earlier, $\mathcal{N}(\mathbf{A}_k) = \mathcal{R}\left(\mathbf{V}_k^0\right)$, but this is the exact same space as the null-space of $\mathbf{V}_k^T$: $\mathcal{N}\left(\mathbf{V}_k^T\right) = \mathcal{R}\left(\mathbf{V}_k^0\right)$, that is:

$$\mathbf{z} \in \mathcal{N}(\mathbf{A}_k) \quad \Longleftrightarrow \quad \mathbf{A}_k\mathbf{z} = \mathbf{0} \quad \Longleftrightarrow \quad \mathbf{V}_k^T\mathbf{z} = \mathbf{0}$$

Now the method for finding the solution can be re-formulated as:

$$\begin{array}{c} \mathbf{x}_{L,k} = \mathbf{x}_k - \mathbf{z}_k \\ \text{where } \mathbf{z}_k \text{ is solution to} \\ \min_{\mathbf{z}} \|\mathbf{L}\mathbf{x}_k - \mathbf{L}\mathbf{z}\|_1 \ \text{ s.t. } \mathbf{V}_k^T\mathbf{z} = \mathbf{0} \end{array} \tag{3.5}$$

Now the problem is not a "pure" l_1 problem, but an l_1 problem with linear constraints. How such a problem can be solved, is shown in section 4.1.

What kind of solutions does this method produce? We can answer this question when $\mathbf{L}$ approximates the pth derivative operator, $\mathbf{L}_p$. To clearify the discussion we will use the original formulation (3.4) where the l_1-problem has no constraints, although the two formulations are equivalent. The key to understanding the solutions lies in observing the residual of the l_1-problem: $\mathbf{r} = \mathbf{L}\mathbf{x}_k - \left(\mathbf{L}\mathbf{V}_k^0\right)\boldsymbol{\omega}_k = \mathbf{L}\left(\mathbf{x}_k - \mathbf{V}_k^0\boldsymbol{\omega}_k\right) = \mathbf{L}\mathbf{x}_{L,k}$. That is, the residual is the operator $\mathbf{L}$ applied to our solution. We now make use of the fact that when

a solution to an l_1-problem $\min \|\mathbf{b} - \mathbf{A}\mathbf{x}\|_1$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, is found, the residual can always be made to contain at least n zeros[1].

In our problem, we have $\mathbf{A} = \mathbf{L}\mathbf{V}_k^0$. Because $\mathbf{L}$ is the pth derivative operator, we have $\mathbf{L} \in \mathbb{R}^{(n-p)\times n}$ and $\mathbf{V}_k^0 \in \mathbb{R}^{n\times(n-k)}$. This way $\mathbf{L}\mathbf{V}_k^0 \in \mathbb{R}^{(n-p)\times(n-k)}$ and thereby ensuring at least $n-k$ zeros in the residual vector. Consequently, there can be at most $(n-p)-(n-k) = k-p$ non-zero elements. Because $\mathbf{L}\mathbf{x}_{L,k}$ approximates the pth derivative and because $\mathbf{L}\mathbf{x}_{L,k}$ is zero except for at most $k-p$ ($\ll n-p$) elements, $\mathbf{x}_{L,k}$ itself will be a $(p-1)$th order polynomial with breakpoints exactly where $\mathbf{L}\mathbf{x}_{L,k}$ has non-zero elements.

For example, if $p = 1$ the operator $\mathbf{L}$ approximates the first derivative, and thereby obtaining solutions that are piecewise constant with at most $k-1$ breakpoint/discontinuities. Similarly, with $p = 2$ we have $\mathbf{L}$ approximating the 2nd derivative and the PP-TSVD solutions will now be piecewise linear with at most $k-2$ breakpoints. Hence the name *Piecewise Polynomial* TSVD.

3.3 Moving to 2 dimensions

We intend to use the PP-TSVD algorithm on images. However, images are 2-dimensional and are not directly applicable to the formulation of the PP-TSVD as seen in (3.5). But if we represent an image in a proper way we can utilize the PP-TSVD method easily.

3.3.1 Representing images

Images are normally represented as a matrix containing the gray tone values. It is not desirable to use a matrix in the context of the PP-TSVD—it is necessary to use a slightly different approach for an image. The image is to take the place of $\mathbf{x}$ in the formulas, hence the image must be transformed into a vector. This can be accomplished by stacking the columns of the image matrix on top of each other. This results in a (large) vector containg the gray tone values of the image. If we have a matrix $\mathbf{P} \in \mathbb{R}^{m\times n}$ which contains an image, the resulting vector $\mathbf{x} \in \mathbb{R}^{mn}$ will have the elements defined by the relation $p_{i,j} = x_{i+(j-1)m}$.

3.3.2 Discrete derivative operators in 2 dimensions

How is a $\mathbf{L}$-matrix generated which can be applied to an image represented as a vector?

In the following we show how to make a $\mathbf{L}$ which computes the 1. derivative in each point. In the one dimensional case a simple approximation to $\frac{\partial}{\partial x}$

[1] Provided that $\mathrm{rank}(\mathbf{A}) = n$ but this will always be the case because $\mathbf{L}$ and $\mathbf{V}_k^0$ have full rank. Note, that in order to get such a solution, not all algorithms can be used, cf. the remarks last in section 2.2.

is used and $\mathbf{L}$ will have the form[2]:

$$\mathbf{L}_1 = \begin{bmatrix} 1 & -1 & & & \\ & 1 & -1 & & \\ & & \ddots & \ddots & \\ & & & 1 & -1 \end{bmatrix} \tag{3.6}$$

Notice that if $\mathbf{L}_1$ is applied to a vector the result will contain one less value. In general, if we approximate the pth derivative and $\mathbf{x} \in \mathbb{R}^n$ we have $\mathbf{L}_p \in \mathbb{R}^{(n-p)\times n}$.

Generalized into 2 dimensions we have the gradient consisting of all partial derivatives $\nabla = \left[\begin{array}{cc} \frac{\partial}{\partial x_1} & \frac{\partial}{\partial x_2} \end{array}\right]$. In order to find a proper $\mathbf{L}$ we first look at $\frac{\partial}{\partial x_2}$. This derivative can be seen as the 1. derivative along each row (because x_2 refers to the row direction in the matrix representation). Hence an $\mathbf{L}$ similar to (3.6) would be a proper choice. The only problem is to place the 1's and -1's in the right places when using the representation of the previous section. The problem is best illustrated by a small example.

We have a 4×4 image represented as stacked coloumns and want to find the 1. derivative along each row. If we denote the pixels of the image like this

$$\begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} & p_{1,4} \\ p_{2,1} & p_{2,2} & p_{2,3} & p_{2,4} \\ p_{3,1} & p_{3,2} & p_{3,3} & p_{3,4} \\ p_{4,1} & p_{4,2} & p_{4,3} & p_{4,4} \end{bmatrix}$$

the vector representation would be

$$[p_{1,1}\ p_{2,1}\ p_{3,1}\ p_{4,1}\ p_{1,2}\ \cdots\ p_{4,2}\ p_{1,3}\ \cdots\ p_{4,4}]^T$$

Using the same approximation method as in (3.6), we get $\mathbf{L}_{\frac{\partial}{\partial x_2}} \in \mathbb{R}^{12\times 16}$ because we loose the approximations of the last column. With some trivial, but time consuming, work one sees that

$$\mathbf{L}_{\frac{\partial}{\partial x_2}} = \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & & & & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & & & & \\ & & & & & & & & & \\ & & & \ddots & \ddots & \ddots & \ddots & \ddots & & \\ & & & & 1 & 0 & 0 & 0 & -1 & \end{bmatrix}$$

$\mathbf{L}_{\frac{\partial}{\partial x_1}} \in \mathbb{R}^{12\times 16}$ approximates the derivative along each column:

$$\mathbf{L}_{\frac{\partial}{\partial x_1}} = \begin{bmatrix} \mathbf{D} & & & \\ & \mathbf{D} & & \\ & & \mathbf{D} & \\ & & & \mathbf{D} \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} 1 & -1 & & \\ & 1 & -1 & \\ & & 1 & -1 \end{bmatrix}$$

[2]The matrix should be normalized, but since we only use $\mathbf{L}$ operators in connection with minimizations such as $||\mathbf{L}(\cdots)||$, this is irrelevant.

Now we have the two parts of the partial gradient approximated by two different $\mathbf{L}$'s. Note that similar transformations can be done for higher order derivatives.

In the toolbox, a MATLAB function operator2d was developed to create such 2-dimensional derivative operators.

3.3.3 Discrete linear blurring models

The blurring models we will be using in this report can all be described by:

$$\mathbf{B}_{i,j} = \sum_{k=1}^{M} \sum_{l=1}^{N} h\left(i-k, j-l\right) \mathbf{X}_{k,l} \tag{3.7}$$

Here, $\mathbf{X}$ and $\mathbf{B}$ are $M \times N$ matrix representatives of the original and blurred image respectively. h denotes a point spread function that defines how surrounding pixels should be weighted when computing an output pixel. Note that (3.7) is actually a discretization of a Fredholm integral of the first kind, see (2.1).

The two blurring models used in this report are:

- *Atmospheric turbulence blur.* This is defined as

$$h\left(x,y\right) = \begin{cases} K \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) & \text{when } -\Delta \le x,y \le \Delta \\ 0 & \text{otherwise} \end{cases}$$

  where K is a normalization constant, σ a distribution parameter and Δ is a positive integer. It is just a Gaussian distribution in all directions and the far away (as specified by Δ, the bandwidth) entries are very small. By letting these be zero, the model is greatly simplified.

- *Horizontal motion blur* is defined as

$$h\left(x,y\right) = \begin{cases} \frac{1}{\Delta} & \text{when } -\frac{\Delta}{2} \le x \le \frac{\Delta}{2} \\ 0 & \text{otherwise} \end{cases}$$

  where Δ is an uneven integer.

As mentioned in the previous section (3.3), we use column vectors to represent images and the model in (3.7) uses matrix representation. The modifications are not great though and are very similar to the considerations concerning the 2-dimensional derivative operators.

Figure 3.1 illustrates how blurring works. The image on the left is the original.

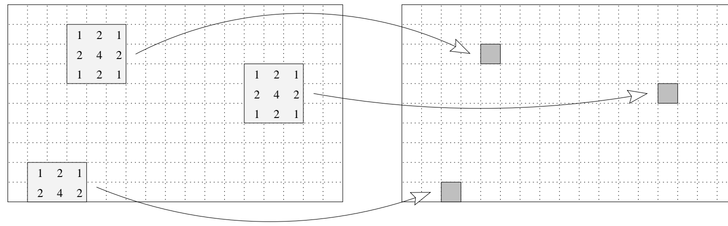


Figure 3.1: Illustrating a blurring operator.

The point spread function is represented by the matrix $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$. For each pixel the matrix covers, the gray tone values are weighed with the corresponding number, these are added up and mapped to the pixel in the image to the right. Three mappings to a single pixel are shown. This procedure is known as a 2D convolution.

It must be mentioned that when we later say that an image just has been blurred, we mean that it has been blurred with the *atmospheric turbulence blur* model. This is also the blurring model which is returned by the MATLAB function blur from the REGULARIZATION TOOLS toolbox. When the horizontal motion blur model is used, it will be specified.

CHAPTER 4

Implementation

As seen in section 3.2, the PP-TSVD consists of three major parts: Computing the singular value decomposition, computing the TSVD and finally solving the linear constrained l_1 problem.

Software exists to compute the SVD and we will not treat this particular subject in this paper. We have used the MATLAB functions `svd` and `svds`.

Computing the TSVD is very straightforward, we simply need to compute $\mathbf{x}_k$ as specified in equation 2.11.

A computation that is not as straightforward is solving the linear constrained l_1 problem seen in (3.5). An efficient method for solving this problem is the subject of the next section.

4.1 The linear constrained discrete l_1 problem

Using an article by Barrodale and Roberts [2] and their FORTRAN program we developed a MATLAB function to solve this problem. The source code of the algorithm can be found in appendix B. The underlying method is the simplex algorithm, but due to the special structure of the LP-problem, optimizations could be applied.

We begin by showing how to change the l_1 problem to a linear programming problem, we then discuss some optimizations. Following, we have a description of the internal representation and finally we show an example of the algorithm in action.

4.1.1 Transformation to an LP-problem

The l_1 problem with linear constraints can be expressed as:

$$\begin{array}{rll} \min & \|\mathbf{b}-\mathbf{A}\mathbf{x}\|_1 & \\ \text{subject to} & \mathbf{C}\mathbf{x}=\mathbf{d} & \\ \text{and} & \mathbf{E}\mathbf{x}\leq\mathbf{f} & \end{array} \tag{4.1}$$

where $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{C} \in \mathbb{R}^{k\times n}$ and $\mathbf{E} \in \mathbb{R}^{l\times n}$. The vector $\mathbf{x} \in \mathbb{R}^n$ is a vector of unknown real values.

The objective of our program is to find a vector, $\mathbf{x}^*$, which minimizes the problem stated in (4.1). It is not required that the matrices $\mathbf{A}$, $\mathbf{C}$ or $\mathbf{E}$ have full rank, nor do we assume that a vector $\mathbf{x}$ exists which satisfies the above constraints, hence the problem is stated completely general.

We reformulate the problem in (4.1) into the following linear programming problem:

$$\begin{array}{rll} \text{minimize} & \mathbf{e}^T(\mathbf{u}+\mathbf{v}) & \\ \text{subject to} & \mathbf{A}(\mathbf{x}'-\mathbf{x}'')+\mathbf{u}-\mathbf{v} & =\mathbf{b} \\ & \mathbf{C}(\mathbf{x}'-\mathbf{x}'') & =\mathbf{d} \\ & \mathbf{E}(\mathbf{x}'-\mathbf{x}'')+\mathbf{u}'' & =\mathbf{f} \\ \text{and} & \mathbf{x}',\,\mathbf{x}''\geq\mathbf{0},\,\mathbf{u},\,\mathbf{v}\geq\mathbf{0}, & \mathbf{u}''\geq\mathbf{0} \end{array} \tag{4.2}$$

where $\mathbf{e} = [1\,1\,\cdots\,1]^T$, $\mathbf{u} \in \mathbb{R}^m$, $\mathbf{v} \in \mathbb{R}^m$, $\mathbf{u}'' \in \mathbb{R}^l$ and instead of $\mathbf{x}$ we use $\mathbf{x}'-\mathbf{x}''$. Examining (4.2) an immediate feasible solution to the problem may not exist, since the restrictions $\mathbf{C}\mathbf{x} = \mathbf{d}$ and $\mathbf{E}\mathbf{x} \leq \mathbf{f}$ could be unfulfilled. Therefore, as stated in [2, p. 605], we introduce a different linear program to solve with new artificial vectors:

$$\begin{array}{rll} \text{minimize} & \mathbf{e}^T(\mathbf{u}'+\mathbf{v}')+\mathbf{e}^T\mathbf{v}'' & \\ \text{subject to} & \mathbf{A}(\mathbf{x}'-\mathbf{x}'')+\mathbf{u}-\mathbf{v} & =\mathbf{b} \\ & \mathbf{C}(\mathbf{x}'-\mathbf{x}'')+\mathbf{u}'-\mathbf{v}' & =\mathbf{d} \\ & \mathbf{E}(\mathbf{x}'-\mathbf{x}'')+\mathbf{u}''-\mathbf{v}'' & =\mathbf{f} \\ \text{and} & \mathbf{x}',\,\mathbf{x}''\geq\mathbf{0},\,\mathbf{u},\,\mathbf{v}\geq\mathbf{0}, & \mathbf{u}',\,\mathbf{v}'\geq\mathbf{0},\,\mathbf{u}'',\,\mathbf{v}''\geq\mathbf{0} \end{array} \tag{4.3}$$

In (4.3) the objective function $\mathbf{e}^T(\mathbf{u}+\mathbf{v})$ has been replaced by $\mathbf{e}^T(\mathbf{u}'+\mathbf{v}') + \mathbf{e}^T\mathbf{v}''$, which forces the program to minimize the $\mathbf{u}'$, $\mathbf{v}'$ and $\mathbf{v}''$ vectors instead. If these artificial variables are reduced to zero we have a feasible, although not necessarily optimal, solution to (4.2). A solution to (4.1) can now be found using (4.2).

The procedure of finding a feasible solution using (4.3) will be called *phase I*. Solving the l_1-problem using (4.2) will be called *phase II*.

In (4.3), we start out with a basis consisting of u_i, u'_i and u''_i, as illustrated by the example in table 4.1. If a right hand side value is negative we change the sign of the row and put the corresponding v_i, v'_i or v''_i into basis instead.

To make the next sections clearer we have to introduce some notation. When we use italic, e.g. x'_2, we refer to the *value* of this variable. If the variable is in the basis it will have the value of the right hand side—if it is not in the basis the value is zero. When we use boldface, e.g. $\mathbf{x}'_2$, we refer to the *column* in the simplex tableau. Finally a C with a variable name as index, e.g. $C_{x'_2}$, will

Basis	**b**	x_1'	x_2'	x_1''	x_2''	u_1	u_2	u_3	u_4	u_5	v_1	v_2	v_3	v_4	v_5	u_1'	v_1'	u_1''	v_1''
?	2	1	1	-1	-1	1					-1								
?	2	1	2	-1	-2		1					-1							
?	3	1	3	-1	-3			1					-1						
?	4	1	4	-1	-4				1					-1					
?	3	1	5	-1	-5					1					-1				
?	5	1	6	-1	-6											1	-1		
?	3	1	1	-1	-1													1	-1
Phase I	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-1	0	-1
Phase II	0	0	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	0	0	0

(a) The initial tableau, no basis.

Basis	**b**	x_1'	x_2'	x_1''	x_2''	u_1	u_2	u_3	u_4	u_5	v_1	v_2	v_3	v_4	v_5	u_1'	v_1'	u_1''	v_1''
u_1	2	1	1	-1	-1	1					-1								
u_2	2	1	2	-1	-2		1					-1							
u_3	3	1	3	-1	-3			1					-1						
u_4	4	1	4	-1	-4				1					-1					
u_5	3	1	5*	-1	-5					1					-1				
u_1'	5	1	6	-1	-6											1	-1		
u_1''	3	1	1	-1	-1													1	-1
Phase I	5	1	6	-1	-6						0	0	0	0	0		-2		-1
Phase II	14	5	15	-5	-15						-2	-2	-2	-2	-2		0		0

(b) A basis has been obtained.

Table 4.1: Obtaining an initial basis

denote the marginal cost of the variable (if the phase is not obvious, we will use $C^{II}_{x'_2}$ to indicate phase II, for instance).

Following this notation we have in table 4.1: $x'_2 = 0$, $\mathbf{x}'_2 = [1\,2\,3\,4\,5\,6\,1]^T$, $C^{I}_{x'_2} = 6$ and $C^{II}_{x'_2} = 15$.

4.1.2 Optimizations

Due to the structure of the l_1-problem an optimization called bypassing can be utilized. Using bypass operations enables us to skip several steps in the traditional simplex algorithm.

To illustrate the bypass procedure we present a simple constructed example. Imagine that we have arrived at the tableau in table 4.2(a). Here the pivotal column is x'_2 (largest positive marginal cost) and the vector leaving the basis is u_2 (found by quotienttest). Normally we would put x'_2 into basis at this point by performing a usual simplex iteration, resulting in $x'_2 = \frac{3}{2}$. But due to the special structure of the tableau we can put v_2 into basis by a single row operation (thereby changing the marginal cost C_{v_2} from -2 to 0), and multiplying the row by -1, see figure 4.2(b). As it turns out, this can only be done because the marginal cost $C_{x'_2}$ is not reduced to zero or below (in this example we reduce 5 to 1). We now have a right hand side which is negative (an infeasible solution), but this will be fixed later on. The just described process is a bypass operation.

We now continue to use the x'_2 as the pivotal column. The vector leaving the basis is u_3 and we check whether we can do another bypass. This however cannot be done because the marginal cost $C_{x'_2}$ would be reduced below zero $(1 - 2 \cdot 1 = -1 \leq 0)$. Instead we perform a normal simplex iteration (see 4.2(c)). Note the elements of the right hand side become non-negative (solution again feasible). This will *always* be the case, because the lower quotients are chosen first when finding the variable to leave the basis.

To resume, every time a pivotal column is found do the following:

1. Find pivot element by quotient test.

2. Do a bypass if possible. That is, if the marginal cost of the pivotal column variable will stay positive. Go to 1.

3. Perform a simplex iteration.

Note that bypass operations always has to be followed by a simplex iteration in order to arrive at a feasible solution once again.

Another optimization technique is utilized, but only in phase I. The algorithm will prefer to put only x'_i or x''_i variables into basis, since experience shows that many of these variables end up in the final simplex solution anyway. Thus we force them into basis at an early stage, hoping to save iterations—as suggested in [2, p. 605].

basis	$\mathbf{b}$	x_2'	x_2''	u_2	u_3	v_2	v_3
u_2	3	$2^\star$	-2	1		-1	
u_3	3	1	-1		1		-1
	11	5	-5	0	0	-2	-2

a) The tableau.

basis	$\mathbf{b}$	x_2'	x_2''	u_2	u_3	v_2	v_3
v_2	-3	-2	2	-1		1	
u_3	3	$1^\star$	-1		1		-1
	5	1	-1	-2	0	0	-2

b) Tableau after bypass, infeasible solution.

basis	$\mathbf{b}$	x_2'	x_2''	u_2	u_3	v_2	v_3
v_2	3	0	0	-1	2	1	-2
x_2'	3	1	-1		1		-1
	2	0	0	-2	-5	0	3

c) Tableau after simplex, solution again feasible.

Table 4.2: Explaining the bypass sequence.

4.1.3 Internal representation

As shown in table 4.1, the tableau can be quite comprehensive, and we therefore seek to compress it. This is easily done due to the structure of the LP-problem, since the $\mathbf{u}_i = -\mathbf{v}_i$, $\mathbf{u}'_i = -\mathbf{v}'_i$, $\mathbf{u}''_i = -\mathbf{v}''_i$ and $\mathbf{x}'_i = -\mathbf{x}''_i$. Consequently we need only to store the $\mathbf{x}'_1$ columns and not the $\mathbf{x}''_1$ and so on. We also know that the columns of variables in basis will consist of a single one and the rest are zero—hence we can eliminate these columns as well. In order to keep track of the elements in basis, we add a column to the very left of the tableau in which the basis variables are written, as proposed in [3].

Obviously, this much smaller tableau (storagewise) decreases the amount of memory used while computing the solution. The left out numbers can easily be computed at any time. For instance we have $\mathbf{u}_i = -\mathbf{v}_i$ and $C_{u_i} + C_{v_i} = -2$ in table 4.1(b). These equations will not change during computations, since we only use simple row operations on the tableau, so if C_{u_i} decreases, the corresponding C_{v_i} will increase accordingly.

4.1.4 Example

In order to illustrate the algorithm, we will now give a simple example using the following data:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 4 \\ 3 \end{bmatrix}$$

$$\mathbf{C} = [1 \quad 6], \quad \mathbf{d} = [5]$$

$$\mathbf{E} = [1 \quad 1], \quad \mathbf{f} = [3]$$

These data can now be put into the simplex tableau in table 4.1, which in turn is compressed into table 4.3(a).

We will now explain each step of the algorithm by solving the tableau in table 4.3(a). We see that the phase I marginal cost is the largest in the column containing x'_2, thus we have found the pivotal column. We find by quotient test ($\min_i \frac{\mathbf{b}(i)}{\mathbf{x}'_2(i)}$, $i \in \{i \in \mathbb{N} \mid \mathbf{x}'_2(i) > 0\}$[1]) that u_5 is to enter basis, and 5 becomes the pivot element.

We see if we can perform a bypass operation. By subtracting $0 \cdot 5 = 0$ from the phase I marginal cost $C^I_{x'_2}(= 6)$ and $2 \cdot 5 = 10$ from the phase II marginal cost $C^{II}_{x'_2}(= 15)$, we find that they both become positive numbers, $C^I_{x'_2} = 6$ and $C^{II}_{x'_2} = 5$. This is easiest seen in table 4.1. Then we replace u_5 by v_5 and the sign of the row is changed. This completes the bypass operation, see table 4.3(b).

The x'_2 column is still the pivotal column, and we find that u'_1 is to leave basis. A bypass operation cannot be done (since $5 - 2 \cdot 6 = -7 \leq 0$) and a

[1] To avoid confusion with the introduced notation, we use $\mathbf{b}(i)$ for the i'th element of the vector $\mathbf{b}$.

Basis	**b**	x'_1	x'_2
u_1	2	1	1
u_2	2	1	2
u_3	3	1	3
u_4	4	1	4
u_5	3	1	5*
u'_1	5	1	6
u''_1	3	1	1
Phase I	5	1	6
Phase II	14	5	15

(a) Initial tableau.

Basis	**b**	x'_1	x'_2
u_1	2	1	1
u_2	2	1	2
u_3	3	1	3
u_4	4	1	4
v_5	-3	-1	-5
u'_1	5	1	6*
u''_1	3	1	1
Phase I	5	1	6
Phase II	8	3	5

(b) After one bypass.

Basis	**b**	x'_1	u'_1
u_1	7/6	5/6	-1/6
u_2	1/3	2/3*	-1/3
u_3	1/2	1/2	-1/2
u_4	2/3	1/3	-2/3
v_5	7/6	-1/6	5/6
x'_2	5/6	1/6	1/6
u''_1	13/6	5/6	-1/6
Phase I	0	0	-1
Phase II	23/6	13/6	5/6

(c) After one ordinary simplex iteration.

Table 4.3: Completing phase I

standard simplex iteration is made. See table 4.3(c). Note how the simplex iteration made the basis solution feasible once again.

The algorithm has now finished its first iteration[2], and the value of the objective function of phase I is 0. Hence we are done with phase I, and move on to phase II. The current solution is $x_1 = 0 \wedge x_2 = \frac{5}{6}$ and the constraints are now fulfilled. It is not an optimal solution since the phase II marginal cost $C^{II}_{x'_1}$ is positive.

We now choose x'_1 as the pivotal column, and we find that the element leaving the basis is u_2, and the pivotelement is $\frac{2}{3}$. Again we notice that a bypass operation is possible, since $\frac{13}{6} - 2 \cdot \frac{2}{3} = \frac{5}{6} > 0$. We do the bypass, thereby exchanging u_2 with v_2 in the basis (see table 4.4(a)).

The same column (x'_1) is still the pivotal column, and we find that the pivotelement is $\frac{1}{2}$ and the variable to leave the basis is u_3. This time we must perform a normal simplex iteration. See table 4.4(b).

Now all the marginal costs are non-positive, including the suppressed columns (note that u'_i, v'_i, and v''_i *cannot* enter basis in phase II—this would cause the constraints not to be fulfilled), and we have an optimal solution to our problem in table 4.4(b). An optimal l_1 approximation of the problem is $x^\star_1 = 1 \wedge x^\star_2 = \frac{2}{3}$. The approximation error is $\frac{7}{3}$, and the slack on the inequality constraint is $\frac{4}{3}$.

4.2 The PP-TSVD

We now have all the tools ready to use the PP-TSVD method. The overall algorithm can be described as in [6, p. 518]. There is a slight modification though, because we now solve the l_1 problem with constraint (see (3.5)) as opposed to

[2] We use the term iteration for (a number of) bypass operations followed by a simplex operation.

Basis	$\mathbf{b}$	x_1'	u_1'
u_1	7/6	5/6	-1/6
v_2	1/3	-2/3	1/3
u_3	1/2	1/2*	-1/2
u_4	2/3	1/3	-2/3
v_5	5/6	1/6	1/6
x_2'	7/6	-1/6	5/6
u_1''	13/6	5/6	-1/6
Phase II	19/6	5/6	-1/6

(a) Beginning phase II with a bypass operation.

Basis	$\mathbf{b}$	u_3	u_1'
u_1	1/3	-5/3	2/3
v_2	1/3	4/3	-1/3
x_1'	1	2	-1
u_4	1/3	-2/3	-1/3
v_5	4/3	1/3	2/3
x_2'	2/3	-1/3	1/3
u_1''	4/3	-5/3	2/3
Phase II	7/3	-5/3	2/3

(b) The final tableau.

Table 4.4: Completing phase II

the l_1 problem without constraints (see (3.4)). Using the new formulation, the PP-TSVD algorithm has the following structure:

1. Choose an initial value of k.
2. Compute the SVD for $\mathbf{A}$. Typically not all singular values and vectors are needed.
3. Compute the TSVD solution $\mathbf{x}_k = \sum_{i=1}^{k} \frac{\mathbf{u}_i^T \mathbf{b}}{\sigma_i} \mathbf{v}_i$.
4. Form the matrix $\mathbf{V}_k = [\mathbf{v}_1\ \mathbf{v}_2\ \ldots\ \mathbf{v}_k]$.
5. Solve the linear constrained l_1 problem for $\mathbf{z}_k$:
   $\min \|\mathbf{L}\mathbf{x}_k - \mathbf{L}\mathbf{z}\|_1 \quad \text{s.t.} \quad \mathbf{V}_k^T \mathbf{z} = \mathbf{0}$.
6. Compute the PP-TSVD solution $\mathbf{x}_{L,k} = \mathbf{x}_k - \mathbf{z}_k$.
7. Inspect the solution.
8. If necessary, adjust k and go to 3.

The remark in step 2 is important. If the user of the algorithm only wishes to experiment with k values less or equal to k_{max} then only the k_{max} singular values (and vectors) need to be found in this step.

CHAPTER 5

Results

In this chapter we will present results both in 1 and 2 dimensions. Although the 2 dimensional problems are the most interesting, the most unexplored, the results in 1 dimension are so illustrative for the PP-TSVD method that they deserve a whole section.

When moving to 2 dimensions, there are a large amount of possibilities both in terms of the blurring model and the derivative operator. As it will be shown, the more the user knows about the blurring model and the effects of the different operators, the better the results.

5.1 Results in 1D

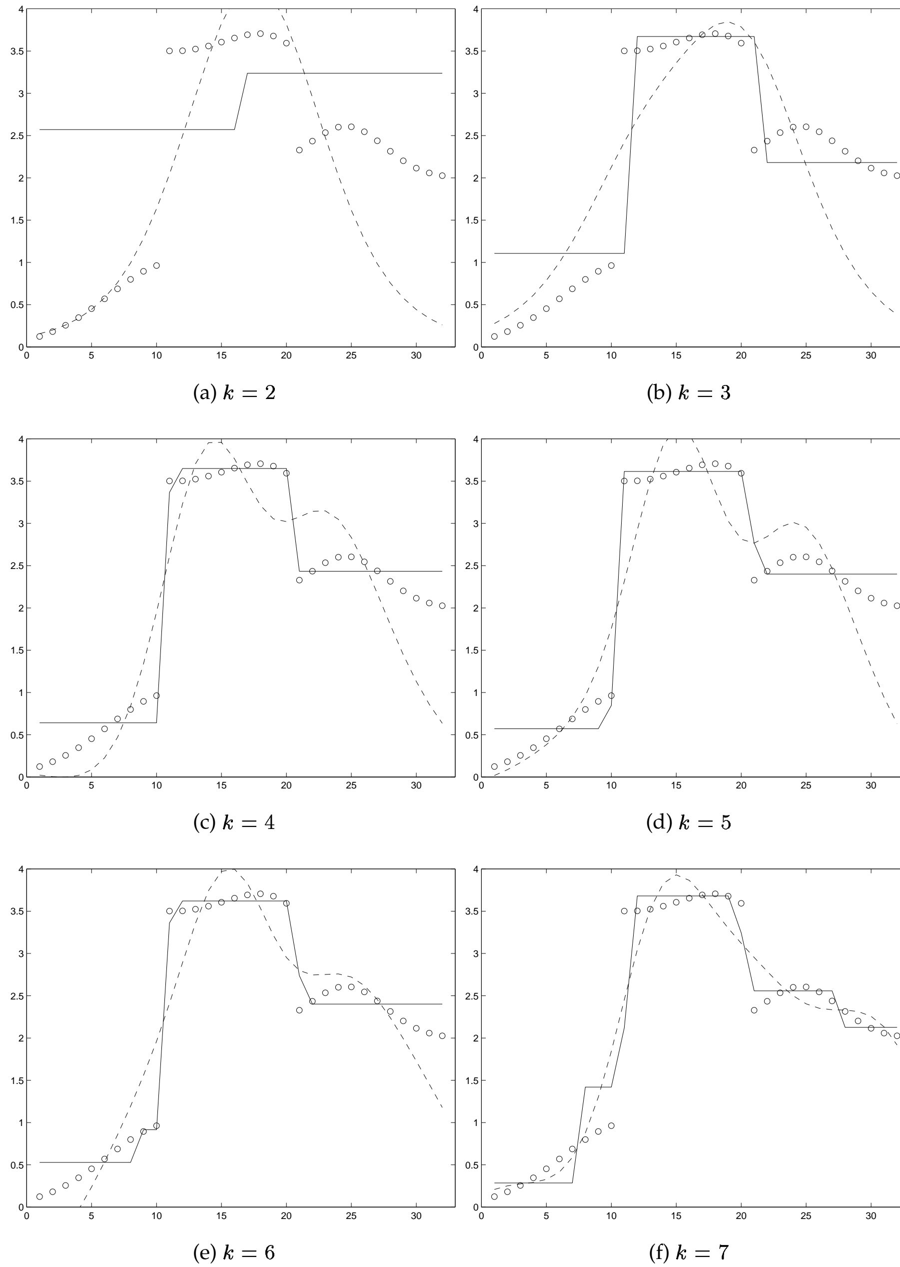
As just mentioned, the one dimensional results are very good when it comes to illustrating the use of the PP-TSVD parameters, the truncation parameter k and the derivative operator $\mathbf{L}$.

We will use the shaw test problem from the REGULARIZATION TOOLS [5] package.

5.1.1 Varying k

The k parameter controls two things at the same time: The truncation parameter when calculating the TSVD and the number of breakpoints in the restored data. In all the plots in figure 5.1 the operator approximates the first derivative, thereby obtaining piecewise constant functions as solutions. The effect of varying k is clear, there are exactly $k-1$ breakpoints in each plot. Note that we have introduced two discontinuities to the shaw test problem in order to show the PP-TSVD algorithms ability to handle such.

To remind of the general result from section 3.2: When using the p'th order derivative operator, there will be at most $k-p$ breakpoints.



(a) $k = 2$ (b) $k = 3$

(c) $k = 4$ (d) $k = 5$

(e) $k = 6$ (f) $k = 7$

Figure 5.1: Varying the parameter k on the testproblem shaw with 2 discontinuities. Lines drawn with circles represent the original, dashed lines the TSVD solution and solid lines the PP-TSVD solution. Notice that there are exactly $k-1$ breakpoints in each figure.

5.1.2 Varying the derivative operator

Another way of controlling the restored result is to vary the derivative operator. In figure 5.2 different derivative operators are applied to the shaw test problem (without discontinuities). Each row use the same derivative operator $\mathbf{L}_p$ and in each row k is varied to illustrate the connection between the two. The vertical dotted lines in each plot show where the breakpoints are. Again, the consequences of varying $\mathbf{L}$ (and k) are clear.

It should be noted that the 1-dimensional results can also be used to simply locate the breakpoints. These can then be fed into other methods for better approximations.

5.2 Results in 2D

For most of the following results, 16×16 gray scale images have been used. This was a matter of being able to get solutions within a resonable amount of time. Because the use of the PP-TSVD method with images was so new, we started out testing only on simple shapes. This would reveal which types of objects were easy to restore and which were difficult.

In the last section we present some results that are reconstructions of gray scale star images. Both 20×20 and 50×50 images were used. The use of the PP-TSVD method is not impressive handling these images. To fully conclude on the usefulness on such images, more testing would be required. They have been included, though, to give some hints on where to be careful, and how better solutions may be achieved.

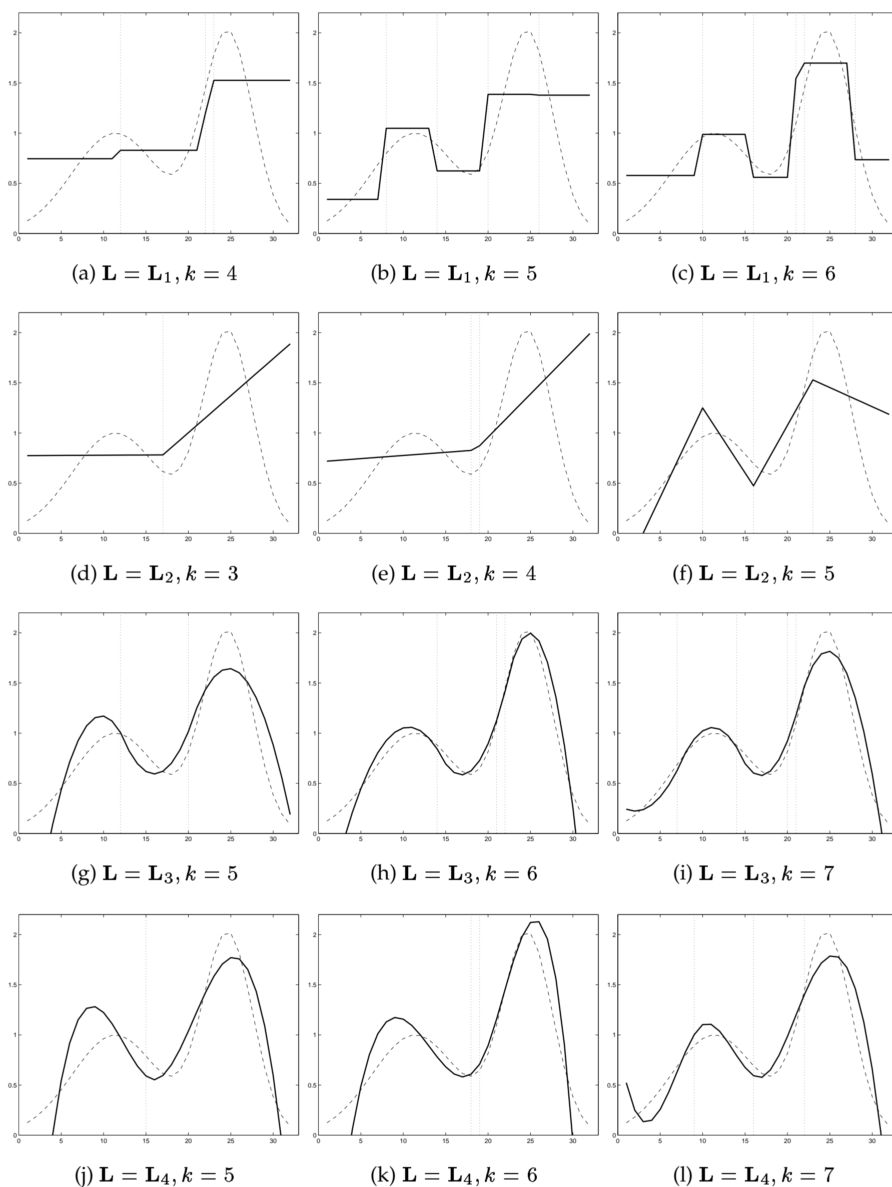
5.2.1 Restoring single point objects

The PP-TSVD method has shown to be very useful when wishing to restore point-like objects. The obvious choice of the $\mathbf{L}$ operator is the identity matrix, which produces solutions that are zero except for (at most) k places.

The test image is shown in figure 5.3(a) and the blurred picture without noise is seen in figure 5.3(b). The blur model used is from the blur test problem in REGULARIZATION TOOLS. Figures 5.3(c) and 5.3(d) show the TSVD and PP-TSVD solutions respectively with $k = 20$.

The TSVD solution is clearly useless in this context, while the PP-TSVD method restores the image almost perfectly (by comparing with the original we get $\frac{\|\mathbf{x}_{orig} - \mathbf{x}_{pptsvd}\|_2}{\|\mathbf{x}_{orig}\|_2} \simeq 10^{-12}$).

In practical situations, e.g. when restoring an astronomical image of stars, there will be noise involved. Noisy test images have been constructed by adding normally distributed noise to the image shown in 5.3(b). Restoring the dot image at different noise levels can be seen in figure 5.4. Each column represents restorations of images with the same noise level, having a dispersion of 10^{-3}, 10^{-2} and 10^{-1} respectively.



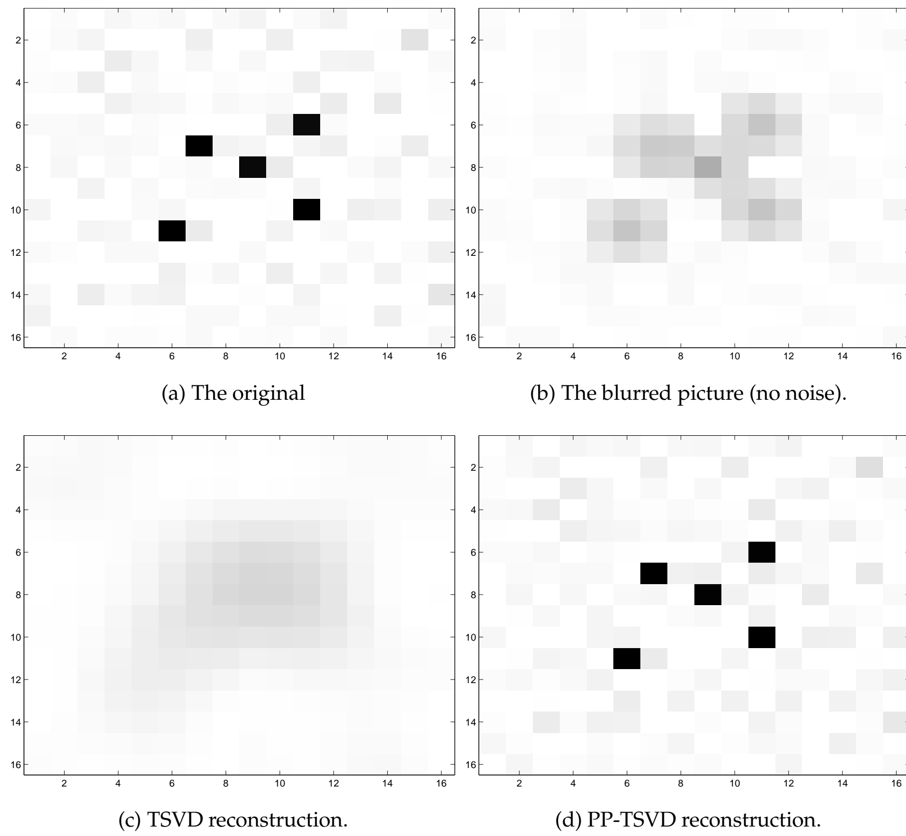
(a) $\mathbf{L} = \mathbf{L}_1, k = 4$ (b) $\mathbf{L} = \mathbf{L}_1, k = 5$ (c) $\mathbf{L} = \mathbf{L}_1, k = 6$

(d) $\mathbf{L} = \mathbf{L}_2, k = 3$ (e) $\mathbf{L} = \mathbf{L}_2, k = 4$ (f) $\mathbf{L} = \mathbf{L}_2, k = 5$

(g) $\mathbf{L} = \mathbf{L}_3, k = 5$ (h) $\mathbf{L} = \mathbf{L}_3, k = 6$ (i) $\mathbf{L} = \mathbf{L}_3, k = 7$

(j) $\mathbf{L} = \mathbf{L}_4, k = 5$ (k) $\mathbf{L} = \mathbf{L}_4, k = 6$ (l) $\mathbf{L} = \mathbf{L}_4, k = 7$

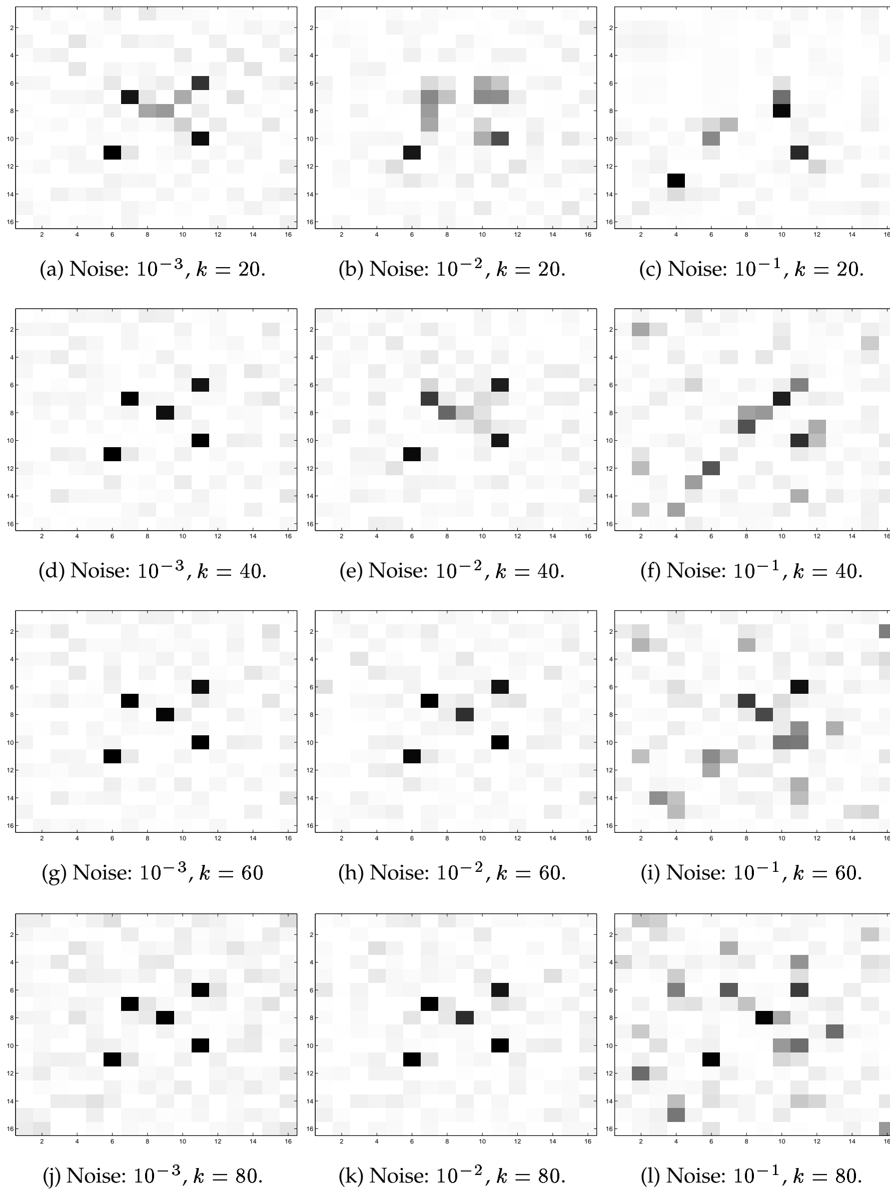
Figure 5.2: Using different derivative operators. The dashed lines represent the original, the solid lines the PP-TSVD solution. The discontinuities are illustrated by vertical lines. Notice when using the pth derivative operator $\mathbf{L} = \mathbf{L}_p$ and truncation parameter k, there are exactly $k - p$ breaks.



(a) The original (b) The blurred picture (no noise).

(c) TSVD reconstruction. (d) PP-TSVD reconstruction.

Figure 5.3: Illustrating the original and the blurred image, together with the TSVD and the PPTSVD reconstructions ($k = 20$).



(a) Noise: 10^{-3}, $k = 20$.

(b) Noise: 10^{-2}, $k = 20$.

(c) Noise: 10^{-1}, $k = 20$.

(d) Noise: 10^{-3}, $k = 40$.

(e) Noise: 10^{-2}, $k = 40$.

(f) Noise: 10^{-1}, $k = 40$.

(g) Noise: 10^{-3}, $k = 60$

(h) Noise: 10^{-2}, $k = 60$.

(i) Noise: 10^{-1}, $k = 60$.

(j) Noise: 10^{-3}, $k = 80$.

(k) Noise: 10^{-2}, $k = 80$.

(l) Noise: 10^{-1}, $k = 80$.

Figure 5.4: Restoring 5 points at various noise levels.

The images in figure 5.4 reveal two important things. They show that the method, in this case, can handle images which have a noise level less than 10^{-2}. Furthermore they show that one has to be careful when choosing the k parameter. For instance, when focusing on the images reconstructed from an image with a 10^{-3} noise level, the best choice of k lies around 40-60. When using a higher k value, the quality falls because the noise influences the reconstruction too much, cf. figure 2.2(d) on page 10.

5.2.2 Horizontal motion blur

Directional derivatives can successfully be used in connection with distortions in certain directions. This section will focus on horizontal blur of images. Horizontal motion blur can arise, for example, when an object moves quickly relative to the closing time of a camera shutter.

If we want piecewise constant surfaces as solutions, then an obvious choice of operator for this purpose is the first order derivative in horizontal direction. Horizontal direction, of course, because this is the blurring direction. The resulting reconstructions can be seen in figure 5.5. It should be noted, that a reconstruction was found for each of the k values $20, 40, \ldots, 240$ and that the best one was found as the one with minimum 2-norm when subtracted from the original.

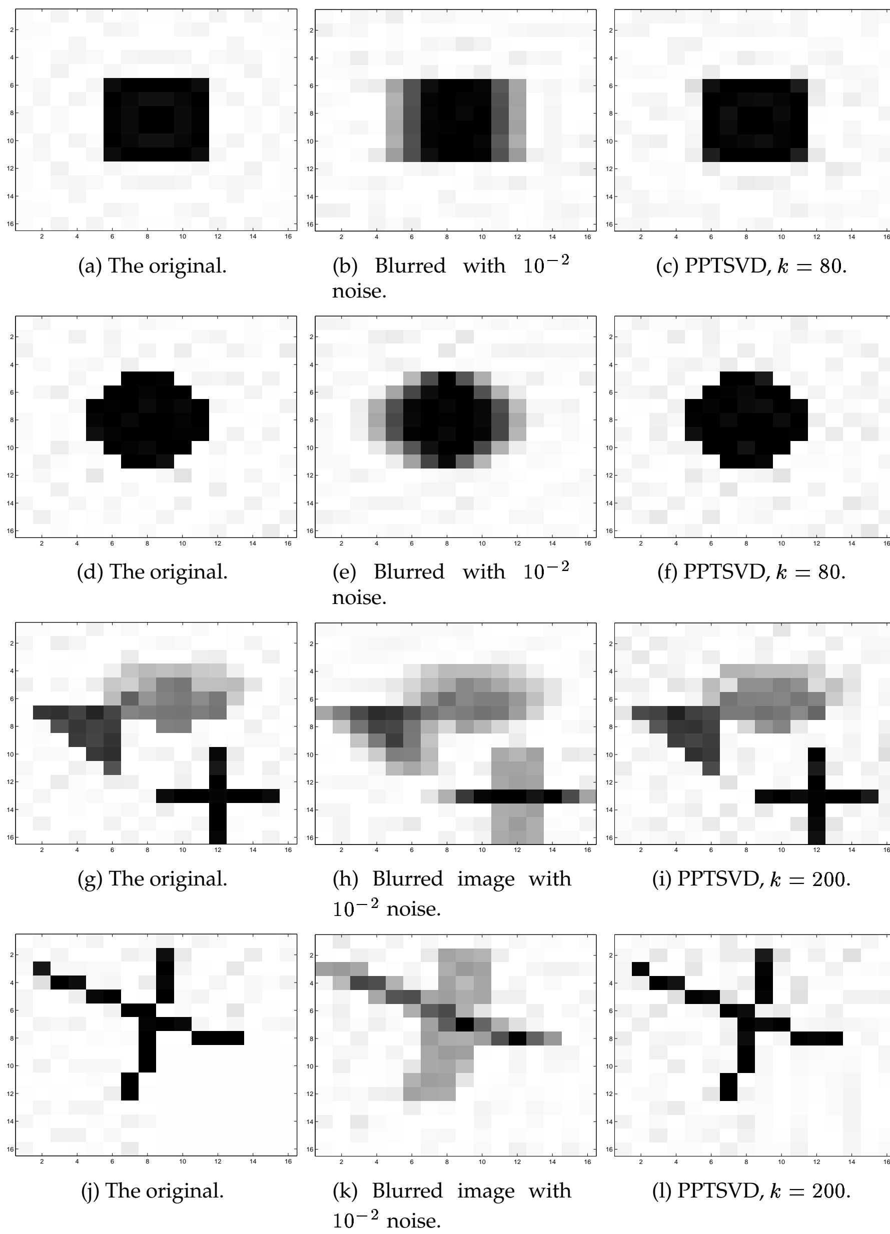
The reconstructions are most successful for the simple shapes, but the more complex shapes get resonable results too. Not very satisfying is it though, that the best k value for those images is very high—using $k = 200$ out of the possible 256 singular values/vectors.

It is worth noticing that the horizontal blur works row-wise (using a matrix point of view on images), that is, the blurring of a certain pixel only depends upon pixels in the *same* row. In a similar manner, the derivative operator also works row-wise. This way of using the PP-TSVD is somehow like using an 1-dimensional approach on each row seperately. It is not exactly the same though, because the k parameter, which controls the number of places the derivative is zero, is "distributed" among all the rows. Figure 5.6 shows a (bad) reconstruction with a low k-value to illustrate this.

How the k and p parameters control the solutions in the case of 2-dimensional data with directional derivatives, can be derived similarly to that in section 3.2. Here, $\mathbf{L}_p \in \mathbb{R}^{n(n-p)\times n^2}$, but this actually results is the exact same as the 1-dimensional case: The operator applied to the solution will have at most $k-p$ non-zero elements. This means, for images, that no more than $k-p$ points will have a non-zero derivative.

5.2.3 Stacking operators

Stacking the derivative operators is a way of using a derivative operator in two directions at the same time. This can be done by identifying the following two minizations as being equivalent:



(a) The original. (b) Blurred with 10^{-2} noise. (c) PPTSVD, $k = 80$.

(d) The original. (e) Blurred with 10^{-2} noise. (f) PPTSVD, $k = 80$.

(g) The original. (h) Blurred image with 10^{-2} noise. (i) PPTSVD, $k = 200$.

(j) The original. (k) Blurred image with 10^{-2} noise. (l) PPTSVD, $k = 200$.

Figure 5.5: Restoring various images applied with horizontal blur.

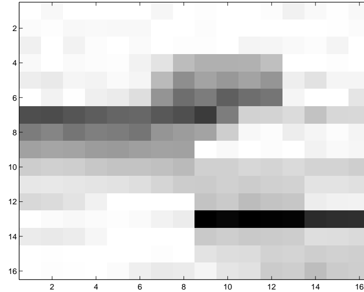


Figure 5.6: Using a directional derivative in horizontal direction results in solutions with piecewise constant functions for each row.

$$\min_{\mathbf{x}} \left(\|\mathbf{b} - \mathbf{A}\mathbf{x}\|_1 + \|\mathbf{d} - \mathbf{C}\mathbf{x}\|_1 \right) \quad \text{and} \quad \min_{\mathbf{x}} \left\| \begin{bmatrix} \mathbf{b} \\ \mathbf{d} \end{bmatrix} - \begin{bmatrix} \mathbf{A} \\ \mathbf{C} \end{bmatrix} \mathbf{x} \right\|_1 \tag{5.1}$$

This can easily be verified by writing out the sums. If we now have two operators $\mathbf{L}_1$ and $\mathbf{L}_2$ we can obtain a stacked version of the PP-TSVD simply by using $\mathbf{L} = \begin{bmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \end{bmatrix}$ instead. The usual formulation now becomes:

$$\min_{\mathbf{x}} \|\mathbf{L}\mathbf{x}_k - \mathbf{L}\mathbf{z}\|_1 \quad \text{equivalent to} \quad \min_{\mathbf{x}} \left\| \begin{bmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \end{bmatrix} \mathbf{x}_k - \begin{bmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \end{bmatrix} \mathbf{z} \right\|_1$$

which in turn by (5.1) is equivalent to

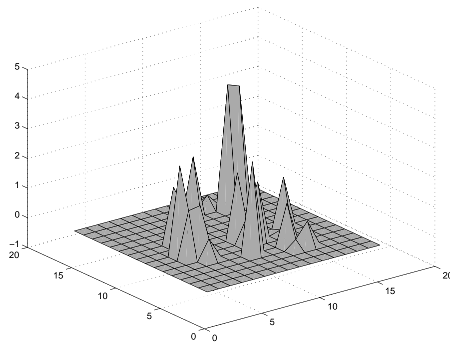
$$\min_{\mathbf{x}} \left(\|\mathbf{L}_1\mathbf{x}_k - \mathbf{L}_1\mathbf{z}\|_1 + \|\mathbf{L}_2\mathbf{x}_k - \mathbf{L}_2\mathbf{z}\|_1 \right)$$

which is exactly what we want.

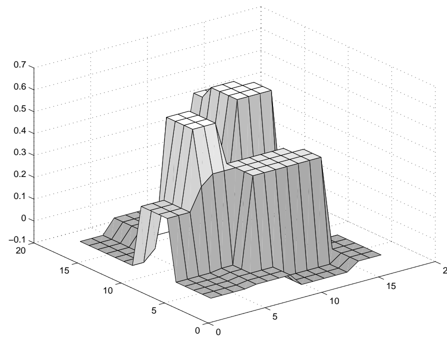
Stacking the operators in this way has shown to be very powerful. They will produce solutions with similar properties to those seen in figure 5.2 (page 32) where the solutions in the 1-dimensinal case become piecewise polynomials. Figure 5.7 shows solutions produced with stacked operators (except 5.7(a) which just uses $\mathbf{L} = \mathbf{I}$). The solution are seen to be very similar to the 1-dimensional case. In fact, every cut along a row or column are piecewise polynomials.

But how do k and p (the derivative order) control the solutions now? The result in this case is that there will be at most $n\,(n - 2p) + k$ non-zeros in the residual vector $\begin{bmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \end{bmatrix} \mathbf{x}_{L,k}$. This is not as "nice" as for the non-stacked results, but the main point still holds (for constant size images): Both the derivative order p and the parameter k controls the number of non-zeros in the residual.

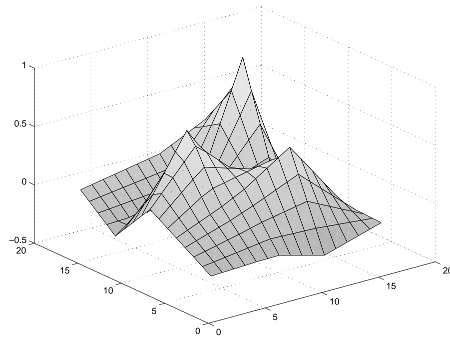
The stacked operators are also quite good when restoring images blurred with the "usual" blur model (from the test problem blur). The result of such a



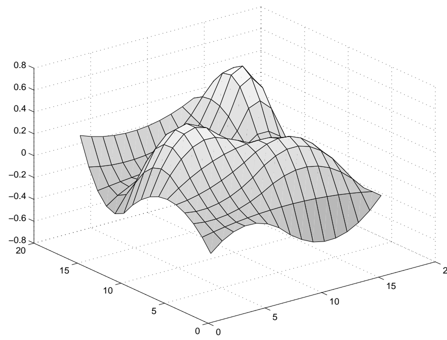
(a) Using the identity matrix as operator.



(b) Stacking the first order operator in vertical and horizontal direction.



(c) Stacking the second order operator in vertical and horizontal direction.



(d) Stacking the third order operator in vertical and horizontal direction.

Figure 5.7: The effects of stacking derivative operators in both horizontal and vertical direction.

reconstruction can be seen in figure 5.8. As it can be seen, the image is restored very well when $k = 120$. Best seen in figure 5.8(d) is also the fact, that the solution consists of constant coloured surfaces (as expected when using the first order derivative).

Other shapes can be seen reconstructed in figure 5.9. The method works quite well, but the best k value varies a lot depending on the complexity of the image.

Note that we have here only experimented with the first order derivatives. For some images it may be more profitable to use higher order derivatives. The method will easily handle this use too. How the results will be though, is another story...

5.2.4 Restoring images of stars

The previously presented results have all been conducted on synthetically made images. But how useful is the method applied to real-life problems? This question will not be fully answered in this report and many aspects still remain to be explored. We have, though, done a few tests on a star image.

The image was kindly made available to us from *Copenhagen University Astronomical Observatory*. It was very large, and in order to get results fairly quickly, we have used smaller excerpts from the original image[1].

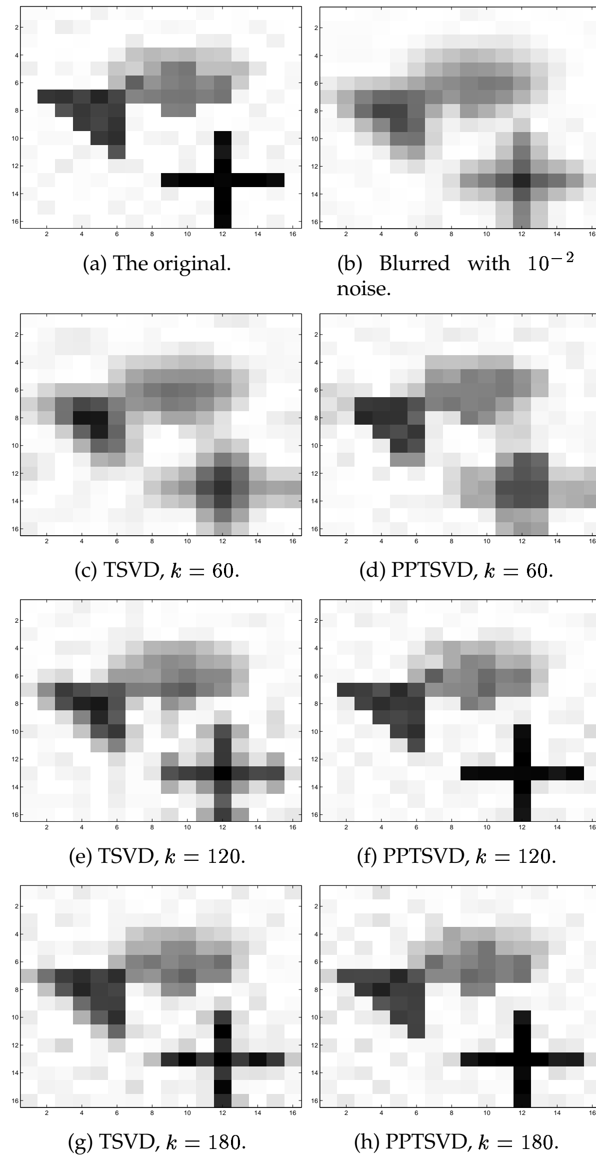
Assuming that each of the stars would appear as single points with an ideal camera, the operator should clearly be the identity matrix. What we did not have was the blurring model. This had to be made on the basis of the blurred image. By making a horizontal or vertical cut through one of the stars in the image, a blurring function could be approximated to fit these data. Assuming the function was normally distributed, only the variance had to estimated. This was easily done by minimizing the 2-norm of the difference between the estimated distribution function and the image data.

This approach assumes several things regarding the blurring model:

- That the blurring function is normally distributed.
- That the blurred star used to approximate the parameters come from just one point-like star, and not several stars lying close together.
- That the blurring is the same in all directions (the blurring parameters could have different values in horizontal and vertical direction).
- That the blurring is the same for each point in the image.

These problems could maybe be solved by having exact data on the physical system. This would probably lead to some complex model calculations, but could be worth considering.

[1]The image was originally 2050x2050 pixels. It was taken 9. of march 1997 with a 1.5 m telescope from an observatory located on the mountain La Silla in Chile.



(a) The original.

(b) Blurred with 10^{-2} noise.

(c) TSVD, $k = 60$.

(d) PPTSVD, $k = 60$.

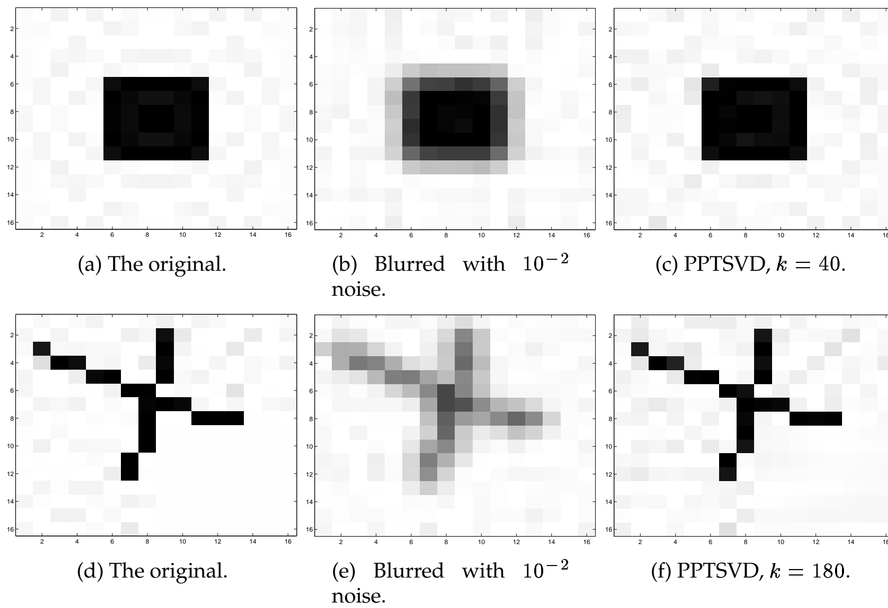
(e) TSVD, $k = 120$.

(f) PPTSVD, $k = 120$.

(g) TSVD, $k = 180$.

(h) PPTSVD, $k = 180$.

Figure 5.8: Restoring an image with stacked operators.



(a) The original. (b) Blurred with 10^{-2} noise. (c) PPTSVD, $k = 40$.

(d) The original. (e) Blurred with 10^{-2} noise. (f) PPTSVD, $k = 180$.

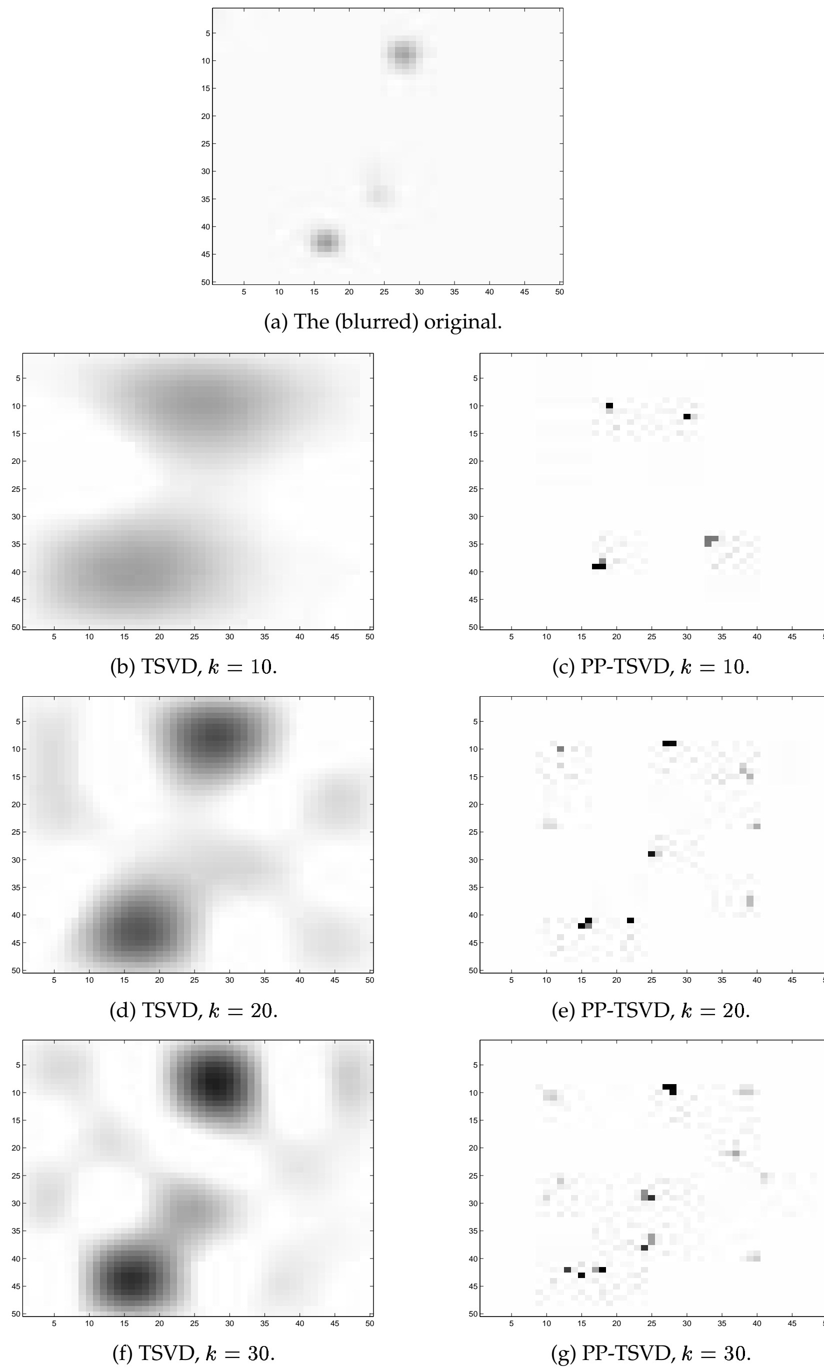
Figure 5.9: Showing the best reconstructions for two different images. Note how there is a big difference in the best k value.

Figure 5.10 shows an attempt to restore a 50×50 image of blurred stars. Along with the original (figure 5.10(a)) are shown 3 reconstructions with $k = 10, 20, 30$. The corresponding TSVD solutions are also shown to remind of one important thing: The PP-TSVD reconstructions are build solely on these. That is, if the TSVD image has dark areas, then this is where the PP-TSVD algorithm will try to emphasize the solution (according to the operator).

In figure 5.11, a 20×20 image has been "cut out" of the original and the figure again shows reconstructions for $k = 10, 20, 30$. This time, the dark spots are in the right area but they are spread out and not concentrated in a single dot (if that was what the "original" looked like!). Again, several gray dots are spread out in the image, due to the TSVD.

The results just presented are certainly not breathtaking, but some experience and pointers can be given in order to get the best results. It is important for the TSVD to be as accurate as possible. This can be accompliced by trying to restore only shapes in the middle of the image, and preferable using small images. These things will allow the TSVD to easier build the desired shape from the low frequency singular vectors. Weighing the high frequencies too much will result in spread out values throughout the image, and this will make the PP-TSVD restore the wrong things.

A generally important thing is to use a blurring model that is as exact as possible. As mentioned earlier this can be difficult, especially if the only infor-



(a) The (blurred) original.

(b) TSVD, $k = 10$.

(c) PP-TSVD, $k = 10$.

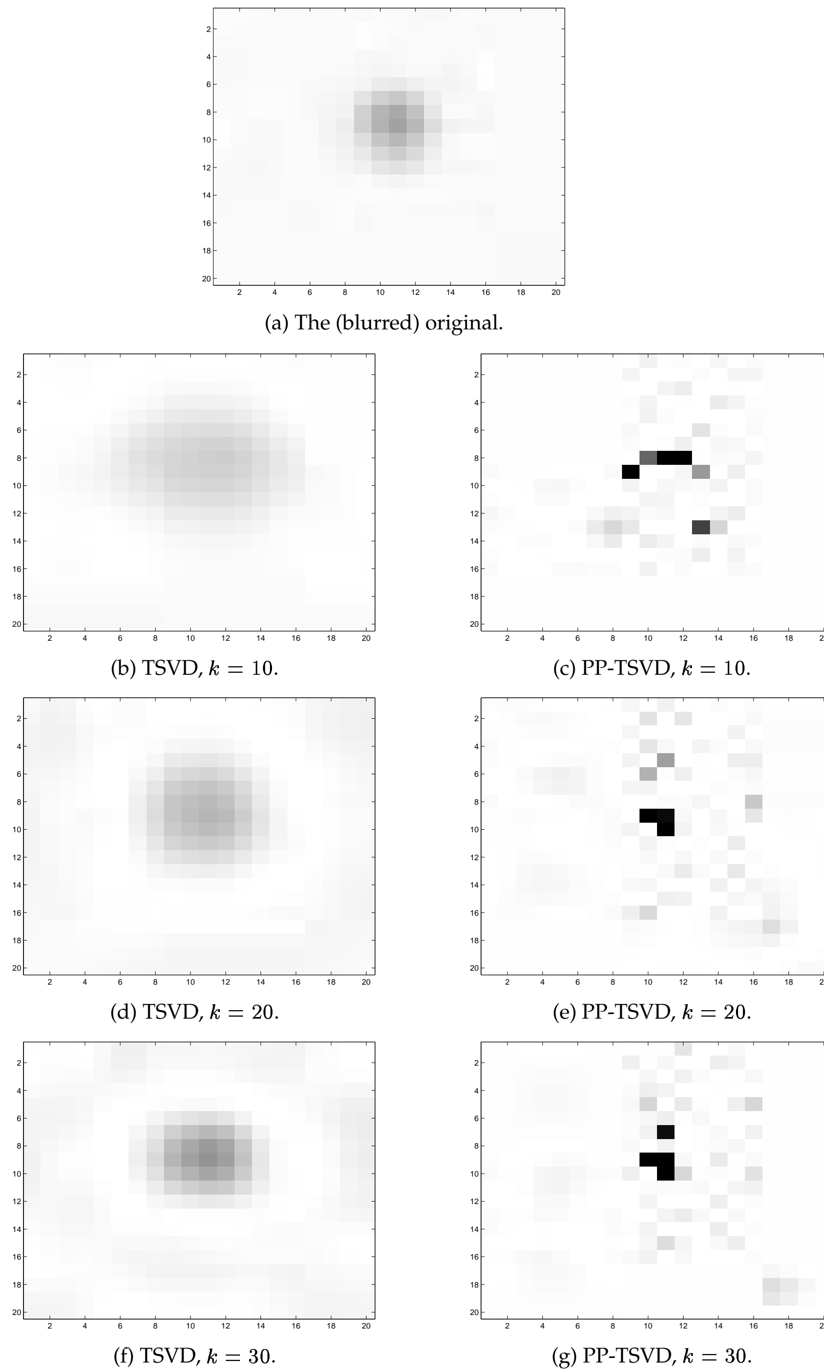
(d) TSVD, $k = 20$.

(e) PP-TSVD, $k = 20$.

(f) TSVD, $k = 30$.

(g) PP-TSVD, $k = 30$.

Figure 5.10: Attempting to restore a 50×50 star photo. Note that the graytones in the TSVD images have been amplified by 10 for a better view.



(a) The (blurred) original.

(b) TSVD, $k = 10$.

(c) PP-TSVD, $k = 10$.

(d) TSVD, $k = 20$.

(e) PP-TSVD, $k = 20$.

(f) TSVD, $k = 30$.

(g) PP-TSVD, $k = 30$.

Figure 5.11: Attempting to restore a single star in a 20×20 picture.

mation available is the blurred image. Several options are possible if there is a point-like object in the image that has been blurred (as is the case with star images):

- Make a horizontal (or vertical) cut through the blurred point and approximate a distribution function to these data (typically a normal distribution).
- Do both a horizontal and vertical cut and use the mean values of the two set of found parameters.
- Use the blurred point data *as they are* and use these as the blurring model. This way, by extracting e.g. a 5×5 submatrix (using a matrix point of view on images), then these data can be used to blur each pixel in a blurring model. This process could be done by our toolbox function patblur (see appendix A).

Only the first approach was used for the experiments shown in this section. Finally it must be mentioned, as stated earlier, that the k parameter is crucial for the (PP)TSVD method. Choosing the right value may just produce the (almost) perfect reconstruction. We did not try using $k > 30$ because of the time used to compute each solution.

Chapter 6

Conclusion

This report has focused on a method for solving certain inverse problems, ill-posed problems. For these kinds of problems, restoration is unstable and regularization had to be applied for the solution to be useful. The method we explored is called the PP-TSVD, Piecewise Polynomial Truncated Singular Value Decomposition. The method first finds a TSVD solution as a basis solution which has the disadvantage of containing only low frequency components. The PP-TSVD now restores the high frequency components in such a way, that some derivative of the solution has minimum 1-norm. This results in piecewise polynomial solutions.

This paper explained the theory behind the TSVD and showed how to derive the PP-TSVD. Due to a certain property of minimizing the 1-norm, the solutions will be piecewise polynomials. The PP-TSVD algorithm, initially described in [6], also had to be modified to be able to handle large problems. Because we had to experiment with 2-dimensional data, we showed how to use images in connection with the PP-TSVD. Images had to be represented as vectors, and derivative operators and blurring models had to work on such a representation too.

The PP-TSVD consists of a few basic computations/algorithms. All of them but one already existed in MATLAB. The missing algorithm was a way to solve a *linear constrained* l_1 problem. On the basis of some papers, we constructed an efficient MATLAB function to solve such a problem. This algorithm is completely generic, and could easily be used in other applications. Furthermore, a little MATLAB toolbox was developed to find PP-TSVD solutions and to compute derivative operators and blurring models for use with images.

An interesting chapter showed the outcome of our experiments with this method. A few 1-dimensional results were shown to illustrate some of the basic properties of the method. Specifically, it was shown how the different parameters of the PP-TSVD algorithm controlled the appearence of the solution. Then a number of application for the PP-TSVD for 2-dimensional data were

illustrated. The different restoration experiments included experiments with small synthetically made images and attempts to restore blurred star images taken with a telescope.

The synthetically made test images contained simple shapes to resolve what the method was able or unable to restore. Reconstructing images constisting of blurred dots was quite successful (when the noise level was not too influencial). Images applied with horizontal blur could also be restored when using the proper semi-norm operator.

Some of the most interesting results appeared when using stacked derivative operators. The solutions had an appearence very similar to those in the 1-dimensional case, that is, piecewise polynomials (for each row and column). This way, if the restored image should consist of constant coloured surfaces, then using the PP-TSVD with stacked operators could be the preferred method.

Common for the results with both horizontal blur and stacked operators was that the simple shapes where reconstructed quite easily, while more complex images where difficult (required carefully adjusted parameters).

Using the PP-TSVD on star images taken by telescope were not satisfactory. There could be a number of reasons for this, for which we gave some suggestions. This should not rule out using the method on such images however. The method can be fine tuned in several ways, and one way just might work.

Exploring the PP-TSVD method is far from over. The method can be used in numerous ways and may have some very good capabilities in some applications. A very large pool of $\mathbf{L}$ operators have not been tested and could prove useful. We have also focused solely on image blurring models and the PP-TSVD could very well prove good in other areas too.

Appendix A

PP-TSVD toolbox user guide

PPTSVD

Purpose Find the PP-TSVD solution and optionally the TSVD.

Synopsis

```
x = pptsvd(b,A,L,ks)
x = pptsvd(b,A,L,ks,svdfile)
[x,tsvd] = pptsvd(...)
```

Description `x = pptsvd(b,A,L,ks)` returns the PP-TSVD of `b` using the derivative operator `L` and `ks` singular values and vectors of `A`. `[x,tsvd] = pptsvd(b,A,L,ks)` returns the TSVD using `ks` singular values and vectors.

Arguments `pptsvd` takes 4 or 5 arguments and returns at most 2 values:

`b` The disturbed data.

`A` The model of the disturbance.

`L` A seminorm operator.

`ks` The wished number of used singular values and vectors. If `ks` is a vector the PP-TSVD (and TSVD if wished) is calculated for each element of `ks`.

`svdfile` A file containing an already calculated SVD for the given problem. If supplied the algorithm skips the calculation of the SVD.

`x` The data restored using the PP-TSVD method. If `ks` is a vector then each column contains a solution.

`tsvd` The data restored using the TSVD method. If `ks` is a vector then each column contains a solution.

Example

Using the test problem **wing** from the package REGULARIZATION TOOLS we generate a problem:

```
[A,b,x] = wing(128);
```

As figure A.1 shows the solution **x** is piecewise constant. Hence we make the 1. derivative using the function **get_l** (also from REGULARIZATION TOOLS):

```
L = get_l(128,1);
```

Now we are ready to perform the PP-TSVD:

```
[pptsvd,tsvd] = pptsvd(b,A,L,6);
```

A plot of **x**, **b**, **tsvd** and **pptsvd** can be seen in figure A.1

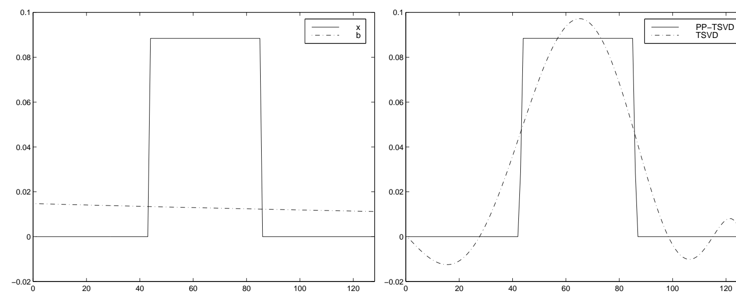


Figure A.1: Plots of **x**, **b**, **tsvd** and **pptsvd**.

Algorithm

Uses the algorithm found in the present report. Utilizes MATLAB's built in function **svds** to find the singular value decomposition.

See also

operator2d, **patblur**

References

Present report.
P. C. Hansen and K. Mosegaard, *Piecewise Polynomial Solution Without a priori Break Points, Num. Lin. Alg. with Appl*, Vol 3(6), 1996

L1C

Purpose Solve the discrete l_1 problem with linear constraints:

$$\begin{array}{rl} & \min \|\mathbf{b} - \mathbf{Ax}\|_1 \\ \text{subject to} & \begin{array}{c}\mathbf{Cx} = \mathbf{d} \\ \mathbf{Ex} \leq \mathbf{f}\end{array}\end{array} \tag{A.1}$$

Synopsis

```
x = l1c(A,b,C,d,E,f)
x = l1c(A,b,C,d,E,f,x0)
[x,r] = l1c(...)
```

Description `x = l1c(A,b,C,d,E,f)` solves the linear problem (A.1).
`x = l1c(A,b,C,d,E,f,x0)` solves the linear problem (A.1) using a starting guess.
`[x,r] = l1c(...)` returns a result status.

Arguments `l1c` accepts 6 or 7 arguments:

`A`, `b`, `C`, `d`, `E`, `f` see (A.1).

`x0` Specifies an initial guess for the algorithm. If none specified the default initial guess `x0`=$[00\ldots0]^{\mathrm{T}}$ is used.

`x` Contains the calculated result of (A.1) on return.

`r` Contains a status of the calculated result on return.

Example The example originates from fitting a straight line $f(t) = x_1 + x_2 t$ to the points $\{(1,2),(2,2),(3,3),(4,4),(5,3)\}$ with the constraints $f(6) = 5$ and $f(1) \leq 3$. This problem is described by the matrices

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 4 \\ 3 \end{bmatrix}$$

$$\mathbf{C} = [1\ 6] \qquad \mathbf{d} = [5]$$

$$\mathbf{E} = [1\ 1] \qquad \mathbf{f} = [3]$$

which the `l1c` routine solves with the solution:

```
[x,r]=l1c(A,b,C,d,E,f)
x = 1.0000 0.6667
r = 0
```

r=0 means that the routine has found an optimal solution.

Algorithm Transforms the problem into a linear programming problem and solves it using a modified simplex algorithm which takes advantage of the structure of this particular problem.

Diagnostics `l1c` returns a status in `r`:

`0`	Succes. Optimal solution found.
`1`	No feasible solution to the constraints.
`2`	Calculation stopped prematurely because of rounding errors.
`3`	Maximum number of iterations reached.

References

I. Barrodale and F. D. K. Roberts, *An efficient algorithm for discrete approximation with linear constraints*, SIAM J. Numer. Anal., vol. 15, No. 3, June 1978.

DISPVEC

Purpose Displays a quadratic image having vector representation.

Synopsis

```
dispvec(x)
dispvec(x,fignr)
```

Description Displays a quadratic image having vector form and values in range [0;1]
dispvec(x) displays **x** in current figure.
dispvec(x,fignr) displays **x** in figure **fignr**.

Arguments **dispvec** accept one or two arguments:

x A vector containing an image.

fignr The MATLAB figure in which to display **x**.

Example **dispvec([0 0.25 0.5 1]',2)** displays the picture with the matrix

$$\begin{bmatrix} 0 & 0.5 \\ 0.25 & 1 \end{bmatrix}$$

in figure 2.

Algorithm Reshapes the vector into quadratic form and maps the interaval [0;1] into [1;64]. Selects colormap gray.

Diagnostics If the image is not quadratic an error is displayed:
RESHAPE dimensions must be real integers

OPERATOR2D

Purpose Construct an operator to be used upon images represented by vectors.

Synopsis

```
L = operator2d(P,m)
L = operator2d(P,m,n)
```

Description **`L = operator2d(P,m)`** constructs an operator for quadratic image of size $m \times m$.
`L = operator2d(P,m,n)` constructs an operator for an image of size $m \times n$.

Arguments **`operator2d`** accepts two or three arguments:

- **`P`** Operator pattern.
- **`m`** Image size. If supplied without **`n`** the the result is meant to be used upon images of size $m \times m$.
- **`n`** Image size. The operator is to be used with images of size $m \times n$.
- **`L`** The resulting operator which can be multiplied in front of a image vector of size mn.

Examples **`L = operator2d([1 -1],16)`** makes the 1. order derivvative along each row for a 16×16 image.
`L = operator2d([1 -1]',256)` makes the 1. order derivative along the row of a vector of size 256.
`L = operator2d([-1 2 -1]',32)` makes the 2. order derivative along each column for a 32×32 image.
`L = operator2d([0 -1 0; -1 4 -1;0 -1 0],32)` makes a 2. order edge detection filter for an image of size 32×32.

Algorithm Distributes the elements of the pattern into **`L`** relative to the position of a pixel in the matrix and vector representation respectively.

References Present report section 3.3.2.

PATBLUR

Purpose Make custom blurring operators for pictures having vector representation. May also be used to create edge detection operators.

Synopsis

```
A = patblur(P,cy,cx,m)
A = patblur(P,cy,cx,m,n)
```

Description **`A = patblur(P,cy,cx,m)`** makes operator with pattern **`P`** and operator center in row **`cy`** and column **`cx`** for use upon a square images of size $m \times m$.
`A = patblur(P,cy,cx,m,n)` makes operator for use upon images of size $m \times n$.

Arguments **`patblur`** accepts 4 og 5 arguments and returns one variable:

- **`P`** Operator pattern.
- **`cy`**,**`cx`** Center pixel coordinates for pattern.
- **`m`** Number of rows in target images.
- **`n`** Number of columns in target images. If not given its equal to **`m.`**
- **`A`** The operator which can be applied on image vectors of size mn.

Examples **`A = patblur([1 1 1 1 1]/5,1,3,16)`** gives a very simple horizontal blurring model for use upon 16×16 images.
`A = patblur([1 1 1;1 1 1;1 1 1]/9,2,2,16,32)` gives a simple blurring model (in all directions) for 16×32 images.

Algorithm Makes a convolution operator using the pattern **`P`** and its center. Elements "outside" the targeted image size are set to zero.

See Also **`filter2`** (standard matlab function)**`, operator2d`**

APPENDIX B

Source code

pptsvd.m

```
function [ppt,t] = pptsvd(b,A,L,ks,svdfile)
% PPTSVD   Piecewise Polynomial Truncated SVD
%
% PPT = PPTSVD(B,A,L,KS) returns a PP-TSVD solution of B. B is the
% disturbed data (e.g. a picture) in the form of a vector. A is a
% model of the disturbance and L is e.g. an approximation to a derivative
% operator. KS denotes the number of singular values used in the
% computations. KS can be a vector with several choices of k.
%
% [PPT,T] = PPTSVD(B,A,L,KS) in addition returns the ordinary truncated
% singular value decompositions (TSVD) in T.
%
% For each element in KS a column in PPT and T is made containing the
% result.

% Last revised 17.06.1998

% Find maximum number of singular values and vectors to use
maxk = max(ks);

% Calculate the SVD (or load)
if nargin == 4
  disp('Finding SVD...');
  [U,S,V] = mysvds(A,maxk);
else
  disp('Loading SVD...');
  load(svdfile);
end

% Allocate space for return variables
ppt = zeros(size(b,1),length(ks));
t = ppt;

z = zeros(size(b));
```

```
for i = 1:length(ks)
  % Which k to use
  k   = ks(i);

  % Display progress information
  disp(sprintf('Making solution for k=%d...',k));

  % Calculate TSVD
  x_k = V(:,1:k)*diag(1./diag(S(1:k,1:k)))*(U(:,1:k)'*b);

  % Calculate l1 problem
  [z,res] = l1c(L, L*x_k, V(:,1:k)', zeros(k,1), zeros(0, size(z,1)), ...
      zeros(0,1), z);
  % In case of l1 failure print out warning
  if res ~= 0
    disp(sprintf('Warning: l1c returned %d',res));
  end

  % Calculate PP-TSVD and insert result in return variables
  ppt(:,i) = x_k - z;
  t(:,i) = x_k;
end

% Now we are done...
disp('Done.');
```

l1c.m

```
function [x,result] = l1c(A,b,C,d,E,f,x0)
% L1C Solve the discrete l1 linear approximation with linear
%     constraints.
%
% [x,result] = L1C(A,b,C,d,E,f,x0) solves the l1 problem with liniar
% constraints. Parameters according (1).
%
% Solves a problem of the form:
%
% (1)           min||b-Ax||_1, Cx=d, Ex<=f
%
% Transforms the problem (1) into a simple linear problem and solves using a
% simplex based algorithm.
%
% Arguments A,B,C,D,E,F are explained by (1)
%
% X0 is a starting guess which in some cases can speed up calculations.
%
% X returns the calculated solution to (1)
%
% result returns an staus value:
%    0    : Optimal solution found
%    1    : No feasible solution to the constraints
%    2    : Calculations stopped prematurely because of rounding errors
%    3    : Maximum number of iterations reached
%
%
% Based upon the algorithm described in [1]
%
% [1] I. Barrodale and F. D. K. Roberts: An efficient algorithm
% for discrete approximation with linear constraints.
% SIAM J. Numer. Anal., vol. 15,  No. 3, June 1978
%

% Last revised 17.06.1998


[m,n] = size(A);
  k   = size(C,1);
  l   = size(E,1);

% Adjust problem if having a starting guess
if nargin == 7
  b = b-A*x0;
  d = d-C*x0;
  f = f-E*x0;
end

toler = 10^(-16*2/3);
maxiter = 10*(m+k+l);
x = zeros(n,1);
result = -1;
mkl1 = m+k+l+1;
iter = 0;
```

```
wn = n;
kforce = 1;

% Setup Q

Q = full([ A b ;
           C d ;
   E f ;
   zeros(1,n+1) ]);

inbasis = n+(1:m+k+l);
insign  = ones(1,m+k+l);
outbasis = 1:n;
outsign  = ones(1,n);

index = find(Q(1:m+k+l,n+1) < 0);
insign(index) = ~insign(index);
Q(index,:) = -Q(index,:);

% Phase 1

% Setup phase 1 costs

cu = zeros(2,n+m+k+l);
cu(:,n+m+1:n+m+k) = 1;
if l > 0
  cu(2,n+m+k+1:n+m+k+l) = 1;
end
iu = cu;

% Compute marginal costs

t = (cu(1,inbasis) .* insign + cu(2,inbasis) .* ~insign);
Q(mkl1,:) = t*Q(1:m+k+l,:);

while 1,

  % Vector to enter basis

  zu = Q(mkl1,1:wn);
  zv = -zu - sum(cu(:,outbasis));

  s = outsign;
  i1 = ~iu(1,outbasis);
  i2 = ~iu(2,outbasis);

  zu = zu .* ((s & i1) | (~s & i2));
  zv = zv .* ((s & i2) | (~s & i1));

  if kforce == 1
    t = (outbasis <= n);
    [val ,index ] = max(zu .* t);
    [val2,index2] = max(zv .* t);

    if (isempty(val) & isempty(val2)) | ((val < toler) & (val2 < toler))
      if Q(mkl1,wn+1) < toler
```

```
      break;
    else
      kforce = 0;
    end
  end
end
if kforce == 0
  [val ,index ] = max(zu);
  [val2,index2] = max(zv);

  if (isempty(val) & isempty(val2)) | ((val < toler) & (val2 < toler))
    break;
  end
end

if val2 > val
  in = index2;
  Q(:,in) = -Q(:,in);
  Q(mkl1,in) = val2;
  outsign(in) = ~outsign(in);
else
  in = index;
end

index = find(Q(1:m+k+l,in) > toler);

while 1,

  if isempty(index)
    result = -2;   % Go to phase 2
    break;
  end

  [val,i] = min(Q(index,wn+1) ./ Q(index,in));
  out = index(i);
  index(i) = index(end);
  index = index(1:end-1);

  pivot = Q(out,in);

  i = inbasis(out);
  cuv = cu(1,i) + cu(2,i);

  if Q(mkl1,in) - cuv*pivot > toler
    Q(mkl1,:) = Q(mkl1,:) - cuv*Q(out,:);
    Q(out,:) = -Q(out,:);
    insign(out) = ~insign(out);
  else
    break;
  end

end

if result == -2
  result = -1;
  break;
```

```
  end;

  iter = iter + 1;

  Q(out,:) = Q(out,:)/pivot;

  RO = speye(mkl1);
  RO(:,out) = -Q(:,in);
  RO(out,out) = 1;

  Q(:,in) = 0;
  Q(out,in) = 1/pivot;

  Q = RO*Q;

  val = insign(out);
  insign(out) = outsign(in);
  outsign(in) = val;
  val = inbasis(out);
  inbasis(out) = outbasis(in);
  outbasis(in) = val;

  if iu(1,val) == 1 & iu(2,val) == 1
    Q(:,in) = Q(:,1);
    Q = Q(:,2:end);
    outbasis(in) = outbasis(1);
    outbasis = outbasis(2:end);
    outsign(in) = outsign(1);
    outsign = outsign(2:end);
    wn = wn - 1;
  end
end

if result == -1

  if Q(mkl1,wn+1) >= toler
    result = 1;
    return;
  end

  % Phase 2
  % Setup phase 2 costs

  cu = zeros(2,n+m+k+1);
  cu(:,n+1:n+m) = 1;
  index  = find(insign==1 & iu(1,inbasis)==1);
  index2 = find(insign==0 & iu(2,inbasis)==1);
  if ~isempty(index)
    cu(1,inbasis(index )) = 0;
  end
  if ~isempty(index2)
    cu(2,inbasis(index2)) = 0;
  end

  t = zeros(1,m+k+1);
  t(index)  = 1;
```

```
t(index2) = 1;

[s,index]  = find(t);
[s,index2] = find(~t);

ia = length(index);

Q = [Q(index,:); Q(index2,:); Q(mkl1,:)];
inbasis = [inbasis(index) inbasis(index2)];
insign  = [ insign(index)  insign(index2)];

% Compute marginal costs

t = (cu(1,inbasis) .* insign + cu(2,inbasis) .* ~insign);
Q(mkl1,:) = t*Q(1:m+k+l,:);

while iter <= maxiter,

  % Vector to enter basis

  zu = Q(mkl1,1:wn);
  zv = -zu - sum(cu(:,outbasis));

  s = outsign;
  i1 = ~iu(1,outbasis);
  i2 = ~iu(2,outbasis);

  zu = zu .* ((s & i1) | (~s & i2));
  zv = zv .* ((s & i2) | (~s & i1));

  [val ,index ] = max(zu);
  [val2,index2] = max(zv);

  if (isempty(val) & isempty(val2)) | ((val < toler) & (val2 < toler))
    result = 0;
    break;
  end

  if val2 > val
    in = index2;
    Q(:,in) = -Q(:,in);
    Q(mkl1,in) = val2;
    outsign(in) = ~outsign(in);
  else
    in = index;
  end

  [val,out] = max(abs(Q(1:ia,in)));

  if (~isempty(val)) & (val > toler)
    Q([out ia],:) = Q([ia out],:);
    inbasis([out ia]) = inbasis([ia out]);
    insign([out ia])  = insign([ia out]);
    out = ia;
    ia = ia - 1;
    pivot = Q(out,in);
```

```
else
  index = find(Q(1:m+k+l,in) > toler);

  while 1,

    if isempty(index)
      result = 2;
      break;
    end

    [val,i] = min(Q(index,wn+1) ./ Q(index,in));
    out = index(i);
    index(i) = index(end);
    index = index(1:end-1);

    pivot = Q(out,in);

    i = inbasis(out);
    cuv = cu(1,i) + cu(2,i);

    if ((insign(out) == 0 & iu(1,i) == 0) | (insign(out) == 1 & ...
  iu(2,i) == 0)) & (Q(mkl1,in) - cuv*pivot > toler)
      Q(mkl1,:) = Q(mkl1,:) - cuv*Q(out,:);
      Q(out,:) = -Q(out,:);
      insign(out) = ~insign(out);
    else
      break;
    end
  end
end

if result == 2
  break;
end

iter = iter + 1;

Q(out,:) = Q(out,:)/pivot;

RO = speye(mkl1);
RO(:,out) = -Q(:,in);
RO(out,out) = 1;

Q(:,in) = 0;
Q(out,in) = 1/pivot;

Q = RO*Q;

val = insign(out);
insign(out) = outsign(in);
outsign(in) = val;
val = inbasis(out);
inbasis(out) = outbasis(in);
outbasis(in) = val;

if iu(1,val) == 1 & iu(2,val) == 1
```

```
      Q(:,in) = Q(:,1);
      Q = Q(:,2:end);
      outbasis(in) = outbasis(1);
      outbasis = outbasis(2:end);
      outsign(in) = outsign(1);
      outsign = outsign(2:end);
      wn = wn - 1;
    end
  end
end

if iter > maxiter
  result = 3;
elseif result == 0 | result == 2
  index = find(inbasis <= n & insign == 1);
  x(inbasis(index)) = Q(index,wn+1);
  index = find(inbasis <= n & insign == 0);
  x(inbasis(index)) = -Q(index,wn+1);
end

% Adjust result if a starting guess was given.
if nargin == 7
  x = x + x0;
end
```

dispvec.m

```
function dispvec(A,fignr)
% DISPVEC Display a vector as a quadratic picture
%
%   DISPVEC(A,fignr)
%
%   A is a vector containing a picture. The graytones of the
%   picture is scaled from [0;1] to [1;64].
%
%   If FIGNR is suplied the picture is displayed in figure
%   FIGNR. If not supplied the picture is displayed in the
%   the current figure
%
%

% Last revised 17.06.1998

% Change fignr if supplied
if nargin == 2
  figure(fignr);
end

% Calculate sidelength of pic
n = sqrt(length(A));

% Scale [0;1] into [1;64] and display using gray colormap
image(1+63*reshape(A,n,n));
colormap(gray);
```

operator2d.m

```
function L = operator2d(P, m, n)
% L = OPERATOR2D(P,M,N)
%
% Generates an L matrix using to be used upon a picture of size MxN. The
% type of L matrix is determined by the vector P. If n is not supplied the
% picture is by default quadratic and N=M.
%
% Examples:
% L11 = OPERATOR2D([1 -1],16);    % Make 1. order row derivative for 16x16
% L12 = OPERATOR2D([1 -1]',16);   % Make 1. order column derivative
% L21 = OPERATOR2D([-1 2 -1],16); % Make 2. order derivative row
%
% Make edge detection filter of 2. order
% Ledge = OPERATOR2D([ 0 -1  0 ;
%                     -1  4 -1 ;
%                      0 -1  0 ],100);
%

% Last revised 17.06.98

% Quadratic image
if nargin == 2
  n = m;
end

% Find size of operator profile
[pm,pn] = size(P);

% Generate operator submatrix
C = spalloc(m,pn,pm*pn);
C(1:pm,1:pn) = P;
C = reshape(C, 1, m*pn);
D = toeplitz([C(1); zeros(m-pm,1)], C);

% Allocate space for sparse L operator
L = spalloc((m-pm+1)*(n-pn+1), m*n, (m-pm+1)*pm*pn);

% Fill in operator
for i = 0:n-pn,
  L(1+i*(m-pm+1)+[0:m-pm], m*i+[1:m*pn]) = D;
end
```

patblur.m

```
function A = patblur(P,cy,cx,m,n)
% PATBLUR Make an operator for images represented as vectors. Initialy made
% to make bluring operators.
%
% A = PATBLUR(P,CY,CX,M,N) return a operator made with profile P with center
% in row CY column CX. Operator can be used upon images of size MxN
% (represented by a vector thoug). If N is not given it is set equal to M.
%
% Examples:
% A = PATBLUR([1 1 1 1 1]/5,1,3,16) makes a very simple horizontal blur with
% bandwidth 2 for use with images of size 16x16
%
% A = PATBLUR([0 0 1 0 0;
%              1 1 1 1 1;
%              0 0 1 0 0]/7,2,3,16,32) makes a combined horizontal and vertical
% blur for use with images of size 16x32. The horizontal with bandwidth 3
% and the vertical with bandwidth 2.
%
% A = PATBLUR([1 2 4 2 1]/10,1,3,16) makes a rough estimation of a Gaussian
% blur for images of size 16x16
%
% For use on images having matrix representation use FILTER2
%
% See also OPERATOR2D, FILTER2

% Last revised 17.06.1998

if nargin == 4
  n = m;
end

[pm,pn] = size(P);

A = sparse(n*m, n*m);

for i=1:pn,
  B = spdiags(ones(m,1)*P(:,i)', [1:pm]-cy, m, m);
  C = spdiags(ones(n,1), i-cx, n, n);
  A = A + kron(C,B);
end
```

APPENDIX C

Internal tests

`l1c.m`

As specified in section 4.1, the linear constrained l_1-problem can be written as a linear programming problem. Using the MATLAB function `lp` from a package called OPTIMIZATION TOOLBOX, a testscript could be written to verify that our `l1c` algorithm worked correctly.

The `lp` function could solve problems of the following sort:

$$\min_{\mathbf{x}} \mathbf{w}^T\mathbf{x} \quad \text{subject to} \quad \mathbf{G}\mathbf{x} \leq \mathbf{h}$$

- but any of the inequality constraints could be specified as equality constraints. Moreover, an initial feasible solution was not required. This way, the LP-problem shown in equation 4.2 (page 22) could be fed directly into the `lp` function:

$$\begin{aligned}
\mathbf{w}^T &= [\overbrace{0 \cdots 0}^{2n} \overbrace{1 \cdots 1}^{2m} \overbrace{0 \cdots 0}^{l}] \\
\mathbf{G} &= \begin{bmatrix} \mathbf{A} & -\mathbf{A} & \mathbf{I}_{m\times m} & -\mathbf{I}_{m\times m} & \mathbf{0} \\ \mathbf{C} & -\mathbf{C} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{E} & -\mathbf{E} & \mathbf{0} & \mathbf{0} & \mathbf{I}_{l\times l} \end{bmatrix} \\
\mathbf{h} &= \begin{bmatrix} \mathbf{b} \\ \mathbf{d} \\ \mathbf{f} \end{bmatrix}
\end{aligned}$$

Of course, all the constraints here have to be equality constraints. Now, a script was made that ran the following steps repeatedly:

- *Generate random matrices* Random size matrices were generated and filled with random numbers. Moreover, with a certain possibility, each of the

matrices could be made to consist of only zeros, negative, positive or all numbers.

- *Solve using lp* By setting up the matrices $\mathbf{w}^T$, $\mathbf{G}$ and $\mathbf{h}$ as described above, the lp function was called to obtain a solution.
- *Solve using l1c* The l1c algorithm was called to get another (or the same) solution.
- *Compare the results* As there are often an infinity of solutions, the solutions were not always the same, but the following could be checked: That the solutions fulfilled the constraints and check that the found minimum values were the same. If this was true, then the l1c was considered to work correctly (for the used dataset).
  If the constraints were impossible to fulfill, then both algorithms should of course reflect this.

By letting this script run for several hours without any "disagreements" it was concluded that our l1c algorithm worked correctly.

Bibliography

[1] I. Barrodale and F. D. K. Roberts, *An Improved Algorithm for Discrete l_1 Linear Approximation*, SIAM J. Numer. Anal., Vol 10 (5), 1973

[2] I. Barrodale and F. D. K. Roberts, *An Efficient Algorithm for Discrete l_1 Linear Approximation with Linear constraints*, SIAM J. Numer. Anal., Vol 15 (3), 1978

[3] S. I. Gass, *Linear Programming*, 4th ed., McGraw-Hill, 1975

[4] P. C. Hansen, *Numerisk behandling af Fredholm-integralligninger af første art*, UNI-C, DTU, 1991

[5] P. C. Hansen, *A Matlab Package for Analysis and Solution of Discrete Ill-Posed Problems*, UNI-C, Lyngby, 1993

[6] P. C. Hansen and K. Mosegaard, *Piecewise Polynomial Solution Without a priori Break Points, Num. Lin. Alg. with Appl*, Vol 3(6), 1996

[7] P. C. Hansen, *Rank-Deficient and Discrete Ill-Posed Problems: Numerical Aspects of Linear Inversion*, SIAM, Philadelphia, 1998

[8] V. L. Hansen, *Grundbegreber i den moderne analyse*, MAT, DTU, 1995

[9] H. B. Nielsen, *Numerisk Lineær Algebra*, IMM, DTU, 1996

[10] G. A. Watson, *Approximation Theory and Numerical Methods*, Wiley, Chichester, 1980.