OPTIMIZATION IN ℓ_1- NORM FOR SPARSE RECOVERY

by

THIEN THANH LAM

THESIS

for the degree of

MASTER OF SCIENCE

(Master i Anvendt matematikk og mekanikk)

Faculty of Mathematics and Natural Sciences
University of Oslo

May 2014

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

ACKNOWLEDGEMENT

First of all, I would like to thank my supervisor Geir Dahl for his good advice and guidance. Thank you for constantly keeping me on the right direction and for supporting me during the work of this thesis. I express my deepest gratitude to you.

I would also like to thank Øyvind Ryan for helping me with the codes. Thank you for always being helpful and available for questions and discussions.

Finally, I would like to share my joy with my family and all friends. Thanks for providing me with love, caring and encouragement through my working period with the thesis. This has given me alot of motivation. A special appreciation to my boyfriend, Kai Yao for always believing in me.

Thien Thanh Lam
Oslo, May 2014

ABSTRACT

This paper is about solving an optimization problem for a sparse solution. Given a matrix A and a vector b, the optimization problem is to solve the linear equation $Ax = b$ for x. The problem can be represented as a minimization problem where we minimize the norm of x subject to $Ax = b$. By using $\|\cdot\|_0$, defined as the number of non-zero components, the problem becomes an NP-hard problem. Therefore we do a convex relaxation by using the ℓ_1- norm. With this relaxation we are able to find a sparse solution, a solution with few non-zero entries. The advantage with sparse vectors is that the computations take less time. Another advantage is that sparse vectors require less space when being stored, since we only need to know the position and the value of the entries.

The problem is used in Compressed Sensing and in many other applications. Compressed Sensing is about representing signals by using few non-zero coefficients in a basis. For the system we are solving to have sparse recovery, it requires conditions such as the Null Space Condition, the Spark, the Restricted Isometry Property and the Coherence. There are many methods for solving this problem, and in this paper Basis Pursuit and Greedy algorithms will be presented.

CONTENTS

1. INTRODUCTION

The main purpose of this paper is to solve a sparse approximation problem. Given a matrix A and a vector b, we are estimating a sparse vector x that satisfies a linear system of equations of A and b. By using the ℓ_1- norm, we are able to find a sparse solution which has many advantages. Sparse approximation is used in many areas. It is used in regularization and compression of signals and images. There are many methods for solving the problem under important conditions.

In this paper we start by presenting the background theory needed for the algorithms for solving an optimization problem for a sparse solution in Chapter **2**. The chapter is based on the book *Convex Optimization* written by S. Boyd and L. Vandenberghe, [12]. Our minimization problem in ℓ_1-norm has its applications in many areas. In Chapter **3** we present how it is used in image denoising and inpainting, [7, 8], cryptography [1] and traffic monitoring. It can be used as a technique for removing the noise an image has been disrupted with or for reconstructing parts of an image which have been lost. It can help us finding back the *plaintext* by a given *ciphertext*. It can be used for reducing the storage of some data.

In Chapter **4**, Compressed Sensing is presented. We explain the difference between bases and frames, and how we regularize the underdetermined system so that there exist sparse solutions. First we use ℓ_2- norm, but unfortunately the solution is not sparse. With $\|\cdot\|_0$ the solutions are sparse, but if the system is large, then finding the solution would be computationally intractable. Finally we regularize the problem with ℓ_1- norm. It is then able to find sparse solutions and yet remains computationally tractable. We present the Null Space Condition, the connection between the Spark and the rank of the matrix Φ, the Restricted Isometry Property and the Coherence. These are conditions required for our problem to have sparse recovery. This chapter is mainly based on the book *Compressed Sensing* written by Y. C. Eldar and G. Kutyniok, [13].

The regularized minimization problem in ℓ_1- norm is known as Basis Pursuit. It is a convex optimization problem and can be recast as a linear programming problem. In Chapter **5** we present algorithms which can solve the problems involving the ℓ_1- norm. They are Simplex method and the Interior point methods, [11, 12]. We also give a description of other methods, Orthogonal Matching Pursuit [13], Stagewise Orthogonal Matching Pursuit [2], Regularized Orthogonal Matching Pursuit [3], Compressive Sampling Matching Pursuit [13] and Iterative Hard Thresholding Algorithm [13]. These algorithms have many similarities with each other. They are greedy algorithms.

We solve Basis Pursuit in Chapter **6**. We show how Basis Pursuit can be recast as a linear programming problem, and solve it first by using the command *linprog* in Matlab [10] and then by the primal-dual interior point method [4] presented in Section **6.3**. Finally we test the codes on a coin example [6] and in image processing, [14, 15]. The results and discussions are in Section **6.4**, **6.5** and **6.6**.

2. BACKGROUND

In this chapter we start by giving a short presentation of what an optimization problem is and also its applications. We will tell shortly about the two subclasses of convex optimization, least-squares problem and linear programming problem. Finally we end the chapter with approximation. This chapter is mainly based on the book *Convex Optimization* written by S. Boyd and L. Vandenberghe, [12].

2.1 Optimization Problem

Optimization problem is a problem where our goal is to find the best solution from all feasible solutions. It consists of minimizing or maximizing a function by selecting values within an allowed set. The optimization problem has the form

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq b_i, \quad i = 1, \dots, m \\ & h_i(x) = 0, \quad i = 1, \dots, p \end{array}$$

Here we are minimizing an objective function $f_0 : \mathbb{R}^n \to \mathbb{R}$. The vector x is the optimization variable. The problem has inequality constraints which correspond to the functions $f_i : \mathbb{R}^n \to \mathbb{R}$, $i = 1, \dots, m$, with the constants $b_1, \dots, b_m$ as bounds. The problem also has equality constraints which correspond to the functions $h_i : \mathbb{R}^n \to \mathbb{R},\ i = 1, \ldots, p$. If there are no constraints, then the problem is unconstrained.

A point x is a feasible point if it is in the domain of the problem, that is if it is in

$$D = \textstyle\bigcap_{i=0}^{m} \textbf{dom}\, f_i \;\cap\; \bigcap_{i=1}^{p} \textbf{dom}\, h_i$$

and that it satisfies the constraints $f_1(x) \le b_1, \ldots, f_m(x) \le b_m$ and $h_1(x) = 0, \ldots, h_p(x) = 0$. The domain, denoted as **dom**, is defined as the subset of $\mathbb{R}^n$ of points x for which *f(x)* and *h(x)* are defined. A point $x^\star$ is an optimal solution if it is feasible and gives us the smallest objective function value $f_0(x^\star)$ among all others. If $f_0(x^\star) = \infty$, then the problem is infeasible. This means that no solution satisfies the constraints. If $f_0(x^\star) = -\infty$, then the problem is unbounded below. This means that there are feasible points x_k with $f_0(x_k) \to -\infty$ as $k \to \infty$.

2.1.1 **Applications**

We have a set of points x which can be seen as choices. The choices can be *possible* or *impossible*. To separate these choices, we need constraints. With the constraint functions $f(x)$ and $h(x)$, we can make a set of *possible* choices x. Among these *possible* choices x we can then choose out the best one $x^\star$, and the objective value $f_0(x^\star)$ is the cost for have taken this choice.

In economics the optimization problem is used, for example when we calculate our budget. We try to minimize the cost we use for buying food or other things for each month. The optimization variable x corresponds to what kind of things we use our budget for. Then we have limits for how much we are going to spend on it, and this correspond to the constraint functions. Finally the objective function will give us the lowest total cost of the choices we have made.

Optimization problem is also used in mechanics and physics. We calculate how much force a machine uses, and we want to minimize this force. The optimization variable x corresponds to the different parameters which affect the force. The adjustment and the limits of the parameters correspond to the constraint functions $f(x)$ and $h(x)$. After we have found the optimal solution $x^\star$, the objective function $f_0(x^\star)$ will give us the total lowest force for the parameters we have chosen.

Optimization problem is very important and it is used in many areas. It is used in economics, engineering, network and it is also for daily use like for scheduling and planning. The list of applications is still expanding.

2.1.2 Nonlinear Optimization

Nonlinear optimization is an optimization problem where the objective function or the constraint functions are not linear. There are different approaches for solving a nonlinear optimization problem.

In local optimization, we are looking for a point which is locally optimal. This point minimizes the objective function among feasible points that are near it. A point x is locally optimal if there is an $R > 0$ such that x is optimal for

$$\begin{array}{lll} \text{minimize} & f_0(z) & \\ \text{subject to} & f_i(z) \leq 0, & i = 1, \dots, m \\ & h_i(z) = 0, & i = 1, \dots, p \\ & \|z - x\|_2 \leq R & \end{array}$$

The methods are fast and can handle large-scale problems, because they only require that the objective function and the constraint functions to be differentiable. The disadvantage with local optimization is that the methods require an initial guess for the optimization variable, and this guess can greatly affect the objective value.

We have global optimization which is used for problems that have small number of variables. Then we can find the global solution of the optimization problem.

2.2 Convex Optimization

A convex optimization problem is an optimization problem where the objective function and the constraint functions are convex. So the functions $f_0, \dots, f_m$ satisfy

$$f_i(\alpha x + \beta y) \leq \alpha f_i(x) + \beta f_i(y)$$

for all $x, y \in \mathbb{R}^n$ and all $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1$, $\alpha \geq 0$, $\beta \geq 0$. The functions $h_0, \dots, h_p$ are affine. A function is affine if it is a sum of a linear function and a constant. The equality constraint functions $h_i(x)$ can be written as $a_i^T x - b_i$. An important property of the convex optimization problem is that the feasible set is convex.

2.2.1 Subclasses of Convex Optimization

In this section we are going to present two very well known subclasses of convex optimization. They are least-squares problem and linear programming problem.

Least-Squares Problem

A least-squares problem is an optimization problem where there are no constraints. The solution x is minimizing the Euclidean norm of the residual vector $r = Ax - b$,

$$\text{minimize } \|r\|_2^2 = \|Ax - b\|_2^2 = \textstyle\sum_{i=1}^{p}(a_i^T x - b_i)^2$$

a_i^T are the rows of the matrix $A \in \mathbb{R}^{p\times n}$, where $p \geq n$. The vector $x \in \mathbb{R}^n$ is the optimization variable. The least-squares problem can be solved analytical,

$$\begin{aligned}
&Ax = b\\
&(A^T A)x = A^T b\\
&x = (A^T A)^{-1} A^T b
\end{aligned}$$

To know that an optimization problem is a least-squares problem, we need to check that the objective function is a quadratic function. There are many good algorithms for solving least-squares problems. If the problem is large, we can exploit the structure of the matrix A. For example, if the matrix A is sparse, which means that it has few non-zero entries, then we can solve the least-squares problem much faster. Now we present two techniques of the least-squares problem.

In a weighted least-squares problem we have weights $w_1, \dots, w_p$ in the objective function. The weights are positive, and they are chosen to reflect the different sizes of the terms $a_i^T x - b_i$. The weighted least-squares problem has the form

$$\textstyle\sum_{i=1}^{p} w_i (a_i^T x - b_i)^2$$

Weighted least-squares is a good method for problems with few variables. The advantage with this method is that it can handle regression. The disadvantage with it is that the weights have to be known exactly. In mostly applications they are almost never known, so instead estimated weights are used.

Regularization is another technique in least-squares. A regularized least-squares problem is also an optimization problem without constraints. In regularization extra terms are added to the objective function for penalizing the large values of x,

$$\textstyle\sum_{i=1}^{p} (a_i^T x - b_i)^2 + \lambda \sum_{i=1}^{n} x_i^2$$

The parameter $\lambda > 0$ is chosen by the user. Our goal here is to make the objective function $\sum_{i=1}^{p} (a_i^T x - b_i)^2$ small and at the same time keeping $\sum_{i=1}^{n} x_i^2$ small. Later in this chapter, we will see how regularization is used in approximation.

Linear Programming

A linear programming problem is an optimization problem where the objective function and the constraint functions are linear.

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & a_i^T x \leq b_i, \quad i = 1, \ldots, m \end{array}$$

The vectors $c, a_1, \ldots, a_m \in \mathbb{R}^n$ and the constants $b_1, \ldots, b_m \in \mathbb{R}$. The objective function is an affine function. The feasible set of the linear programming problem is a convex polyhedron. The set is defined as the intersection of finitely number of half spaces, and each half space is defined by a linear inequality.

There are many good methods for solving linear programming problems, and two of them which we are going to present later in details in Chapter **5** are Simplex method and Interior point methods. An optimization problem is not always in a linear program form, but it can be recast as one by using techniques. We give an example here by using the Chebyshev approximation problem,

$$\text{minimize} \quad \max_{i=1,\ldots,p} |a_i^T x - b_i|$$

We recast this problem as a linear programming problem,

$$\begin{array}{lll} \text{minimize} & t & \\ \text{subject to} & a_i^T x - t \leq b_i, & i = 1, \ldots, p \\ & -a_i^T x - t \leq -b_i, & i = 1, \ldots, p \end{array}$$

where $x \in \mathbb{R}^n$ and $t \in \mathbb{R}$.

2.3 Approximation

Approximation is defined as areas close to the exactly. When the exactly value is unknown and difficult to obtain, then approximation is used. This is when some known information around the exact value may be able to represent the exact value.

2.3.1 Norm Approximation

A norm approximation problem can be on the form

$$\text{minimize} \;\; \|Ax - b\| \tag{2.1}$$

A is an $m \times n$ matrix, $b \in \mathbb{R}^m$, $x \in \mathbb{R}^n$ and $\|\cdot\|$ is a norm on $\mathbb{R}^m$. We present some approximation interpretations.

In a weighted norm approximation we have weights in the objective function,

$$\text{minimize} \quad \|W(Ax - b)\|$$

The matrix $W \in \mathbb{R}^{m \times m}$ is the weighting matrix. If we denote $\hat{A} = AW$ and $\hat{b} = bW$, then the problem will have the same standard form like (2.1).

In a least-squares approximation problem the objective function is a sum of squares of the residual $Ax - b$,

$$\text{minimize} \quad \|Ax - b\|_2^2 \; = r_1^2 + \ldots + r_m^2$$

This problem is equivalent to a norm approximation problem using the Euclidean norm.

In a Chebyshev approximation problem we use the ℓ_∞- norm instead,

$$\text{minimize} \quad \|Ax - b\|_\infty = \max|r_1|, \ldots, |r_m|$$

This problem can be recast as a linear programming problem similar to how we described earlier in Section **2.2.1**.

If we now use the ℓ_1- norm, we have that the objective function is a sum of the absolute value of the residuals,

$$\text{minimize} \quad \|Ax - b\|_1 = |r_1| + \ldots + |r_m|$$

This problem can be recast as a linear programming problem

$$\begin{array}{ll} \text{minimize} & \mathbf{1}^T t \\ \text{subject to} & Ax - t \leq b \\ & -Ax - t \leq -b \end{array}$$

where $t \in \mathbb{R}^m$.

2.3.2 Regularized Approximation

We consider a convex optimization problem with two objective functions, $\|Ax - b\|$ and $\|x\|$, which we want to minimize. In a regularized approximation we want to make x small and at the same time also make $Ax - b$ small.

$$\text{minimize (w.r.t. } \mathbb{R}^2) \quad (\|Ax - b\|, \|x\|)$$

This formulation is called Bi-criterion formulation. If we first take the minimum value of $\|x\|$, that is when $x = 0$. We will then get that the residual norm is $\|b\|$. This is one endpoint of the optimal trade-off between $\|Ax - b\|$ and $\|x\|$. If both norms are Euclidean, the other endpoint is $x = A^{\dagger}b$, where $A^{\dagger}$ is the pseudo-inverse of A.

We can for example use weights in the objective function for regularizing this problem,

$$\text{minimize} \quad \|Ax - b\| + \lambda\|x\|$$

where $\lambda > 0$ varies. λ is chosen for making both $\|Ax - b\|$ and $\|x\|$ small.

ℓ_1- norm Regularization

Now if we use the ℓ_1- norm for the regularization, then we can find a sparse solution. We have the problem

$$\text{minimize} \quad \|Ax - b\|_2 + \lambda\|x\|_1$$

The Euclidean norm is used on the residual and the ℓ_1- norm is used on the regularized term. By choosing different values for λ, we get an approximation of the optimal trade-off between $\|Ax-b\|_2$ and the sparsity of the vector x. By sparsity of the vector x, we mean the number of non-zero elements. We let A be an $m \times n$ matrix and we have a vector $b \in \mathbb{R}^m$ that can be fit by a linear combination of $p < n$ columns of A. One approach is to find a small value λ so that the sparsity of the vector x is equal to p. Then we find the value of x that minimizes $\|Ax - b\|_2$.

2.3.3 Sparse Approximation

Consider we have a given matrix $A \in \mathbb{R}^{m\times n}$ and an observation vector $b \in \mathbb{R}^m$. Our sparse approximation is the problem of estimating a sparse vector $x \in \mathbb{R}^n$ that satisfies a linear system of equations of A and b. The problem is represented as

$$\begin{array}{ll} \text{minimize} & \|x\|_0 \\ \text{subject to} & Ax = b \end{array}$$

$\|x\|_0$ is defined as the number of non-zero components of x,

$$\|x\|_0 = \#\{i : x_i \neq 0,\ i = 1, \ldots, n\}$$

This is an NP-hard problem. NP stands for "Non-deterministic Polynomial". It represents one type of problem where the solutions can be determined using a "non-deterministic" computer. An NP-hard problem can not be solved on a standard computer in polynomial time, although it can be simulated. The simulation takes exponential time, so as the problem size grows, the time take for solving also grows. We therefore do a convex relaxation of the problem by using ℓ_1- norm instead of $\|\cdot\|_0$. A relaxation is an approximation of a difficult problem to a problem which is easier to solve. The ℓ_1- norm of the vector x is a sum of the absolute value of the entries in x,

$$\|x\|_1 = \textstyle\sum_{i=1}^{n} |x_i|$$

We can solve the problem with the ℓ_1- norm and be able to find a sparse solution under some conditions. This will be detailed explained in Chapter **4**.

3. APPLICATIONS

Sparse approximation is used in mathematical and engineering settings. It is used for compression of signals and images. Sparse approximation is also used in regularization as we have seen. Each application below which we are going to introduce briefly about requires different problem formulation, but the main goal is to find a good approximation, a sparse approximation. The images that are used for the applications, image denoising and inpainting, are taken from [7, 8]. Cryptography is based on [1]. The figure in traffic monitoring is taken from `http://www.ibm.com/developerworks/websphere/library/techarticles/ind-inteltrans/`.

3.1 Image Denoising and Inpainting

Image denoising is an image process where we want to remove the noise that the image has been disrupted with during transfer or while being stored. For removing this noise, we use sparse approximation. Consider we have an image f, and this image has been disrupted with the noise v. We let Φ be a sensing basis. Our measured image is y,

$$y = \Phi f + v$$

To remove the noise from the measured image y, we solve the problem

$$\begin{aligned}&\hat{f} = \arg\ \min_f \|f\|_1\\ &\text{subject to } \ \|\Phi f - y\|_2 \leq \epsilon\end{aligned} \tag{3.1}$$

Another possible formulation of this problem is

$$\hat{f} = \arg\ \min_f \frac{1}{2}\|\Phi f - y\|_2^2 + \lambda\|f\|_1 \tag{3.2}$$

For some choices of the parameter λ, the problem (3.1) is equivalent to the problem (3.2).

The images below taken from [8] show the original image "Barbara", when the image is disrupted with noise and when the image has been denoised.

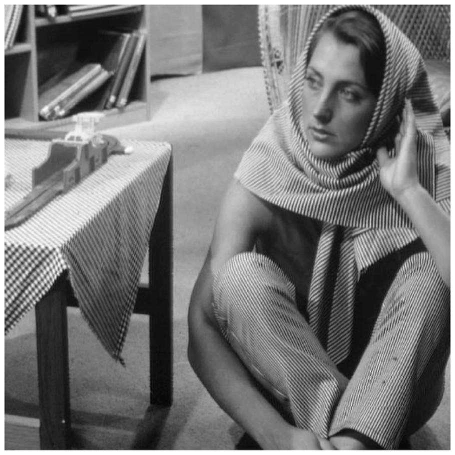


Fig. 3.1: Original image

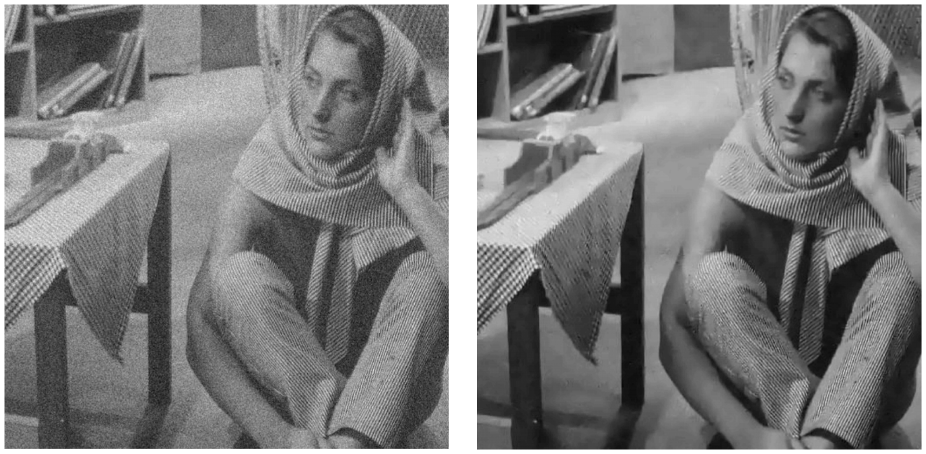


Fig. 3.2: Left: Noisy image, Right: Denoised image

Image inpainting is an image process for reconstructing parts of the images that have been lost. We are using the remaining parts of the image for building an approximation while we ignore the lost parts of the image. We have the measured image

$$y = \Phi f$$

Φ is the sensing basis and f is the original image. We denote the parts of the image that have been lost by a diagonal matrix M. To inpaint the image, we solve the problem

$$\hat{f} = \arg\min_f \|f\|_1$$
$$\text{subject to} \quad My = M\Phi f$$

Another formulation of the problem is

$$\hat{f} = \arg\min_f \|f\|_1 + \lambda\|M(y - \Phi f)\|_2^2$$

where $\lambda > 0$.

Below the images of "Barbara" taken from [7] show this. On the left side, the images have been inpainted, and on the right side, it shows how the images looked like before inpainting. From top, the image is with 20% missing pixels. In the middle, it is 50%. The last image is with 80 % missing pixels.

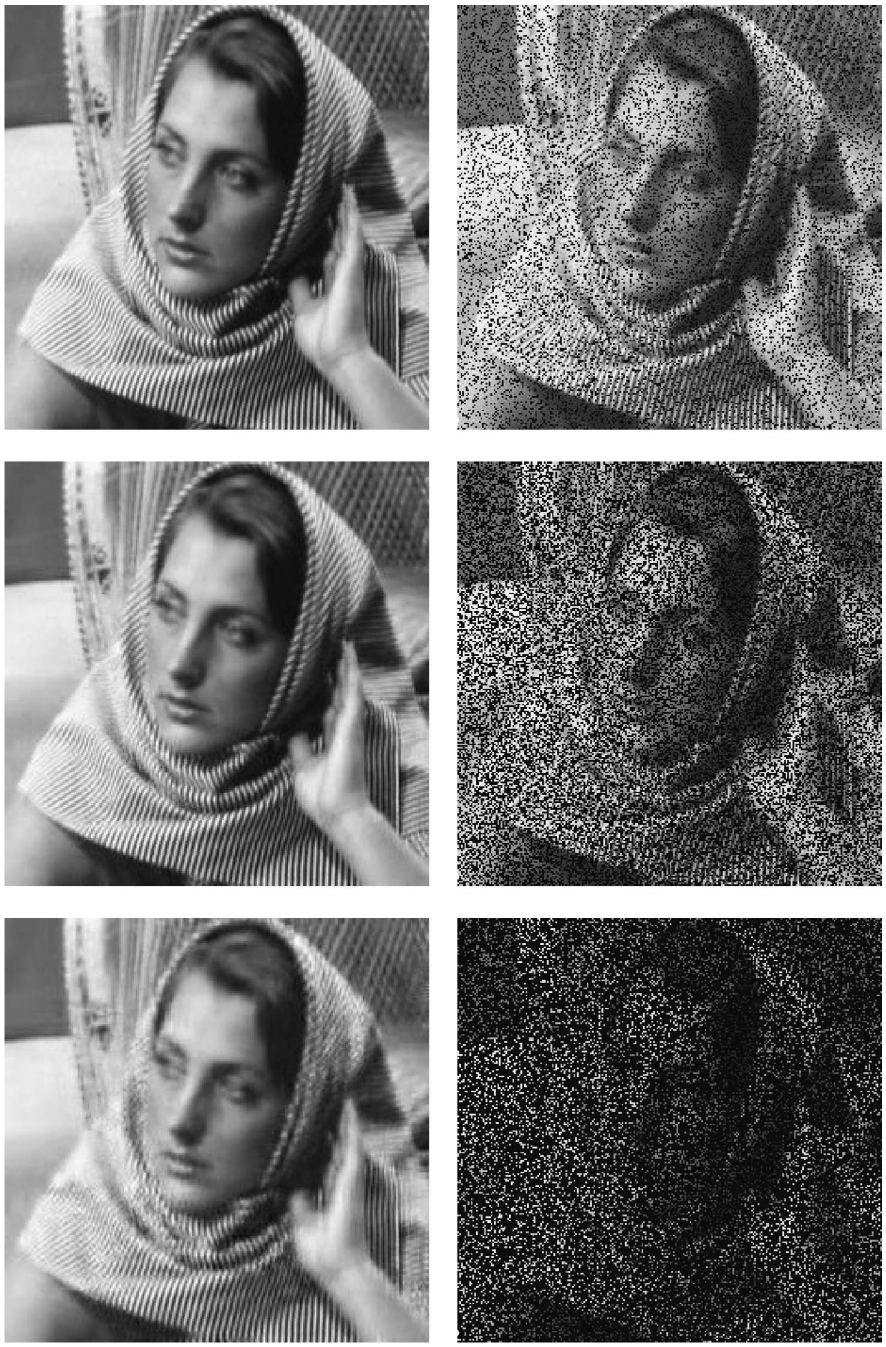


Fig. 3.3: From top: images with 20%, 50% and 80% missing pixels. Left: after inpainted. Right: before inpainted

3.2 Cryptography

Cryptography [1] is the study of hiding information from secure communication. This is for keeping the information secret and safe. Modern cryptography is a mix of mathematics and computer science.

By using cryptography on the message we want to send, it will be changed or encrypted before it is sent. The message we want to send is called a *plaintext*. The method of changing text is called a *code* or a *cipher*. The changed text is called *ciphertext*. The message is difficult to read after it has been changed. The one who wants to read it must therefore change it back or decrypt it. Only the sender and the receiver know the secret way to change it, while other people can not.

The problem is described as follow. We have $A = \Psi\Phi$, where $\Psi \in \mathbb{R}^{m\times m}$ is a random matrix, $\Phi \in \mathbb{R}^{m\times n}$ where $m < n$. We also have a vector x which is the secret data. Person1 sends A to Person2, then Person2 adds the secret data with A by computing

$$z = Ax$$

and then sends z to Person1. Person1 tries to get the secret data x by solving

$$z = Ax$$
$$z = \Psi\Phi x$$

We set $y = \Phi x$. So first we solve for y,

$$z = \Psi y$$
$$y = \Psi^{-1} z$$

Finally we can solve for x,

$$y = \Phi x$$

This is equivalent to solving the minimization problem

$$\begin{aligned} &\text{minimize} && \|x\|_1 \\ &\text{subject to} && Ax = z \end{aligned}$$

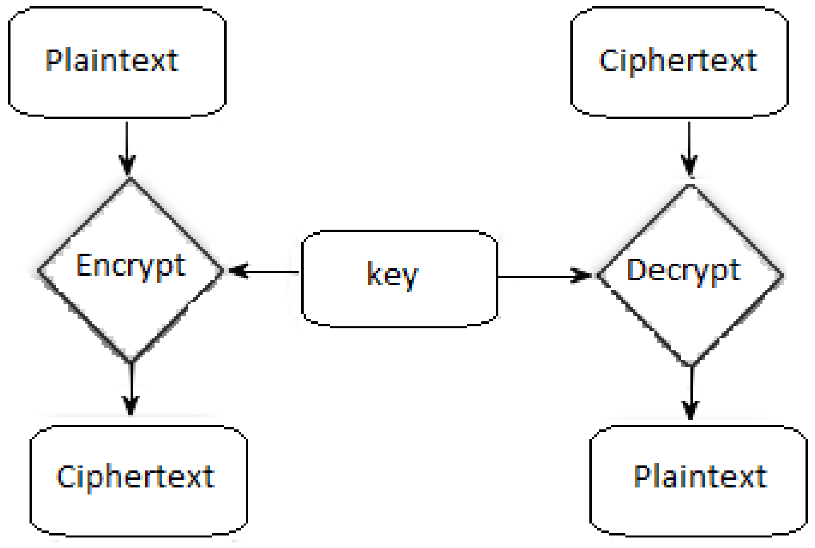


Fig. 3.4: Cryptography

3.3 Traffic Monitoring

Traffic monitoring is a system that monitors traffic data. It could be monitoring of network traffic data, transportation traffic data or other infrastructure traffic data. In traffic monitoring we are dealing with very large data. We store the data in multiple dimensions and time scales. Large data is transported across a network to individual operators. Some data are being analyzed and some other are being stored for other purposes. The management of these data will eventually increase, therefore we need to reduce the storage size of the data. For this we need techniques for compression, and our goal is to obtain low error representations with as little storage as possible. We use the sparse approximation.

For example, we let a matrix A representing the traffic data collection. It can consist of periodic and non-periodic variations. The problem can be formulated similar to the problem for image denoising in Section **3.1**. The measured data is y, the data we want to store is x and the noise is v.

$$y = Ax + v$$

We remove the noise from the data by solving the problem

$$\begin{aligned} &\hat{x} = \arg\,\min_x \|x\|_1 \\ &\text{subject to } \ \|Ax - y\|_2 \leq \epsilon \end{aligned}$$

For solving sparse approximation problems we use greedy algorithms. These algorithms will be presented later in Chapter **5**.

The figure below is taken from `http://www.ibm.com/developerworks/websphere/library/techarticles/ind-inteltrans/`. It shows a traffic data collection. We see first that the data is stored in a *Data collection.* In the process *Data cleansing* errors are found and the data is being corrected. When coming to the stage *Data fussion*, the data is being filtered so the unnecessary information from the data is removed. In our case, this stage is where we are solving the sparse approximation problem. After this stage, the data continues to *Data warehouse management* for analysis, visualization or being stored.

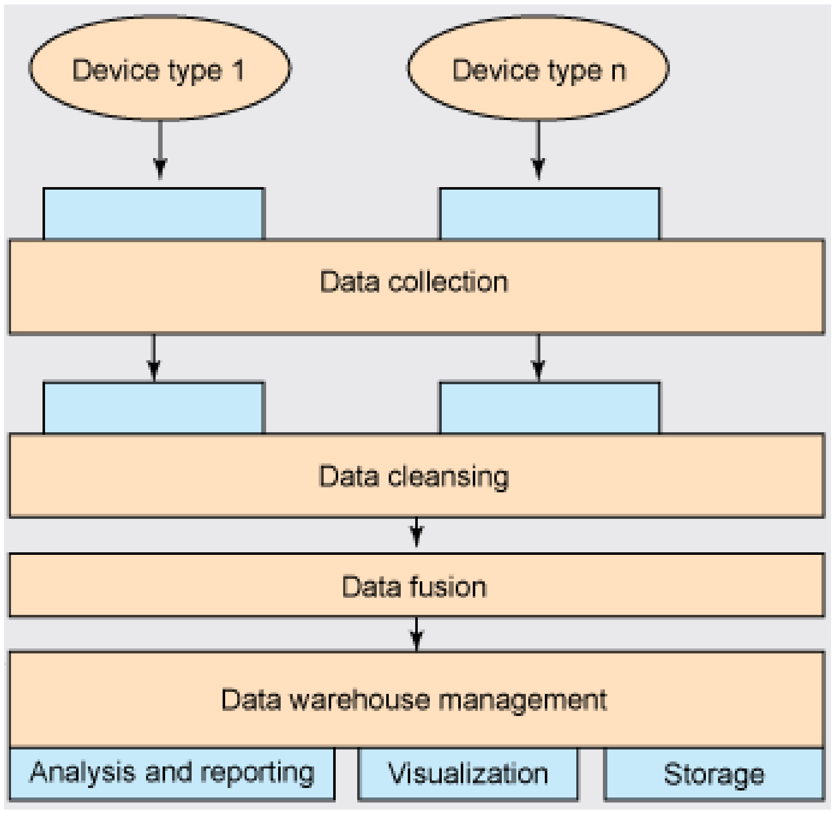


Fig. 3.5: Traffic monitoring

4. COMPRESSED SENSING

Compressed Sensing is about representing signals by using few non-zero coefficients in a basis or a dictionary. A basis is a set of vectors which are linearly independent. The definitions on what a basis and a dictionary are will be more discussed later in this chapter. A vector which have few non-zero coefficients is called a sparse vector.

Our goal is to reconstruct a signal by finding solutions to an underdetermined linear system. An underdetermined linear system is a system where we have more unknowns than equations. In general, the system has an infinite number of solutions. Compressed Sensing offers the use of sparsity by allowing solutions with few non-zero coefficients, but not all underdetermined linear systems have sparse solutions. However, if the system does have a unique sparse solution, then Compressed Sensing will allow the recovery of that solution. For the system to have sparse recovery, it requires conditions which will be presented.

Compressed Sensing has been used in many areas, such as engineering and computer science. In this chapter we will first define what a basis and a frame are, and then how they are used in sparse models. We present important conditions, and finally how to solve the problem. This chapter is mainly based on the book *Compressed Sensing* written by Y. C. Eldar and

G. Kutyniok, [13].

4.1 Bases and Frames

A basis is a spanning set of vectors which are linearly independent. Let $\{\phi_i\}_{i=1}^n$ in $\mathbb{R}^n$ be vectors in a basis. Since the vectors are linearly independent, the basis has the property that for all $c_1, \ldots, c_n \in \mathbb{R}$, if

$$c_1\phi_1 + \ldots + c_n\phi_n = 0$$

then

$$c_1 = \ldots = c_n = 0$$

Since this set of vectors is a spanning set, the basis has the property that each vector in the set has a unique representation as a linear combination of these basis vectors. For every $x \in \mathbb{R}^n$, there exist $c_1, \ldots, c_n \in \mathbb{R}$ such that

$$x = \textstyle\sum_{i=1}^n c_i\phi_i$$

Let Φ denote an $n \times n$ matrix with columns given by ϕ_i. Then we can represent this more compactly as

$$x = \Phi c$$

The coefficients $c_1, \ldots, c_n$ are uniquely determined by x.

A generalization of a basis to a spanning set of vectors which are linearly dependent results in what it is called a frame. Since the vectors are linearly dependent, the coefficients $c_1, \ldots, c_n$ become more difficult to determine. One way is to remove the vectors from the frame until it becomes linearly independent, but the problem is that the frame might lose its possibility to span the space. To solve this we add more vectors than necessary to represent x. This means that we do not need to remove the vectors from the frame. The coefficients $c_1, \ldots, c_n$ are therefore no longer uniquely determined by x, as like for the basis. The vector x can be represented as a linear combination of ϕ_i in many ways.

Let $\{\phi_i\}_{i=1}^n$ in $\mathbb{R}^m$ be vectors in a frame corresponding to a matrix $\Phi \in \mathbb{R}^{m \times n}$, where $m < n$. For all vectors $x \in \mathbb{R}^m$, the frame satisfies

$$A \, \|x\|_2^2 \; \leq \; \|\Phi^T x\|_2^2 \; \leq B \, \|x\|_2^2$$

where A and B are two real numbers and $0 < A \leq B < \infty$. They can be chosen independently of x, they only depend on the matrix Φ. A and B are called lower and upper frame bounds.

A basis or a frame is sometimes referred as a dictionary or an overcomplete dictionary, with the dictionary elements being called atoms.

4.2 Sparse Models

Let Φ be an $m \times n$ matrix, where $m < n$. We have an underdetermined linear system $\Phi x = b$. If it is consistent, it will have an infinite number of solutions. Therefore we want to regularize the problem so that there exists a unique solution x.

For regularizing an underdetermined linear system, one common choice is to choose the vector x that satisfies $\Phi x = b$ and minimizes the standard ℓ_2- norm, $\|x\|_2 = (\sum_i x_i^2)^{\frac{1}{2}}$. Unfortunately the minimum ℓ_2- norm solution is almost never sparse. In figure 4.1, the hyperplane is representing the set of all solutions to $\Phi x = b$. The circle is representing the ℓ_2- norm solution. When the radius to this circle increases until it touches the hyperplane, the point of contact is the minimum value of $\sqrt{x_1^2 + x_2^2}$ among all points on the line and it is the solution to $\Phi x = b$ with minimum ℓ_2- norm. As we can see in the figure, both components are non-zero at this point.

Now we want to look for solutions to $\Phi x = b$ that have the fewest possible non-zero components, sparse solutions. We first define $\|x\|_0$ as the number of non-zero components in x. For finding a sparse solution, we will now regularize our problem by looking for a vector x that satisfies $\Phi x = b$ and minimizes $\|x\|_0$.

In figure 4.1, the thick black line is representing the set $\{x : \|x\|_0 = 1\}$. It coincides with the coordinate axes. The points of contact between the solution set $\Phi x = b$ and the set $\{x : \|x\|_0 = 1\}$ are at two places, each on a coordinate axis. These solutions are sparse.

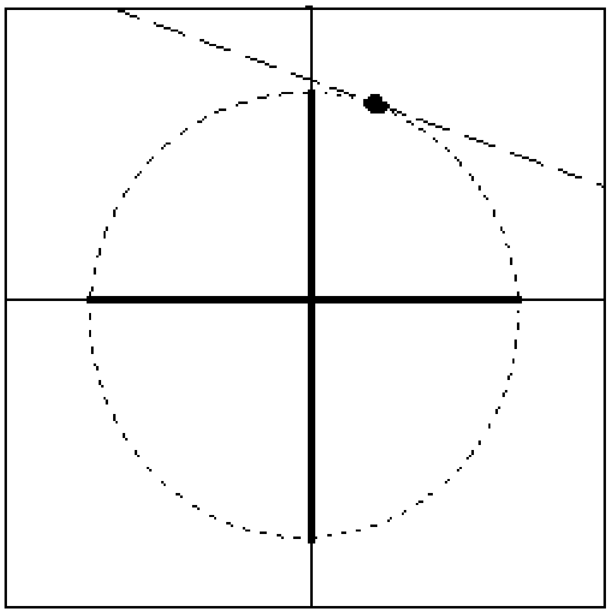


Fig. 4.1

$----$	$\{x : \Phi x = b\}$
thick black line	$\{x : \|x\|_0 = 1\}$
$\cdots\cdots\cdots$	$\{x : \|x\|_2 = c\}$, (c is a constant)

Unfortunately if the system is large, then finding the solution to a linear system $\Phi x = b$ with the fewest non-zero components will be computationally intractable. Computationally intractable means that a problem can be solved in theory, but in practise it takes too long for their solution to be useful.

Now we need a regularization technique that can find sparse solutions and yet remains computationally tractable. Computationally tractable means that a problem can be solved in polynomial time. Again we regularize our problem by looking for a vector x that satisfies $\Phi x = b$, but this time x minimizes the ℓ_1- norm, defined as

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

In figure 4.2, the square is representing the ℓ_1- norm solution. We see that the point of contact between the solution set $\Phi x = b$ and the set of ℓ_1- norm solutions is a non-zero component. It also agrees with a sparse solution produced by ℓ_0 regularization.

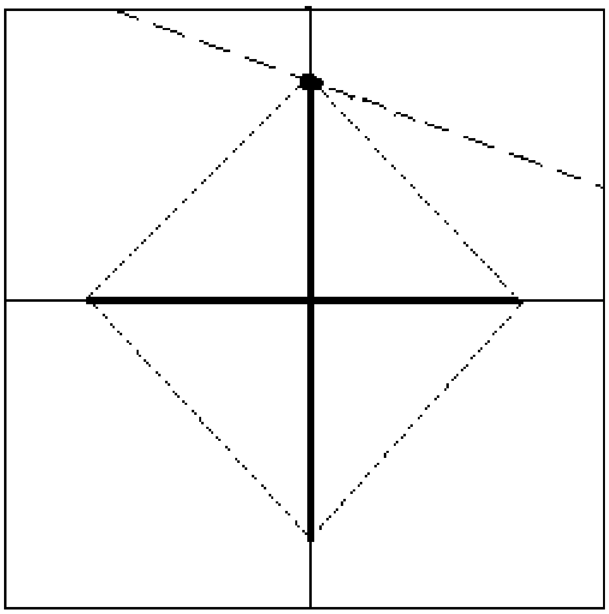


Fig. 4.2

$----$	$\{x : \Phi x = b\}$
thick black line	$\{x : \|x\|_0 = 1\}$
$\cdots\cdots\cdots$	$\{x : \|x\|_1 = d\}$, (d is a constant)

Solving the ℓ_1- norm minimization problem might look difficult, because $|x_1|$ is not differentiable. But it turns out that this problem can be solved by using convex optimization. The problem can be recast as a linear programming problem.

4.3 Conditions for Sparse Recovery

A vector x is *k-sparse* when it has at most k non-zero components, that is $\|x\|_0 \leq k$. We let

$$\textstyle\sum_k = \{x : \|x\|_0 \leq k\}$$

denote the set of all k-sparse vectors.

We are now going to present conditions guaranteeing that there exists k-sparse solution to $\Phi x = b$. This will help us being able to distinguish it from all of the other solutions.

4.3.1 Null Space Condition

The null space of a matrix Φ is defined as

$$\mathcal{N}(\Phi) = \{z : \Phi z = 0\}$$

First we let $x^\star$ satisfy $\Phi x^\star = b$. All the other solutions to $\Phi x = b$ are on the form $x = x^\star + z$, where $\Phi z = 0$. This means that z is in $\mathcal{N}(\Phi)$, the null space of Φ. We assume $x^\star \in \Sigma_k$ for some k, and we will see that $x^\star$ is the only k-sparse solution under some conditions.

Let $x^{\star\star}$ be another distinct k-sparse solution, that is $x^{\star\star} \in \Sigma_k$. If we want to recover all sparse vectors x from Φx, we must have that $\Phi x^{\star} \neq \Phi x^{\star\star}$, since otherwise we can not distinguish $x^{\star}$ from $x^{\star\star}$. If we have $\Phi x^{\star} = \Phi x^{\star\star}$, then $\Phi(x^{\star} - x^{\star\star}) = 0$, that is $x^{\star} - x^{\star\star} \in \mathcal{N}(\Phi)$, but $x^{\star} - x^{\star\star}$ is not the zero vector. If $x^{\star}$ and $x^{\star\star}$ are any vectors in Σ_k, then $x^{\star} - x^{\star\star} \in \Sigma_{2k}$. Therefore we conclude that if $\Phi x = b$ has more than one k-sparse solution, $\mathcal{N}(\Phi)$ must contain a non-zero *2k*-sparse vector. The contrapositive of this statement yields the next lemma [6].

Lemma 1

Suppose that $\Sigma_{2k} \cap \mathcal{N}(\Phi) = \{0\}$, that is all non-zero elements in the nullspace of Φ have at least *2k+1* non-zero components. Then any k-sparse solution to $\Phi x = b$ is unique.

Lemma 2

The condition $\Sigma_{2k} \cap \mathcal{N}(\Phi) = \{0\}$ holds if and only if every subset of *2k* columns of Φ is linearly independent.

This property can also be explained by using the *spark* of the matrix Φ. It is defined as the smallest number of linearly dependent columns in the matrix Φ.

$$\text{spark}(\Phi) = \min \{\|x\|_0 : \Phi x = 0,\ x \neq 0\}$$

This definition implies a guarantee , [9].

Theorem 1

For any vector $b \in \mathbb{R}^m$, there exists at most one vector $x \in \sum_k$ such that $b = \Phi x$ if and only if $spark(\Phi) > 2k$.

Let us now define the *rank* of a matrix. The rank of the matrix Φ is defined as the maximum number of columns of Φ that are linearly independent.

There is a connection between the two definitions of the spark and the rank of the matrix Φ. If we have a k-dimensional subset of column vectors of Φ that are linearly independent, but there is a subset of *k+1* column vectors which are linearly dependent, then $spark(\Phi) = k + 1$. For an $m \times n$ matrix Φ, if $m = n = 1$, then $spark(\Phi) = 1$. If $m = n \geq 2$ and Φ is invertible, we have that $spark(\Phi) = n + 1$. With $m \geq 2$, the spark and the rank of Φ are related by

$$2 \leq spark(\Phi) \leq rank(\Phi) + 1 \tag{4.1}$$

.

Example 1

$$\Phi = \begin{pmatrix} 1 & 0 & 1 & -1 \\ 0 & 1 & -1 & 1 \end{pmatrix}$$

We get that $rank(\Phi) = 2$, since there are two linearly independent column vectors. We get $spark(\Phi) = 3$, since there are minimum three column vectors that are linearly dependent. So we see that (4.1) holds.

Example 2

$$\Phi = \begin{pmatrix} 1 & 0 & -1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

We get that $rank(\Phi) = 2$, since there are two linearly independent column vectors. But here we get that $spark(\Phi) = 2$, since there are minimum two column vectors that are linearly dependent. Again we see that (4.1) holds.

4.3.2 The Restricted Isometry Property

The condition $\Sigma_{2k} \cap \mathcal{N}(\Phi) = \{0\}$ requires that no non-zero vector $x \in \Sigma_{2k}$ satisfies $\Phi x = 0$. We are now going to restrict to unit vectors with respect to the ℓ_2- norm, because if $x \neq 0$, then $\Phi x = 0$ if and only if $\Phi u = 0$, where $u = \frac{x}{\|x\|_2}$ is a unit vector. Therefore we look for a condition that will make sure that no unit vector $u \in \Sigma_{2k}$ satisfies $\Phi u = 0$. First we require that there exists a positive constant c_1 such that for all *2k*-sparse unit vectors u, we have $\|\Phi u\|_2^2 \geq c_1$. Since $\|\Phi u\|_2^2 = 0$, this will remove $\Phi u = 0$. So if our requirement holds, then $\Sigma_{2k} \cap \mathcal{N}(\Phi) = \{0\}$.

The mapping $x \to \|\Phi x\|_2^2$ is continuous from $\mathbb{R}^n$ to $\mathbb{R}$. The set of *2k*-sparse unit vectors in $\mathbb{R}^n$ is also compact, so this means that $\|\Phi u\|_2^2$ has a maximum value on this set. Therefore there exists a constant $c_2 > 0$ such that $\|\Phi u\|_2^2 \leq c_2$ for all unit vectors $u \in \Sigma_{2k}$.

Combining all this together, we will have that

$$c_1 \leq \|\Phi u\|_2^2 \leq c_2$$

We start by redefining Φ by multiplying it with $\sqrt{2/(c_1+c_2)}$. We will multiply the whole equation with $\frac{2}{c_1+c_2}$.

$$\tfrac{2c_1}{c_2+c_1} \leq \|\Phi u\|_2^2 \leq \tfrac{2c_2}{c_2+c_1}$$

$$\tfrac{c_2+c_1-c_2+c_1}{c_2+c_1} \leq \|\Phi u\|_2^2 \leq \tfrac{c_2+c_1+c_2-c_1}{c_2+c_1}$$

$$1-\tfrac{c_2-c_1}{c_2+c_1} \leq \|\Phi u\|_2^2 \leq 1+\tfrac{c_2-c_1}{c_2+c_1}$$

We set $\delta = \frac{c_2-c_1}{c_2+c_1}$. Then we get

$$1-\delta \leq \|\Phi u\|_2^2 \leq 1+\delta$$

where both Φ and b are rescaled by a factor $\sqrt{2/(c_1+c_2)}$.

Definition 1

An $m \times n$ matrix Φ satisfies the restricted isometry property (RIP) of order k if there is some constant $\delta_k \in (0,1)$ such that

$$1-\delta_k \leq \|\Phi u\|_2^2 \leq 1+\delta_k$$

for all k-sparse unit vectors $u \in \mathbb{R}^n$. If Φ satisfies the RIP of order *2*k for some $k \geq 1$, then $\Sigma_{2k} \cap \mathcal{N}(\Phi) = \{0\}$ and any k-sparse solution to $\Phi x = b$ is unique.

For any vector $x \in \mathbb{R}^n$, we can write x as $x = \|x\|_2 u$, where $u = \frac{x}{\|x\|_2}$ is a unit vector. We will then get that Definition 1 is equivalent to

$$(1-\delta_k)\|x\|_2^2 \leq \|\Phi x\|_2^2 \leq (1+\delta_k)\|x\|_2^2$$

for any k-sparse vector $x \in \mathbb{R}^n$.

4.3.3 Coherence

We have now discussed about the spark, the null space condition and the restricted isometry property. They all give us guarantees for the recovery of k-sparse vectors. Sometimes we want to use properties of the matrix Φ that are easily to compute to give us more concrete recovery guarantees, and one such property is the *coherence* of a matrix.

Definition 2

The coherence of a matrix Φ, $\mu(\Phi)$, is the maximum absolute value of the inner product between any two columns ϕ_i, ϕ_j of Φ,

$$\mu(\Phi) = \max{}_{1 \leq i < j \leq n} \frac{|\langle \phi_i, \phi_j \rangle|}{\|\phi_i\|_2 \|\phi_j\|_2}$$

The coherence of a matrix tells us about the dependence between the columns of Φ. If Φ is an orthogonal matrix, the inner products between the columns would be zero, so $\mu(\Phi) = 0$. For matrices with more columns than rows, $\mu(\Phi) > 0$. We desire a small $\mu(\Phi)$ for recovery problems, Φ is then closer to being orthogonal. The coherence of a matrix is always in the range $\mu(\Phi) \in \left[\sqrt{\frac{n-m}{m(n-1)}}, 1\right]$.

By relating the coherence and spark [13], we get that for any matrix Φ,

$$\text{spark}(\Phi) \geq 1 + \frac{1}{\mu(\Phi)}$$

4.4 Recovery via ℓ_1 Minimization

Our goal is to recover the vector x from $\Phi x = b$. The first approach we are considering here is to recover x by solving an optimization problem of the form

$$\begin{aligned} &\hat{x} = \arg\,\min_x \|x\|_0 \\ &\text{subject to } \; x \in \mathcal{B}(b) \end{aligned} \tag{4.2}$$

where $\mathcal{B}(b)$ ensures that $\hat{x}$ is consistent with the measurements b.

We set $\mathcal{B}(b) = \{x : \Phi x = b\}$ if our measurements are exact and without being disrupted by noise. We set $\mathcal{B}(b) = \{x : \|\Phi x - b\|_2 \leq \epsilon\}$ if our measurements are disrupted with noise. For both cases, (4.2) can recover the sparsest vector x that is consistent with the measurements b.

Under the right conditions on Φ, it is possible to solve (4.2). But since the objective function $\|\cdot\|_0$ is non-convex, (4.2) is very difficult to solve. It is an NP-hard problem.

As we have mentioned earlier, if the system is large, then finding the solution to this problem would be computationally intractable. Therefore we want to translate this problem into something more easier to solve. That is to replace $\|\cdot\|_0$ with its convex approximation $\|\cdot\|_1$. We consider

$$\begin{aligned} &\hat{x} = \arg\,\min_x \|x\|_1 \\ &\text{subject to } \; x \in \mathcal{B}(b) \end{aligned} \tag{4.3}$$

We assume that $\mathcal{B}(b)$ is convex, then (4.3) is computationally feasible. If $\mathcal{B}(b) = \{x : \Phi x = b\}$, then our problem can be recast as a linear programming problem. So the use of ℓ_1 minimization can find sparse solutions and yet remains computationally tractable.

There exist efficient and accurate numerical solvers for convex optimization. If $\mathcal{B}(b) = \{x : \|\Phi x - b\|_2 \leq \epsilon\}$, the minimization problem (4.3) becomes a convex program with a conic constraint. One possible formulation of this problem is that we can consider the unconstrained version of this problem, that is

$$\hat{x} = \arg\min_x \tfrac{1}{2}\|\Phi x - b\|_2^2 + \lambda\|x\|_1$$

For some choices of the parameter λ, this optimization problem will give us the same result as the constrained version of the problem given by

$$\begin{aligned} &\hat{x} = \arg\min_x \|x\|_1 \\ &\text{subject to} \quad \|\Phi x - b\|_2 \leq \epsilon \end{aligned}$$

4.5 Recovery via Greedy Algorithms

Greedy algorithms are algorithms which use many iterations to compute the result. At each stage, it makes a locally optimal choice with the hope of obtaining a global optimum. By making one greedy choice after another, it reduces each given problems into a smaller one. The idea behind a greedy algorithm is that it performs an iteration process and keep repeating until a convergence criterion is met. In our case, the algorithm obtains an improved estimate of the sparse vector at each iteration as the process runs. Some of the greedy algorithms are similar to ℓ_1 minimization algorithms. However, the techniques required to prove performance guarantees are different.

We will here mention two of the oldest and simplest greedy approaches, they are Orthogonal Matching Pursuit (OMP) and Iterative Thresholding.

The greedy approach OMP begins by finding the column of Φ that resembles the most with the current residual. The process will repeat and at each step it selects these columns which will then be added into a set. The algorithm will update the residuals by projecting the vector b onto the linear subspace spanned by the columns that have been selected.

Iterative thresholding algorithms are more straightforward. We consider Iterative Hard Thresholding (IHT). The algorithm iterates a gradient descent step followed by hard thresholding until a convergence criterion is met.

These two algorithms are detailed explained in Section **5.2** and Section **5.6**, also along with other algorithms.

5. ALGORITHMS

In this chapter we are going to discuss about methods used to solve sparse approximation problems. The two most common methods which are in use are Basis Pursuit and Orthogonal Matching Pursuit. Basis Pursuit has the advantage that the sparse approximation problem can be replaced by a convex problem, and there are efficient algorithms that can find the solutions. We will present algorithms which solve the problems involving the ℓ_1-norm, like the Simplex method and the Interior point methods, [11, 12]. Orthogonal Matching Pursuit [13] is a greedy method. The approximation is generated by going through an iteration process which builds up the solution.

We are also going to present a brief description of other methods, like Stagewise Orthogonal Matching Pursuit [2], Regularized Orthogonal Matching Pursuit [3], Compressive Sampling Matching Pursuit [13] and Iterative Hard Thresholding Algorithm [13].

5.1 Basis Pursuit

Our sparse problem is given as

$$\begin{aligned} &\text{minimize} && \|x\|_0 \\ &\text{subject to} && Ax = b \end{aligned} \tag{5.1}$$

where $b \in \mathbb{R}^m$ is a given vector, A is an $m \times n$ matrix and $x \in \mathbb{R}^n$ is the vector we want to find.

We have discussed earlier that since the objective function $\|\cdot\|_0$ is non-convex, the problem (5.1) is difficult to solve. Therefore we want to replace $\|\cdot\|_0$ with the ℓ_1- norm. This is the basic idea of Basis Pursuit, and it is given as

$$\begin{aligned} &\text{minimize} && \|x\|_1 \\ &\text{subject to} && Ax = b \end{aligned} \tag{5.2}$$

There are algorithms that will solve the Basis Pursuit problem (5.2). With the right conditions Basis Pursuit can find a sparse solution. Later in Chapter **6** we will see how we solve Basis Pursuit.

5.1.1 Simplex Method

The Simplex method is a method used to solve problems in linear optimization. The algorithm was first used by the American mathematician George Dantzig in 1947. The Simplex method solves a linear program of the form

$$\begin{aligned} &\text{minimize} && \textstyle\sum_{j=1}^n c_j x_j \\ &\text{subject to} && \textstyle\sum_{j=1}^n a_{ij} x_j \le b_i, \quad i = 1, \dots, m < n \\ & && x_j \ge 0, \qquad\qquad j = 1, \dots, n \end{aligned}$$

We have a set of equations. If no solution is found yet, we introduce the slack variables $x_{n+1}, \dots, x_{n+m}$. The initial basic feasible solution is the solution to the problem that satisfies

$$x_i = 0, \qquad i = 1, \ldots, n$$

$$x_i = b_{n-i}, \quad i = n+1, \ldots, n+m$$

When we have found the solution, we can make improvements for this solution. One nonbasic variable is chosen to be increased so that the value of the objective function, $\sum_{j=1}^{n} c_j x_j$, decreases. The variable which is being increased maintains the equality of all the equations while keeping the other nonbasic variables at zero, until one of the basic variables is reduced to zero and then being removed from the basis. This means that when a new variable becomes basic, another one becomes nonbasic. This process will be repeated.

There are three possible outcomes for this process. The first one is when the nonbasic variable no longer decreases the objective function. In this case the current solution is the optimal solution. The second possible outcome is when we get an unbounded solution. This is the result of when a nonbasic variable increases to infinity without making the basic variable decreasing to zero. The last possible outcome is when no solution exists.

Another alternative way to explain the Simplex method is that it is a procedure for making and testing vertex solutions to a linear program. It starts at an arbitrary vertex which is seen as a corner of the solution set. In each iteration, it selects the variable that makes the largest change towards the minimum or the maximum solution. That variable will be replaced, and then the Simplex method keep moving to a different vertex or corner of the solution set. Eventually it will get closer to the final solution. The algorithm is greedy since it selects the best choice at each iteration without needing information from previous or next iterations. Figure 5.1 is taken from `http://en.wikipedia.org/wiki/Simplex_algorithm`.

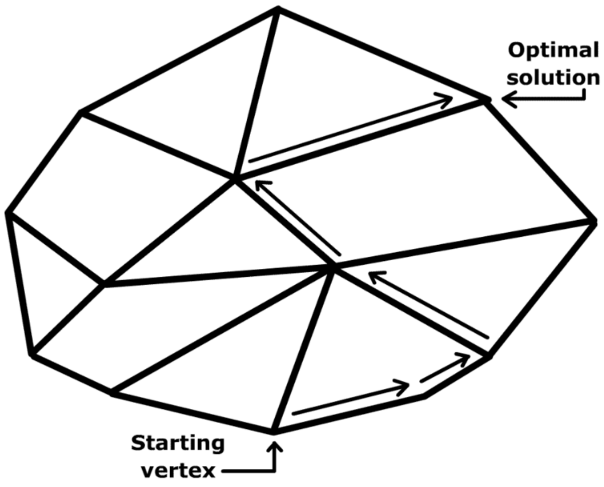


Fig. 5.1: Simplex method

5.1.2 Interior Point Methods

Interior point methods are algorithms used to solve linear and nonlinear convex optimization problems. The method was invented by John Von Neumann.

We consider a convex optimization problem with inequality constraints,

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m \\ & Ax = b \end{array} \tag{5.3}$$

where $A \in \mathbb{R}^{p \times n}$ with $rank(A) = p < n$, and $f_0, \ldots, f_m : \mathbb{R}^n \to \mathbb{R}$ are convex and twice continuously differentiable. We assume that an optimal solution $x^\star$ exists. Let $p^\star$ denote the optimal value $f_0(x^\star)$.

We assume that the problem is strictly feasible. This means that there exists an x that satisfies $Ax = b$ and $f_i(x) < 0$ for $i = 1, \ldots, m$. Therefore there also exists dual optimal $\lambda^\star \in \mathbb{R}^m$, $\nu^\star \in \mathbb{R}^p$, which together with $x^\star$ satisfy the Karush-Kuhn-Tucker (KKT) conditions

$$\begin{array}{rcl} Ax^\star = b, \;\; f_i(x^\star) & \leq & 0, \quad i = 1, \ldots, m \\ \lambda_i^\star & \geq & 0, \quad i = 1, \ldots, m \\ \nabla f_0(x^\star) + \sum_{i=1}^m \lambda_i^\star \nabla f_i(x^\star) + A^T \nu^\star & = & 0 \\ \lambda_i^\star f_i(x^\star) & = & 0, \quad i = 1, \ldots, m \end{array} \tag{5.4}$$

Interior point methods solve the problem (5.3) or the KKT conditions (5.4) by using Newton's method. Interior point methods solve an optimization problem with linear equality and inequality constraints by reducing it to an optimization problem with only linear equality constraints.

Logarithmic Barrier Function and Central Path

Now we want to formulate the inequality constrained problem (5.3) as an equality constrained problem so that we can use Newton's method. We do this by adding the inequality constraints into the objective function,

$$\begin{array}{ll} \text{minimize} & f_0(x) + \sum_{i=1}^m I_-(f_i(x)) \\ \text{subject to} & Ax = b \end{array} \tag{5.5}$$

where $I_- : \mathbb{R} \to \mathbb{R}$ is the indicator function. Below $u = f_i(x),\ i = 1, \ldots, m$.

$$I_-(u) = \begin{cases} 0 & u \leq 0 \\ \infty & u > 0 \end{cases}$$

Now the problem (5.5) is an equality constrained problem, but the objective function is not differentiable, so Newton's method cannot be used.

The barrier method approximates the indicator function I_- by the function

$$\hat{I}_-(u) = -\tfrac{1}{t} \log(-u)$$

where $t > 0$ is a parameter that sets the accuracy of the approximation. The function $\hat{I}_-$ is convex and differentiable. By changing out I_- with $\hat{I}_-$ in (5.5), we get

$$\begin{array}{ll} \text{minimize} & f_0(x) + \sum_{i=1}^{m} -\tfrac{1}{t} \log(-f_i(x)) \\ \text{subject to} & Ax = b \end{array} \tag{5.6}$$

The objective function is now convex and differentiable, so Newton's method can be applied. We let

$$\phi(x) = -\sum_{i=1}^{m} \log(-f_i(x))$$

This function is called the logarithmic barrier for the problem (5.3).

Now we consider the problem (5.6). We multiply the objective function by t,

$$\begin{array}{ll} \text{minimize} & tf_0(x) - \sum_{i=1}^{m} \log(-f_i(x)) \\ \text{subject to} & Ax = b \end{array} \tag{5.7}$$

We define $x^\star(t)$ as the solution of problem (5.7). The set of points $x^\star(t),\ t > 0$, are called central points. The condition of these points is that $x^\star(t)$ is strictly feasible, which means that it satisfies

$$Ax^\star(t) = b$$

$$f_i(x^\star(t)) < 0, \ \ i = 1, \ldots, m$$

and there exists a $\hat{\nu} \in \mathbb{R}^p$ such that

$$0 \ = \ t\nabla f_0(x^\star(t)) + \sum_{i=1}^{m} \frac{1}{-f_i(x^\star(t))} \nabla f_i(x^\star(t)) + A^T \hat{\nu} \tag{5.8}$$

holds. We define

$$\lambda_i^\star(t) = -\frac{1}{tf_i(x^\star(t))}, \ \ i = 1, \ldots, m, \qquad \nu^\star(t) = \frac{\hat{\nu}}{t}$$

Every central point, $x^\star(t)$, yields a dual feasible point, $\lambda^\star(t)$, $\nu^\star(t)$, and therefore a lower bound on the optimal value $p^\star$. The conditions (5.8) can be expressed as

$$t\nabla f_0(x^\star(t)) + t\sum_{i=1}^{m} \lambda_i^\star(t) \nabla f_i(x^\star(t)) + tA^T \nu^\star(t) = 0$$

$$\nabla f_0(x^\star(t)) + \sum_{i=1}^{m} \lambda_i^\star(t) \nabla f_i(x^\star(t)) + A^T \nu^\star(t) = 0$$

$x^\star(t)$ is minimizing the Lagrangian

$$L(x, \lambda, \nu) = f_0(x) + \textstyle\sum_{i=1}^{m} \lambda_i f_i(x) + \nu^T(Ax - b)$$

for $\lambda = \lambda^\star(t)$ and $\nu = \nu^\star(t)$. The dual function is

$$\begin{aligned} g(\lambda^\star(t), \nu^\star(t)) &= f_0(x^\star(t)) \ + \ \textstyle\sum_{i=1}^{m} \lambda_i^\star(t) f_i(x^\star(t)) \ + \ \nu^\star(t)^T(Ax^\star(t) - b) \\ &= f_0(x^\star(t)) - \tfrac{m}{t} \end{aligned}$$

The duality gap is $\frac{m}{t}$.

With a specified accuracy ϵ we can solve the problem (5.3). Let $t = \frac{m}{\epsilon}$. By using Newton's method we can solve the equality constrained problem

$$\begin{array}{ll} \text{minimize} & \frac{m}{\epsilon} f_0(x) + \phi(x) \\ \text{subject to} & Ax = b \end{array}$$

Newton's Method

An equality constrained minimization problem can be reduced to an equivalent unconstrained problem. This is done by eliminating the equality constraints. Another approach is to solve the dual problem by using an unconstrained minimization method, and then we get a dual solution. From this solution, we can recover the solution of the equality constrained problem.

Now we extend Newton's method so that it can directly handle the equality constraints. This method is better than reducing an equality constrained problem to an unconstrained problem, because the problem structure, such as sparsity, can be destroyed when eliminating the constraints or when forming it to a dual problem.

We consider a convex optimization problem with equality constraints

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & Ax = b \end{array} \tag{5.9}$$

where $A \in \mathbb{R}^{p \times n}$ with $rank(A) = p < n$, and $f : \mathbb{R}^n \to \mathbb{R}$ is convex and twice continuously differentiable. A point $x^\star$ is optimal for (5.9) if and only if there is a $\nu^\star \in \mathbb{R}^p$ such that

$$\begin{array}{l} Ax^\star = b \\ \nabla f(x^\star) + A^T \nu^\star = 0 \end{array} \tag{5.10}$$

The equality constrained optimization problem (5.9) and the KKT equations (5.10) are equivalent.

To extend Newton's method, we need that the initial point must be feasible, which means that x satisfies $Ax = b$. We must also have that the Newton step Δx_{nt} is a feasible direction, which means that $A\Delta x_{nt} = 0$. At a feasible point x, the Newton step Δx_{nt} solves the second-order Taylor approximation of *f*,

$$\begin{array}{ll} \text{minimize} & f(x) + \nabla f(x)^T v + \frac{1}{2}\nu^T \nabla^2 f(x) v \\ \text{subject to} & A(x + v) = b \end{array}$$

with a variable v. We define the Newton decrement

$$\lambda(x) = (\Delta x_{nt}^T \nabla^2 f(x) \Delta x_{nt})^{\frac{1}{2}}$$

Algorithm:

The inputs are a starting point x that satisfies $Ax = b$ and a tolerance $\epsilon > 0$.

1. Compute the Newton step Δx_{nt} and the Newton decrement $\lambda(x)$.

2. Stopping criterion, we stop if $\frac{\lambda^2}{2} \leq \epsilon$.

3. Choose step size t by using backtracking line search.

4. Update $x = x + t\Delta x_{nt}$.

Primal-Dual Interior Point Method

Primal-dual interior point method is similar to the barrier method. Both the primal and dual variables are updated at each iteration. When it requires high accuracy, primal-dual interior point method is more efficient than the barrier method. We are using Newton's method together with the modified KKT equations for computing the search directions.

So first we start with the modified KKT conditions

$$\begin{aligned} \nabla f_0(x) + \textstyle\sum_{i=1}^{m} \lambda_i \nabla f_i(x) + A^T \nu &= 0 \\ -\lambda_i f_i(x) &= \tfrac{1}{t}, \quad i = 1, \ldots, m \\ Ax &= b \end{aligned}$$

We express this as $r_t(x, \lambda, \nu) = 0$. For $t > 0$, we define

$$r_t(x,\lambda,\nu) = \begin{pmatrix} \nabla f_0(x) + Df(x)^T\lambda + A^T\nu \\ -\mathbf{diag}(\lambda) f(x) - \frac{1}{\tau}\mathbf{1} \\ Ax - b \end{pmatrix}$$

where $f : \mathbb{R}^n \to \mathbb{R}^m$ and the matrix Df is its derivative.

$$f(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix}, \qquad Df(x) = \begin{pmatrix} \nabla f_1(x)^T \\ \vdots \\ \nabla f_m(x)^T \end{pmatrix}$$

If x, λ, ν satisfy $r_t(x,\lambda,\nu) = 0$, then $x = x^\star(t)$, $\lambda = \lambda^\star(t)$ and $\nu = \nu^\star(t)$. x is primal feasible, and λ, ν are dual feasible. The duality gap is $\frac{m}{t}$.

The first equation of r_t,

$$r_{\text{dual}} = \nabla f_0(x) + Df(x)^T\lambda + A^T\nu$$

is called the dual residual. The second equation,

$$r_{\text{cent}} = -\mathbf{diag}(\lambda) f(x) - \tfrac{1}{\tau}\mathbf{1}$$

is called the centrality residual. This is the residual for the modified complementarity condition. The third and last equation,

$$r_{\text{pri}} = Ax - b$$

is called the primal residual.

For fixed t, at a point (x,λ,ν) that satisfies $f(x) < 0$, $\lambda > 0$, we are going to use Newton step for solving $r_t(x, \lambda, \nu) = 0$. So we denote the point and Newton step as

$$y = (x, \lambda, \nu), \qquad \Delta y = (\Delta x, \Delta\lambda, \Delta\nu)$$

The step is characterized by the linear equations

$$\begin{aligned}
&r_t(y + \Delta y) \approx r_t(y) + Dr_t(y)\Delta y = 0\\
&Dr_t(y)\Delta y = -r_t(y)\\
&\Delta y = -Dr_t(y)^{-1} r_t(y)
\end{aligned}$$

In terms of x, λ, ν, we have

$$\begin{pmatrix} \nabla^2 f_0(x) + \sum_{i=1}^m \lambda_i \nabla^2 f_i(x) & Df(x)^T & A^T \\ -\mathbf{diag}(\lambda) Df(x) & -\mathbf{diag}(f(x)) & 0 \\ A & 0 & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta\lambda \\ \Delta\nu \end{pmatrix} = - \begin{pmatrix} r_{\text{dual}} \\ r_{\text{cent}} \\ r_{\text{pri}} \end{pmatrix}$$

The solution of this is the primal-dual search direction $\Delta y_{pd} = (\Delta x_{pd}, \Delta\lambda_{pd}, \Delta\nu_{pd})$. From the second equation, we eliminate $\Delta\lambda_{pd}$ using

$$\begin{aligned}
&-\mathbf{diag}(\lambda) Df(x)\Delta x_{pd} - \mathbf{diag}(f(x))\Delta\lambda_{pd} = -r_{\text{cent}}\\
&\mathbf{diag}(f(x))\Delta\lambda_{pd} = -\mathbf{diag}(\lambda) Df(x)\Delta x_{pd} + r_{\text{cent}}\\
&\Delta\lambda_{pd} = -\mathbf{diag}(f(x))^{-1}\mathbf{diag}(\lambda) Df(x)\Delta x_{pd} + \mathbf{diag}(f(x))^{-1} r_{\text{cent}}
\end{aligned}$$

By substituting this into the first equation, we get

$$\begin{pmatrix} \nabla^2 f_0(x) + \sum_{i=1}^m \lambda_i \nabla^2 f_i(x) + \sum_{i=1}^m \frac{\lambda_i}{-f_i(x)} \nabla f_i(x) \nabla f_i(x)^T & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} \Delta x_{pd} \\ \Delta\nu_{pd} \end{pmatrix}$$

$$= -\begin{pmatrix} r_{\text{dual}} + Df(x)^T \mathbf{diag}(f(x))^{-1} r_{\text{cent}} \\ r_{\text{pri}} \end{pmatrix}$$

$$= -\begin{pmatrix} \nabla f_0(x) + \frac{1}{t}\sum_{i=1}^{m} \frac{1}{-f_i(x)} \nabla f_i(x) + A^T \nu \\ r_{\text{pri}} \end{pmatrix}$$

For the primal-dual interior point method, we define the surrogate duality gap. For any x that satisfies $f(x) < 0$ and $\lambda \geq 0$, it is defined as

$$\eta(x, \lambda) = -f(x)^T \lambda$$

If x is primal feasible and λ, ν are dual feasible, which means that $r_{\text{pri}} = 0$ and $r_{\text{dual}} = 0$, then the surrogate gap η would be the duality gap.

The basic primal-dual interior point algorithm is taken from [12]:

The inputs are a point x that satisfies $f_1(x) < 0, \ldots, f_m(x) < 0, \quad \lambda > 0$, $\mu > 1, \ \ \epsilon_{feas} > 0$ and $\epsilon > 0$.

1. Set $t = \mu m / \eta$.

2. Compute primal-dual search direction $\Delta y_{pd} = (\Delta x_{pd}, \Delta \lambda_{pd}, \Delta \nu_{pd})$.

3. Line search and update.

   We determine the step length $s > 0$ and compute $y = y + s\Delta y_{pd}$ until $\|r_{pri}\|_2 \leq \epsilon_{feas}, \ \ \|r_{dual}\|_2 \leq \epsilon_{feas}$ and $\eta \leq \epsilon$.

Later in Chapter **6** we will see how we recover a sparse signal by using this method.

5.2 Orthogonal Matching Pursuit

Orthogonal Matching Pursuit (OMP) is a greedy algorithm. It is based on an algorithm called Matching Pursuit. OMP starts by finding the column of a matrix A that resembles the most with the residual, and then this column will be added into a set of selected columns. This process will repeat. The algorithm will update the residuals by projecting the vector b onto the space spanned by the selected columns in the set. After each step, the residuals are orthogonal to all the selected columns. This means that no column is chosen twice, and the set with the selected columns will increase after each step. The advantage with OMP is its fast implementation.

Algorithm:

The inputs are a vector $b \in \mathbb{R}^m$, a matrix $A \in \mathbb{R}^{m \times n}$ and a stopping criterion. The output is an approximation vector $x \in \mathbb{R}^n$.

1. Let the initial solution $x_0 = 0$, and set the residual $r_0 = b$.
   Set the iteration counter t = *1* and the index set $\Lambda_0 = \emptyset$.

2. Compute the inner product and choose the one with the largest magnitude, $\lambda_t = \arg\min_{j=1,\ldots,n} | < r_{t-1}, a_j > |$.

3. Update the index set, $\Lambda_t = \Lambda_{t-1} \cup \{\lambda_t\}$.

4. Compute the approximation, $(x_t)_{\Lambda_t} = A^{\dagger}_{\Lambda_t} b$,
   where $A^{\dagger}_{\Lambda_t} b$ is the pseudo inverse of $A_{\Lambda_t} b$.

5. Update the new residual, $r_t = b - Ax_t$.

6. Increase the iteration counter, t = t + *1*.

7. Check the stopping criterion. If the criterion is not satisfied, then return to step 2.

5.3 Stagewise Orthogonal Matching Pursuit

Stagewise Orthogonal Matching Pursuit (StOMP) is an efficient algorithm for finding sparse solution to large underdetermined problems. It is based on OMP. The algorithm runs faster than Basis Pursuit and OMP. Compared with OMP, which at each step adds only one vector into the set of selected columns, StOMP adds several vectors. The algorithm runs similar as OMP. The advantage of StOMP is that it uses a small number of iterations, and therefore the algorithm runs fast.

Algorithm:

The inputs are a vector $b \in \mathbb{R}^m$, a matrix $A \in \mathbb{R}^{m \times n}$, a threshold parameter s_t and a stopping criterion. The output is an approximation vector $x \in \mathbb{R}^n$.

1. Let the initial solution $x_0 = 0$, and set the residual $r_0 = b$.
   Set the iteration counter $t = 1$ and the index set $\Lambda_0 = \emptyset$.

2. Compute the inner product, $\lambda_t = A^T r_{t-1}$.
   Create a set J_t consisting of the vectors with large coordinates
   $$J_t = \{j : |\lambda_t(j)| > s_t\}.$$

3. Update the index set, $\Lambda_t = \Lambda_{t-1} \cup J_t$.

4. Project the vector b onto the space spanned by the selected columns

of A by computing the approximation

$$(x_t)_{\Lambda_t} = (A_{\Lambda_t}^T A_{\Lambda_t})^{-1} A_{\Lambda_t}^T b.$$

5. Update the new residual, $r_t = b - Ax_t$.

6. Increase the iteration counter, $t = t + 1$.

7. Check the stopping criterion. If the criterion is not satisfied, then return to step 2.

5.4 Regularized Orthogonal Matching Pursuit

Regularized Orthogonal Matching Pursuit (ROMP) is an iterative algorithm which is also based on OMP with some differences. The algorithm also selects many vectors at each iteration like StOMP, not like OMP, which at each step only selects one vector. Another difference is its regularized step which we will see in the algorithm.

Algorithm:

The inputs are a vector $b \in \mathbb{R}^m$, a matrix $A \in \mathbb{R}^{m\times n}$, a sparsity level s and a stopping criterion. The output is an approximation vector $x \in \mathbb{R}^n$.

1. Let the initial solution $x_0 = 0$, and set the residual $r_0 = b$.
   Set the iteration counter $t = 1$ and the index set $\Lambda_0 = \emptyset$.

2. Compute the inner product, $\lambda_t = A^T r_{t-1}$.
   Choose a set J of the s largest non-zero coordinates in the magnitude of the vector λ_t.

3. Regularize step. Choose a subset J_0 with the maximum $\|\lambda|_{J_0}\|_2$

among all the subsets $J_0 \subset J$ which satisfy $|\lambda(i)| \leq 2|\lambda(j)|$ for all $i, j \in J_0$.

4. Update the index set, $\Lambda_t = \Lambda_{t-1} \cup J_0$.

5. Project the vector b onto the space spanned by the selected columns of A by computing the approximation

$$(x_t)_{\Lambda_t} = (A^T_{\Lambda_t} A_{\Lambda_t})^{-1} A^T_{\Lambda_t} b.$$

6. Update the new residual, $r_t = b - Ax_t$.

7. Increase the iteration counter, $t = t + 1$.

   Check the stopping criterion. If the criterion is not satisfied, then return to step 2.

5.5 Compressive Sampling Matching Pursuit

Compressive Sampling Matching Pursuit (CoSaMP) is similar to StOMP and ROMP, it is based on OMP and it selects many vectors at each iteration. An approximation is made at each iteration by using the largest coordinates. The advantage with CoSaMP is that it works well when the samples are disrupted with noise.

Before we present the CoSaMP algorithm, we define the support of a vector x as a set of indices to the elements of x which are non-zeros.

$$\text{supp}(x) = \{i \colon x_i \neq 0\}$$

Algorithm:

The inputs are a sample vector b which are disrupted with noise, a matrix A, a sparsity level s and a stopping criterion. The output is an approximation vector x.

1. Let the initial solution $x_0 = 0$. Set the current sample $v = b$ and the iteration counter t = *0*.
2. Update the iteration counter, $t = t$ + *1*.
   Compute a vector $y = A^T v$.
   Choose out the largest components, $J = \text{supp}(y_{2s})$.
3. Set $\Lambda = J \cup \text{supp}(x_{t-1})$.
4. Estimate a vector u,
   $$u|_\Lambda = A_\Lambda^\dagger b$$
   $$u|_{\Lambda^c} = 0$$
5. Update the approximation, $x_t = b_s$.
   Update the current samples, $v = b - Ax_t$.
6. Check the stopping criterion. If the criterion is not satisfied, then return to step 2.

5.6 Iterative Hard Thresholding Algorithm

Iterative Hard Thresholding algorithm (IHT) is different from all the previous algorithms. It is a greedy algorithm. IHT solves a local approximation to the problem

$$\begin{aligned} \text{minimize}_x \quad & \|b - Ax\|_2^2 \\ \text{subject to} \quad & \|x\|_0 \leq k \end{aligned} \tag{5.11}$$

Instead of directly handle the problem (5.11), we introduce a surrogate objective function of it. Each x can then be optimized independently. By ignoring the constraint $\|x\|_0 \leq k$, the problem (5.11) has a minimizer,

$$x^\star = x + \mu A^T(b - Ax)$$

The algorithm uses a nonlinear operator, $H_s()$, that sets all but the largest s elements of its argument to zero.

Algorithm:

The inputs are a vector b, a matrix A, a step size μ, a sparsity level s and a stopping criterion. The output is an approximation vector x.

1. Let the initial solution $x_0 = 0$ and the iteration counter $t = 0$.
2. Update the iteration counter, $t = t + 1$.
3. Compute $x_t = H_s(x_{t-1} + \mu A^T(b - Ax_{t-1}))$.
4. Check the stopping criterion. If the criterion is not satisfied, then return to step 2.

6. IMPLEMENTATION AND COMPUTATIONAL RESULTS

In this chapter we will present our implementation and computational results. We want to solve the Basis Pursuit problem, so we start with recasting it as a linear programming problem. First we solve it by using the command *linprog* in Matlab [10] and then using the primal-dual interior point method [4]. Finally we use it on our coin example [6] and on image processing, [14, 15].

6.1 Solving Basis Pursuit

Basis Pursuit finds the best representation of an image or a signal by minimizing the l_1- norm of the components of x, that is the coefficients in the representation. We would like the components of x to be zero or as close to zero as possible.

We would like to solve

$$\begin{array}{ll} \text{minimize} & \|x\|_1 \\ \text{subject to} & Ax = b \end{array} \tag{6.1}$$

This problem can be recast as a linear programming problem (LP) of the form

$$\begin{aligned} \text{minimize} \quad & f^T x \\ \text{subject to} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{6.2}$$

where $f^T x$ is the objective function, $Ax = b$ is a collection of equality constraints, and $x \geq 0$ is a set of bounds.

Starting with problem (6.1), we have that

$$\|x\|_1 = \textstyle\sum_{i=1}^n |x_i|$$

We can then transfer the nonlinearities to the set of constraints by adding the new variables $u_1, \ldots, u_n$. This gives

$$\begin{aligned} \text{minimize} \quad & \textstyle\sum_{i=1}^n u_i \\ \text{subject to} \quad & -u \leq x \leq u \\ & Ax = b \end{aligned} \tag{6.3}$$

Rewriting this we get

$$\begin{aligned} \text{minimize} \quad & \textstyle\sum_{i=1}^n u_i \\ \text{subject to} \quad & x_i - u_i \leq 0, \quad i = 1, \ldots, n \\ & -x_i - u_i \leq 0, \quad i = 1, \ldots, n \\ & Ax = b \end{aligned}$$

By using identity matrices, we can write the problem above in matrix form as

$$\begin{aligned} \text{minimize} \quad & \begin{bmatrix} \mathbf{0} & \mathbf{1} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \\ \text{subject to} \quad & \begin{bmatrix} I & -I \\ -I & -I \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq \mathbf{0} \\ & \begin{bmatrix} A & \mathbf{0} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} = b \end{aligned}$$

Thus we have rewritten our problem as an LP, which is the same as problem (6.2), where $f^T = \begin{bmatrix} \mathbf{0} & \mathbf{1} \end{bmatrix}$.

For our Matlab code, we set

$$E = \begin{bmatrix} I & -I \\ -I & -I \end{bmatrix}, \qquad d = 0, \qquad C = \begin{bmatrix} A & \mathbf{0} \end{bmatrix}$$

Thus we have,

$$\begin{aligned} \text{minimize} \quad & f^T \begin{bmatrix} x \\ u \end{bmatrix} \\ \text{subject to} \quad & E \begin{bmatrix} x \\ u \end{bmatrix} \leq d \\ & C \begin{bmatrix} x \\ u \end{bmatrix} = b \end{aligned} \tag{6.4}$$

An alternative way for instead of rewriting the problem (6.3), we can directly solve it in Matlab by setting

$$lb = -u$$
$$ub = u$$

where *lb* is the lower bound and *ub* is the upper bound. Later in Section **6.4** we are using both ways to solve the problem for the coin example.

6.2 Solving Linear Programming Problems with Matlab

Matlab provides the command *linprog* to find the minimizer x of a linear programming minimum problem.

Let f be a column vector of length n, b a column vector of length m, and let A be an $m \times n$ matrix. A linear program problem may have inequality constraints or equality constraints.

A linear program problem associated with f, A, b, A_{eq}, b_{eq} is the minimization problem

$$\begin{aligned} \text{minimize} \quad & f^T x \\ \text{subject to} \quad & Ax \leq b \\ & A_{eq}x = b_{eq} \end{aligned}$$

where b_{eq} is a column vector of length p and A_{eq} is a $p \times n$ matrix.

For solving this problem, we use the command

$$\text{x} = \text{linprog(f, A, b, A}_{eq}\text{, b}_{eq}\text{)}$$

or

$$[\text{x,fval}] = \text{linprog}(\text{f, A, b, A}_{eq}, \text{b}_{eq})$$

The general form of calling *linprog* is

$$[\text{x,fval,exitflag,output,lambda}] = \text{linprog}(\text{f, A, b, A}_{eq}, \text{b}_{eq}, \text{lb, ub, x}_0, \text{options})$$

lb is the lower bound and *ub* is the upper bound. x_0 is a startvector for the algorithm. *options* are set using the optimset function, they determine what algorithm to use, for example Simplex method or Interior point method. If there are no inequality constraints, we can set A=[ ] and b=[ ]. If there are no equality constraints, we can set A_{eq}=[ ] and b_{eq}=[ ].

For the output arguments, we have that x is the optimal solution. *fval* is the optimal value of the objective function. *exitflag* tells whether the algorithm converges or not, $exitflag > 0$ means convergence. *output* shows the number of iterations and the algorithm being used. *lambda* shows the Lagrange multipliers corresponding to the constraints.

6.3 Recovery of Sparse Signals via Convex Programming

A sparse signal is a signal with few nonzero elements. Most of its entries are zeros. To recover a sparse signal x from a number of linear measurements $b = Ax$, we can solve a convex program. This convex program can be recast as a linear program like we have showed before.

In this section we will use the primal-dual interior point method, which was presented earlier in Chapter **5**. We are going to follow the steps in Section **5.1.2** for solving Basis Pursuit

$$\begin{aligned} &\text{minimize} && \|x\|_1 \\ &\text{subject to} && Ax = b \end{aligned}$$

in Matlab. In Section **5.1** we showed how Basis Pursuit can be recast as an LP problem

$$\begin{aligned} &\text{minimize}_{x,u} && \textstyle\sum_i u_i \\ &\text{subject to} && x_i - u_i \leq 0 \\ & && -x_i - u_i \leq 0 \\ & && Ax = b \end{aligned}$$

For our implementation in Matlab, we set

$$\begin{aligned} f_{u_1;i} &= x_i - u_i \\ f_{u_2;i} &= -x_i - u_i \end{aligned}$$

$\lambda_{u_1;i}$ and $\lambda_{u_2;i}$ are the corresponding dual variables,

$$\begin{aligned} \lambda_{u_1;i} &= -\tfrac{1}{f_{u_1;i}} \\ \lambda_{u_2;i} &= -\tfrac{1}{f_{u_2;i}} \end{aligned}$$

At a point $(x, u; \lambda_{u_1}, \lambda_{u_2}, \nu)$ we have that

$$r_{\text{dual}} = \begin{pmatrix} \lambda_{u_1} - \lambda_{u_2} + A^T\nu \\ \mathbf{1} - \lambda_{u_1} - \lambda_{u_2} \end{pmatrix}$$

$$r_{\text{cent}} = \begin{pmatrix} -\mathbf{diag}(\lambda_{u_1}) f_{u_1} \\ -\mathbf{diag}(\lambda_{u_2}) f_{u_2} \end{pmatrix} - \frac{1}{\tau}\mathbf{1}$$

$$r_{\text{pri}} = Ax - b$$

We have that

$$\nabla f_{u_1;i} = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \quad \nabla f_{u_2;i} = \begin{pmatrix} -1 \\ -1 \end{pmatrix}, \qquad \nabla^2 f_{u_1;i} = 0, \quad \nabla^2 f_{u_2;i} = 0$$

so using the core system from Section **5.1.2**, we get

$$\begin{pmatrix} D_1 & D_2 & A^T \\ D_2 & D_1 & 0 \\ A & 0 & 0 \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta u \\ \Delta \nu \end{pmatrix} = \begin{pmatrix} -\frac{1}{\tau} \cdot (-f_{u_1}^{-1} + f_{u_2}^{-1}) - A^T v \\ -\mathbf{1} - \frac{1}{\tau} \cdot (f_{u_1}^{-1} + f_{u_2}^{-1}) \\ b - Ax \end{pmatrix} \tag{6.5}$$

where

$$D_1 = -\mathbf{diag}(\lambda_{u_1})\mathbf{diag}(f_{u_1})^{-1} - \mathbf{diag}(\lambda_{u_2})\mathbf{diag}(f_{u_2})^{-1}$$
$$D_2 = \mathbf{diag}(\lambda_{u_1})\mathbf{diag}(f_{u_1})^{-1} - \mathbf{diag}(\lambda_{u_2})\mathbf{diag}(f_{u_2})^{-1}$$

In our code we set

$$\begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} -\frac{1}{\tau} \cdot (-f_{u_1}^{-1} + f_{u_2}^{-1}) - A^T v \\ -\mathbf{1} - \frac{1}{\tau} \cdot (f_{u_1}^{-1} + f_{u_2}^{-1}) \\ b - Ax \end{pmatrix}$$

From the first equation of (6.5), we have

$$D_1\Delta x + D_2\Delta u + A^T\Delta\nu = w_1$$

$$D_1\Delta x = w_1 - D_2\Delta u - A^T\Delta\nu$$

$$\Delta x = D_1^{-1}(w_1 - D_2\Delta u - A^T\Delta\nu)$$

From the second equation of (6.5), we have

$$D_2\Delta x + D_1\Delta u = w_2$$

$$D_1\Delta u = w_2 - D_2\Delta x$$

$$\Delta u = D_1^{-1}(w_2 - D_2\Delta x)$$

By substituting the second equation into the first equation, we get

$$\Delta x = D_1^{-1}(w_1 - D_2 D_1^{-1}(w_2 - D_2\Delta x) - A^T\Delta\nu)$$

$$D_1\Delta x = w_1 - D_2 D_1^{-1} w_2 + D_2^2 D_1^{-1}\Delta x - A^T\Delta\nu$$

$$\Delta x(D_1 - D_2^2 D_1^{-1}) = w_1 - D_2 D_1^{-1} w_2 - A^T\Delta\nu$$

We set

$$D_3 = D_1 - D_2^2 D_1^{-1}$$

Thus

$$\Delta x = D_3^{-1}(w_1 - D_2 D_1^{-1} w_2 - A^T\Delta\nu)$$

Using Δx in the third equation gives

$$A\Delta x = w_3$$

$$A D_3^{-1}(w_1 - D_2 D_1^{-1} w_2 - A^T\Delta\nu) = w_3$$

$$-AD_3^{-1}A^T\Delta\nu = w_3 - AD_3^{-1}(w_1 - D_2D_1^{-1}w_2)$$

$$\Delta\nu = (-AD_3^{-1}A^T)^{-1}(w_3 - A(D_3^{-1}w_1 - D_3^{-1}D_2D_1^{-1}w_2))$$

Now that we have $\Delta x, \Delta u$ and $\Delta\nu$, we can calculate the change for the dual variables like in Section **5.1.2**

$$\Delta\lambda_{u_1} = \mathbf{diag}(\lambda_{u_1})\mathbf{diag}(f_{u_1})^{-1}(-\Delta x + \Delta u) - \lambda_{u_1} - \tfrac{1}{\tau}f_{u_1}^{-1}$$

$$\Delta\lambda_{u_2} = \mathbf{diag}(\lambda_{u_2})\mathbf{diag}(f_{u_2})^{-1}(\Delta x + \Delta u) - \lambda_{u_2} - \tfrac{1}{\tau}f_{u_2}^{-1}$$

For the step length, we choose $0 < s \leq 1$. It is based on the norm of the residuals. We also have to make sure that the step is feasible, which means that $\lambda_{u_1},\ \lambda_{u_2} > 0$ and $f_{u_1},\ f_{u_2} < 0$. We start the backtracking line search with

$$\mathrm{s} = 0.99 \cdot \min\{1,\ \min\{-\tfrac{\lambda_i}{\Delta\lambda_i} \mid \Delta\lambda_i < 0\}\}$$

We multiply s by $\beta \in (0,1)$ until we have

$$\|r_\tau(x + s\Delta x,\ \lambda + s\Delta\lambda,\ \nu + s\Delta\nu)\|_2 \ \leq\ (1-\alpha s)\cdot\|r_\tau(x,\lambda,\nu)\|_2$$

α is usually chosen between *0.01* to *0.1.* In our code we are using the surrogate duality gap described in Section **5.1.2**, that is

$$\eta(x,\lambda) = -f(x)^T\lambda$$

The implementation for this problem is in the file pd.m, see Appendix **A**.

6.4 Coin Example

We are now going to present a coin example that is from [6]. Suppose we have 7 coins. We know that one of the coins is counterfeit, and this coin will have a different mass than the other coins. If we know how much a coin weighs, we can find out the counterfeit coin by using 7 weighs. The main point here is that it is possible to find out the counterfeit coin by using 3 weighs. First we denote the coins with numbers 1 to 7. For the first weighing, the coins 1, 3, 5 and 7 are on the scale. For the second weighing we have coins 2, 3, 6 and 7. For the third weighing we use coins 4, 5, 6 and 7. We can express these choices by using a 0-1 matrix Φ. The kth row shows which coins to include in the kth weighing.

$$\Phi = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

From the outcome of these three weighings we can find out any single bad coin. For example, if the first set's mass is different from the nominal value, then we know that coin #1 is the counterfeit.

coin #2 is the counterfeit: second set's mass deviates

coin #3 is the counterfeit: first and second set's mass deviate

coin #4 is the counterfeit: third set's mass deviates

coin #5 is the counterfeit: first and third set's mass deviate

coin #6 is the counterfeit: second and third set's mass deviate

coin #7 is the counterfeit: first, second and third set's mass deviate

The problem can be formulated as follows. The 0-1 matrix Φ is the sensing basis. We want to recover x from the given matrix Φ and the measurements b. We are therefore solving the Basis Pursuit problem

$$\begin{aligned} &\text{minimize} \quad \|x\|_1 \\ &\text{subject to} \;\; \Phi x = b \end{aligned}$$

We can solve this problem with Matlab. One way is to use (6.4) by setting $C = \begin{bmatrix} \Phi & \mathbf{0} \end{bmatrix}$. To run this in *linprog*, we use the command

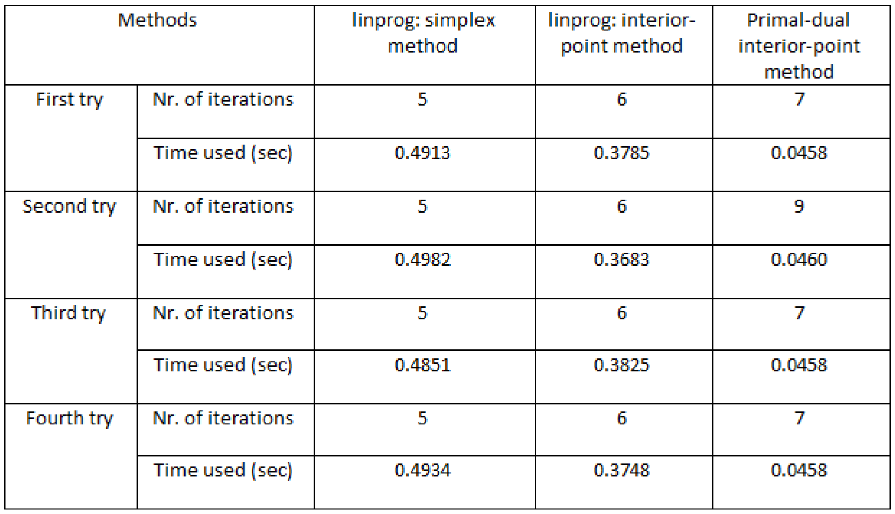
[x,fval,exitflag,output,lambda]=linprog(f, E, d, C, b, lb, ub, x_0, options)

Another way is to use (6.3). We then use the *linprog* command

[x,fval,exitflag,output,lambda]=linprog(f, [ ], [ ], A, b, lb, ub, x_0, options)

where $A = \Phi$, lb $= -u$ and ub $= u$ in (6.3).

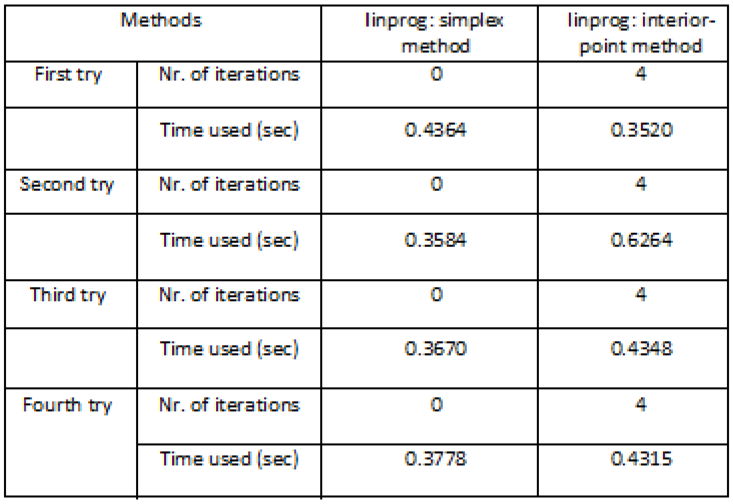
The coin example has been solved by *linprog* in Matlab using Simplex method and Interior point method as *options*. The number of iterations and the time used for running these two methods have been compared with the solution of this example solved by primal-dual interior point method described in Section **6.3**. There is not much difference when it comes to the number of iterations between the three methods. *linprog*'s Interior point method and Simplex method use almost the same time for solving the problem, just that the Interior point method runs a little faster. Among the three methods, the primal-dual method is the fastest as shown below in figure 6.1. It shows the results of the code after it runs four times, and here we use (6.4) in *linprog*.



Methods		linprog: simplex method	linprog: interior-point method	Primal-dual interior-point method
First try	Nr. of iterations	5	6	7
	Time used (sec)	0.4913	0.3785	0.0458
Second try	Nr. of iterations	5	6	9
	Time used (sec)	0.4982	0.3683	0.0460
Third try	Nr. of iterations	5	6	7
	Time used (sec)	0.4851	0.3825	0.0458
Fourth try	Nr. of iterations	5	6	7
	Time used (sec)	0.4934	0.3748	0.0458

Fig. 6.1: The results of the three algorithms, coin example, using (6.4)

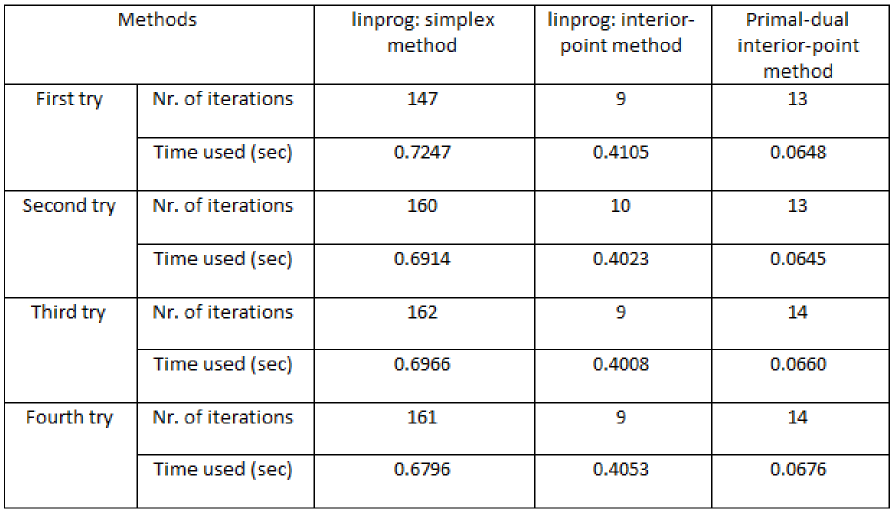
Figure 6.2 shows the results when using (6.3) in *linprog*. Comparing the results in this figure with the results above, we see that the time used is almost the same, but the methods used less iterations.



Methods		linprog: simplex method	linprog: interior-point method
First try	Nr. of iterations	0	4
	Time used (sec)	0.4364	0.3520
Second try	Nr. of iterations	0	4
	Time used (sec)	0.3584	0.6264
Third try	Nr. of iterations	0	4
	Time used (sec)	0.3670	0.4348
Fourth try	Nr. of iterations	0	4
	Time used (sec)	0.3778	0.4315

Fig. 6.2: The results of the three algorithms, coin example, using (6.3)

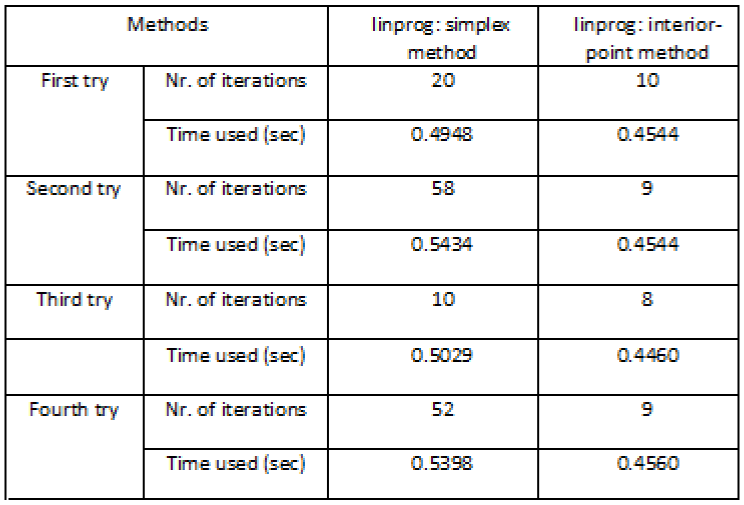
Now we make a 20×100 matrix Φ by choosing $n = 20$ random subsets of coins to weigh. Each entry is chosen randomly as 0 or 1. The component b_i of the vector $b = \Phi x$ shows the different mass from the nominal of the mass of the ith subset. From a given Φ and the measurements b, our goal is to recover x. Again, we solve this with *linprog*'s Interior point method and Simplex method, and primal-dual method. By comparing we still see that primal-dual method runs fastest among the three methods. When it comes to the number of iterations, *linprog*'s Simplex method uses the most iterations compared to the two other methods. This is shown in figure 6.3 where we use (6.4) in *linprog*.



Methods		linprog: simplex method	linprog: interior-point method	Primal-dual interior-point method
First try	Nr. of iterations	147	9	13
	Time used (sec)	0.7247	0.4105	0.0648
Second try	Nr. of iterations	160	10	13
	Time used (sec)	0.6914	0.4023	0.0645
Third try	Nr. of iterations	162	9	14
	Time used (sec)	0.6966	0.4008	0.0660
Fourth try	Nr. of iterations	161	9	14
	Time used (sec)	0.6796	0.4053	0.0676

Fig. 6.3: The results of the three algorithms, 20 x 100 matrix, using (6.4)

Below figure 6.4 shows the results when using (6.3) in *linprog*. Again we see that the methods are using less iterations, especially *linprog*'s Simplex method we can also see that the time used for solving is less.



Methods		linprog: simplex method	linprog: interior-point method
First try	Nr. of iterations	20	10
	Time used (sec)	0.4948	0.4544
Second try	Nr. of iterations	58	9
	Time used (sec)	0.5434	0.4544
Third try	Nr. of iterations	10	8
	Time used (sec)	0.5029	0.4460
Fourth try	Nr. of iterations	52	9
	Time used (sec)	0.5398	0.4560

Fig. 6.4: The results of the three algorithms, 20 x 100 matrix, using (6.3)

The implementation for this is in the file coinexample.m, see Appendix **B**.

6.5 Image Processing

By reducing the size in bytes of an image, it can for example allow us to store more images in a disk or reducing the time for when we send the images through internet. The most common compressed image formats are JPEG and JPEG-2000. The vectors in the image which represent the pixel sampling are being transformed. This means that it will be represented in a new coordinate system.

In JPEG, the discrete cosine transform (DCT) is used. It is a variant of Fourier transform (DFT). DFT transforms a sampling of a function into a combination of complex sinusoids. Instead of sinusoids, DCT transforms the sampling to cosine functions. The first transform coefficients are large, and the later ones are small and can therefore be seen as zeros. By approximating the first coefficients we will get a sequence which can be stored in a few bits. This sequence can then be inverse transformed by using IDCT to get back to the original representation.

In JPEG-2000, the discrete wavelet transform (DWT) is used. Similar to DCT, the small coefficients are seen as zeros. The large coefficients are approximated and then we get a sequence which can be stored. Again, to get back to the original representation, it can be inverse transformed by using IDWT. A different property of the DWT compared to the DCT is that there are few large coefficients, so the DWT of such content is more sparse than the DCT.

One technique of image compression is therefore to find sparse solutions to underdetermined systems of linear equations. Now we want to recover f from an underdetermined system $\Phi f = b$. f may not be sparse, but it could be that $f = \Psi x$ where Ψ is an $n \times n$ orthogonal matrix and x is a sparse vector. This means that f has a sparse representation in a basis spanned by the columns of Ψ. This leads to the system $\Phi\Psi x = b$, where we first recover x and then recover f.

For our image compression code, we denote f as our image, the matrix Φ is the sensing basis, the matrix Ψ is the representation basis and b is the compression of the image. We are going to solve the optimization problem

$$\begin{array}{ll} \text{minimize} & \|x\|_1 \\ \text{subject to} & Ax = b \end{array} \tag{6.6}$$

where $A = R\Phi\Psi$ and $b = R\Phi f$. R is a vector consisting of m random elements from a vector in $\mathbb{R}^n$.

We have from earlier that

$$f = \Psi x$$

In image processing, the vector x is a sparse vector for images. It is also a sparse vector for many wavelets. So we have that

$$x = DWT(f)$$

The problem (6.6) can be rewritten as

$$\begin{array}{ll} \text{minimize} & \|x\|_1 \\ \text{subject to} & R\Phi(DWT)^T x = R\Phi f \end{array}$$

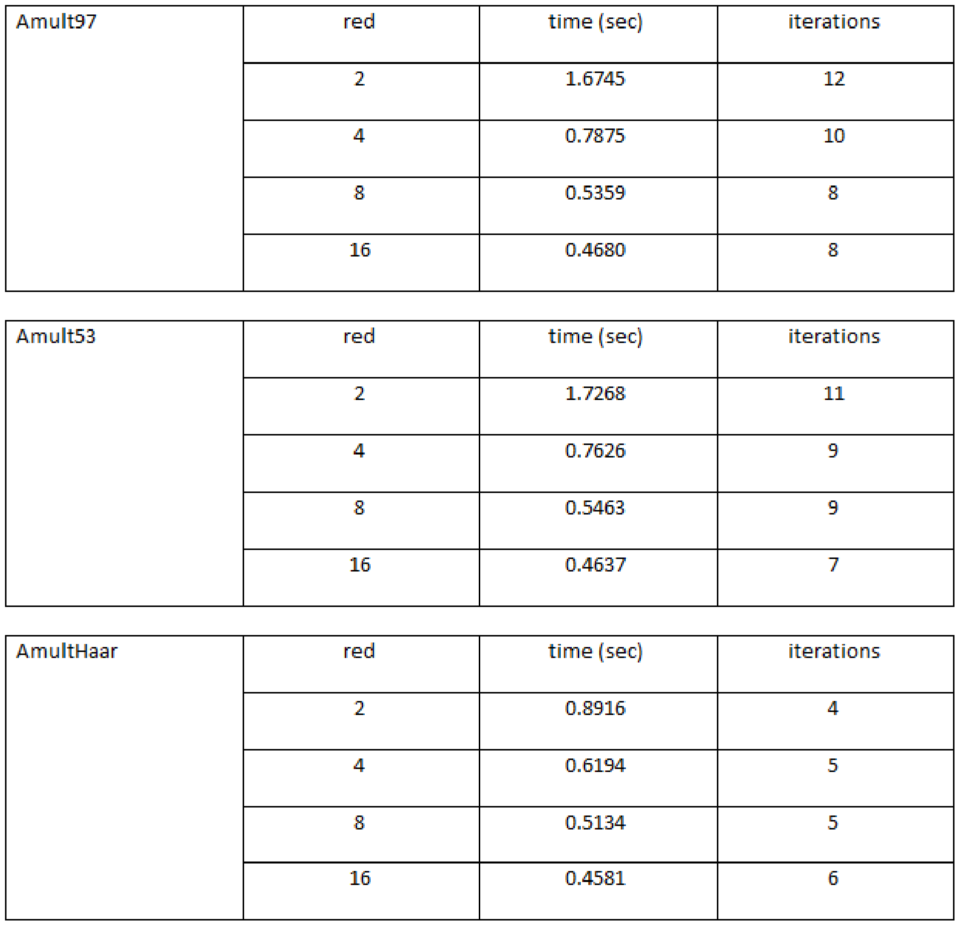
We solve this problem by first obtaining the solution x. Then the image can be found from $f = (DWT)^T x$.

The steps in our code are as follows. First we make our image f and take the size of it. The matrix of the image consists of non-zero coefficients in the left top corner, and the rest are zeros. It is a 16×16 matrix. The variable *red* in our code shows how many parts we want to divide the image in. If *red* is small, then the image is divided in many parts. If it is large, then we are working with few parts of the image.

We let Φ be a random matrix. We make a QR decomposition of Φ, and we get an orthogonal matrix Q and an upper triangular matrix R. We create the random vector R. Then we make the vector b by computing $b = R\Phi f$. Next we make the matrix A by computing $A = R\Phi(DWT)^T$. In this step we are using methods Amult53.m, Amult97.m and AmultHaar.m which are functions that do the multiplication between A and x. These three methods are using three different wavelets. They are Spline 5/3-wavelet, CDF 9/7-wavelet and Haar-wavelet. The numbers 5/3 and 9/7 correspond to the number of filter coefficients in the corresponding lowpass or highpass filters. The Spline 5/3-wavelet is a function approximation scheme based on piecewise linear functions. The Haar-wavelet is a function approximation scheme based on piecewise constant functions. By using the standard basis as input for x in these functions, we will get out the matrix A.

After we have computed A and b, we use *linprog* in Matlab as described in Section **6.2** to solve for x. Finally we compute the approximation $(DWT)^T x$ to get the image. The implementation of this code is in the file imagecomp.m, see Appendix **C**.

We test our code with *red = 2, 4, 8* and *16*. We also compare the results when we are using Amult53.m, Amult97.m and AmultHaar.m. Below figure 6.5 shows that the time used for solving are almost the same for all three methods. When the image is divided in few parts, the code runs faster. Method AmultHaar.m uses less iterations than the other two methods.



Amult97	red	time (sec)	iterations
	2	1.6745	12
	4	0.7875	10
	8	0.5359	8
	16	0.4680	8

Amult53	red	time (sec)	iterations
	2	1.7268	11
	4	0.7626	9
	8	0.5463	9
	16	0.4637	7

AmultHaar	red	time (sec)	iterations
	2	0.8916	4
	4	0.6194	5
	8	0.5134	5
	16	0.4581	6

Fig. 6.5: The results of the three methods with red = 2, 4, 8 and 16

Figure 6.6 shows the original image *f*. Figure 6.7 shows the images when using Amult97.m. We see that the image is the most clear when the image is divided in many parts, that is when *red* = *2*. Figure 6.8 shows the results for Amult53.m. For this method, all the images are very unclear. Figure 6.9 shows the results when using AmultHaar.m, and here we see that all the images are very clear. The wavelets in AmultHaar.m are orthogonal, while the wavelets in Amult53.m and Amult97.m are not. Among all the three methods, AmultHaar.m works best.

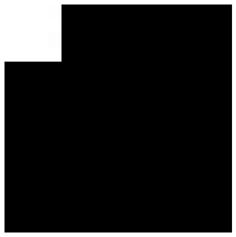


Fig. 6.6: The original image

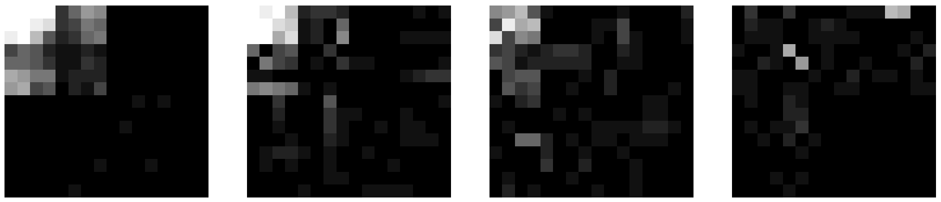


Fig. 6.7: Amult97, from left to right: red = 2, 4, 8 and 16

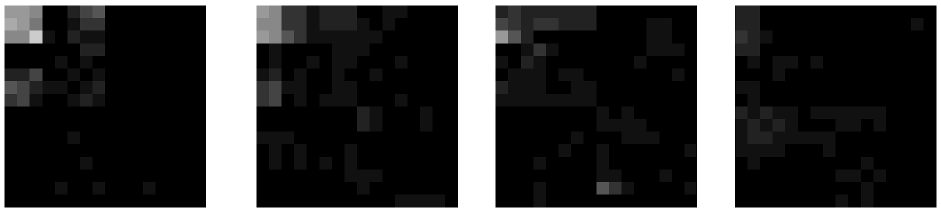


Fig. 6.8: Amult53, from left to right: red = 2, 4, 8 and 16

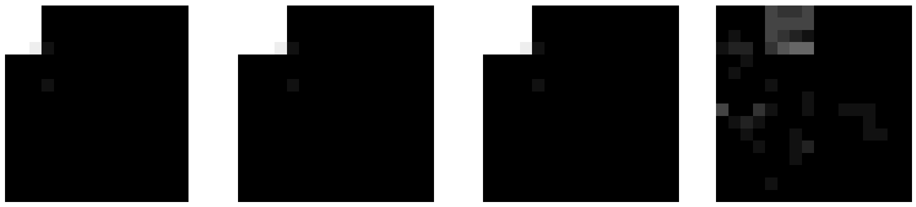


Fig. 6.9: AmultHaar, from left to right: red = 2, 4, 8 and 16

6.6 Discussion

In Section **6.1** we recast the Basis Pursuit problem to a linear programming problem, and we show two ways to solve it in Matlab. One way is to use (6.4) and the other way is to use (6.3). We use both ways for our coin example in Section **6.3** and see that by solving the problem using (6.3), the time the methods used for solving and the number of iterations are less compared to when solving the problem using (6.4). The reason for this is when we rewrite Basis Pursuit to a problem on the form (6.4), we are using identity matrices making the matrix larger. The problem (6.4) has also both inequality constraints and equality constraints which we use as inputs for the command *linprog*. By solving the problem on the form (6.3), we have an equality constraint, a lower bound $lb = -u$ and a upper bound $ub = u$. So if we solve a problem with larger matrices, it is more efficient to use (6.3) than (6.4).

In our code for image processing we are using a very small and simple image. The reason for doing this is that in our code we are using a full matrix expression. If we use an "advanced" image, the computation will take more time because the matrix will be much larger, so this will not work well with large images. To solve this we can instead of using the full matrix expression, we implement a function that compute the implementation with the DWT-matrix. With this we can then use a more ”advanced” image. So the use of full matrix expression in our code restricts us to the use of small and very simple images.

The idea behind the coin example and image processing is to use a sparse matrix, a matrix with few non-zero elements. In the coin example we use a 0-1 matrix Φ with rows as the weighings of the coins. The matrix is also chosen randomly as 0 or 1. In image processing, we let the sensing basis Φ be a random matrix, the representation basis Ψ correspond to three different wavelets and the image f consists of few non-zero coefficients. The results show that since the wavelets being used in method Amult97.m are close to being orthogonal, and the wavelets in method AmultHaar.m are orthogonal, they work better than method Amult53.m. The advantages with working with a sparse matrix are that the computation time is fast and the storage takes less space since the matrix contains a large number of zero-valued elements, and also we only need to know the indices and the value of the elements.

7. CONCLUSION

In this paper we have presented the background theory for optimization and approximation. We have discussed about the applications where the main goal is to find a sparse approximation. Compressed Sensing relies on the ℓ_1-norm optimization for reconstructing signals. We explained the conditions for the system we want to solve to have sparse recovery. We presented the methods for solving it, and also gave a brief description of greedy algorithms. Finally we tested our codes on a coin example and in image processing.

Optimization based on ℓ_1- norm for sparse recovery has a huge research area. Its applications are still expanding, and are connected with other areas such as physics, mechanics, biology, medical, informatics and economics. Some changes in the conditions for the system to have sparse recovery can lead to many other research problems. Algorithms for solving the problem are still being developed and finely adjusted. Compressed Sensing is an active area, especially now when the data technology is expanding. The storage of huge data and the computation time need compression. Key words as sparse, convex optimization, ℓ_1 minimization and compressed sensing are important tools for the engineers in the future.

Bibliography

[1] Cryptography, http://en.wikipedia.org/wiki/Cryptography

[2] D. L. Donoho, Y. Tsaig, I. Drori, J-L. Starck, *Sparse Solution of Underdetermined Linear Equations by Stagewise Orthogonal Matching Pursuit.* Stanford University, March 2006.

[3] D. Needell and R. Vershynin, *Signal Recovery From Incomplete and Inaccurate Measurements via Regularized Orthogonal Matching Pursuit.* Selected Topics in Signal Processing, IEEE Journal, April 2010.

[4] E. Candes and J.Romberg, ℓ_1*MAGIC: Recovery of Sparse Signals via Convex Programming.* Caltech, October 2005.

[5] E. J. Candes and T. Tao, *Decoding by Linear Programming.* University of California, Los Angeles, December 2004.

[6] K. Bryan and T. Leise, *Making Do with Less: An Introduction to Compressed Sensing.* Society for Industrial and Applied Mathematics, 2013.

[7] M. Elad, J-L. Starck, P. Querre, D. L. Donoho, *Simultaneous cartoon and texture image inpainting using morphological component analysis (MCA).* Journal on Applied and Computational Harmonic Analysis ACHA Vol.19, 2005.

[8] M. Elad and M. Aharon, *Image Denoising Via Sparse and Redundant Representations Over Learned Dictionaries.* IEEE transactions on image processing Vol.15 No.12, December 2006.

[9] M. F. Duarte and M. A. Davenport, *Null Space Conditions.* produced by The Connexions Project and licensed under the Creative Commons Attribution License, April 2011.

[10] Matlab linprog, http://www.mathworks.se/help/optim/ug/linprog.html

[11] R. J. Vanderbei. *Linear Programming: Foundations and Extensions.* Princeton University, Princeton, 2001.

[12] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[13] Y. C. Eldar and G. Kutyniok. *Compressed Sensing, Theory and Applications.* Cambridge University Press, 2012.

[14] Ø. Ryan. *Compressed sensing and image compression.* `http://folk.uio.no/oyvindry/cs.pdf`.

[15] Ø. Ryan. *Linear algebra, signal processing, and wavelets. A unified approach.* University of Oslo, Manuscript under preparation.

APPENDIX

A. Matlab Code pd.m

This code is for solving the problem described in Section **6.3**.

```
function xpd = pd(x0, A, b)

epsilon = 1e-3;  % tolerance
maxiter = 50;    % maximum iterations

alpha = 0.01;
beta = 0.5;
mu = 10;

n = length(x0);
gradf0 = [zeros(n,1); ones(n,1)];

x = x0;
u = (0.95)*abs(x0) + (0.10)*max(abs(x0));

% first iteration
fu1 = x - u;
fu2 = -x - u;

% lambda: the corresponding dual variables
lamu1 = -1./fu1;
lamu2 = -1./fu2;

v = -A*(lamu1-lamu2);
Atv = A'*v;
```

```
% surrogate duality gap
eta = -(fu1'*lamu1 + fu2'*lamu2);
tau = mu*2*n/eta;

% dual residual
rdual = gradf0 + [lamu1-lamu2; -lamu1-lamu2] +
         [Atv; zeros(n,1)];
% central residual
rcent = [-lamu1.*fu1; -lamu2.*fu2] - (1/tau);
% primal residual
rpri = A*x - b;
% norm of the residuals
resnorm = norm([rdual; rcent; rpri]);

iter = 0;
done = (eta < epsilon) | (iter >= maxiter);
while (~done)

   iter = iter + 1;

   D1 = -lamu1./fu1 - lamu2./fu2;
   D2 = lamu1./fu1 - lamu2./fu2;
   D3 = D1 - D2.^2./D1;

   w1 = -1/tau*(-1./fu1 + 1./fu2) - Atv;
   w2 = -1 - 1/tau*(1./fu1 + 1./fu2);
   w3 = -rpri;

   % Solving for dv
   rside = -(w3 - A*(w1./D3 - w2.*D2./(D3.*D1)));
   lside = A*(sparse(diag(1./D3))*A');
   dv = inv(lside)*rside;

   dx = (w1 - w2.*D2./D1 - A'*dv)./D3;

   Atdv = A'*dv;

du = (w2 - D2.*dx)./D1;
```

```
% calculating the change in the inequality dual
  variables
dlamu1 = (lamu1./fu1).*(-dx+du) - lamu1 -
         (1/tau)*1./fu1;
dlamu2 = (lamu2./fu2).*(dx+du) - lamu2 - 1/tau*1./fu2;

% make sure that the step is feasible: keeps
  lamu1,lamu2 > 0, fu1,fu2 < 0
negind1 = find(dlamu1 < 0);
negind2 = find(dlamu2 < 0);
s = min([1; -lamu1(negind1)./dlamu1(negind1);
    -lamu2(negind2)./dlamu2(negind2)]);

posind1 = find((dx-du) > 0);
posind2 = find((-dx-du) > 0);
s = (0.99)*min([s;
  -fu1(posind1)./(dx(posind1)-du(posind1));
  -fu2(posind2)./(-dx(posind2)-du(posind2))]);

% using backtracking line search
normdec = 0;
backiter = 0;

while (~normdec)

    xpd = x + s*dx;
    upd = u + s*du;
    vpd = v + s*dv;

    Atvpd = Atv + s*Atdv;

    lamu1pd = lamu1 + s*dlamu1;
    lamu2pd = lamu2 + s*dlamu2;

    fu1pd = xpd - upd;
    fu2pd = -xpd - upd;

    rdualpd = gradf0 + [lamu1pd-lamu2pd;
              -lamu1pd-lamu2pd] + [Atvpd; zeros(n,1)];
    rcentpd = [-lamu1pd.*fu1pd; -lamu2pd.*fu2pd]
```

```
                - (1/tau);
    rpripd = rpri + s*A*dx;

    normdec = (norm([rdualpd; rcentpd; rpripd]) <=
       (1-alpha*s)*resnorm);
    s = beta*s;
    backiter = backiter + 1;

end

% next iteration
x = xpd;
u = upd;
v = vpd;
Atv = Atvpd;

lamu1 = lamu1pd;
lamu2 = lamu2pd;

fu1 = fu1pd;
fu2 = fu2pd;

  % surrogate duality gap
  eta = -(fu1'*lamu1 + fu2'*lamu2);
  tau = mu*2*n/eta;

  rpri = rpripd;
  rcent = [-lamu1.*fu1; -lamu2.*fu2] - (1/tau);
  rdual = gradf0 + [lamu1-lamu2; -lamu1-lamu2]
          + [Atv; zeros(n,1)];
  resnorm = norm([rdual; rcent; rpri]);

  done = (eta < epsilon) | (iter >= maxiter);

disp(sprintf('Iterations = %d, tau = %8.3e,
     Primal = %8.3e, PDGap = %8.3e, Dual res = %8.3e,
     Primal res = %8.3e', iter, tau, sum(u), eta,
     norm(rdual), norm(rpri)));

end
```

B. Matlab Code coinexample.m

This code is for solving the problem described in Section **6.4**.

```
function coinexample(m, n)
    % m - number of rows
    % n - number of columns

    if m == 3 && n == 7
        % Matrix for coin example
        A = [1 0 1 0 1 0 1; 0 1 1 0 0 1 1;
             0 0 0 1 1 1 1];
        xsol = [0; -0.18; 0; 0; 0; 0; 0];
        b = A*xsol;

    else
        % A 20x100 matrix:
        % randerr(m,n,ones) makes an mxn matrix with
        % fixed number of "1" in each row
        A = randerr(m,n,randi([1 n]));
        b = randi([0 1],m,1);

    end

    I = eye(n);
    E = [I -I; -I -I];
    d = zeros(2*n,1);

    C = [A zeros(m,n)];
    x0 = rand(2*n,1);
    f = [zeros(n,1); ones(n,1)];

    % lower and upper bounds
    for i=1:2*n
        lb(i) = -Inf;
        ub(i) = Inf;
    end
    lb = lb';
    ub = ub';

    % linprog_Interior point method:
    disp(sprintf('Linprog: Interior point method'))
    options = optimset('MaxIter',3000,'TolFun',1e-3);
    tic
    [x_ip,fval,exitflag,output,lambda] = linprog(f,E,
     d,C,b,lb,ub,x0,options)
    toc
    x_ip(1:n);

    % linprog_Simplex method:
    disp(sprintf('Linprog: Simplex method'))
    options = optimset('LargeScale','off','Simplex',
     'on');
    tic
    [x_s,fval,exitflag,output,lambda] = linprog(f,E,
     d,C,b,lb,ub,x0,options)
    toc
    x_s(1:n);

    % primal-dual interior method from pd.m:
    disp(sprintf('Primal-dual interior-point method'))
    tic
    x_pd = pd(x0, C, b)
    toc
    x_pd(1:n);
    obj = sum(x_pd(1:n));

end
```

C. Matlab Code imagecomp.m

This code is for solving the problem described in Section **6.5**.

```
m = 2;
red = 2;     % red = 2,4,8,16

% create image, a 16x16 matrix
f = zeros(16);
f(1:4, 1:4) = 255*ones(4);
N = size(f,1);

% Phi
phi=rand(N);
% upper triangular matrix R and unitary matrix Q
[Q,R]=qr(phi);
% random column vector R
Rvect = randsample(1:(N^2), N^2/red);

% Q and Rvect together describe the matrix Phi.
Y = Q*f*Q';          % Phi*f:
y = mattovec(Y)';
% y = R*Phi*f
y = y(Rvect);     % this is the compression of the
                    image

% methods used: Amult97, Amult53, AmultHaar
opA=@(x,mode) Amult97( Q,Rvect,m,N,x,mode);

% standard basis as input to get out matrix A
A = zeros(N^2/red,N^2);
for i=1:N^2
    e_n = zeros(N^2,1);
    e_n(i) = 1;

    A(:,i) = opA(e_n,1);
end
```

```
% Create inputs for using linprog: interior point
  method
C = [A zeros(N^2/red,N^2)];
x0 = rand(N^2,1);

I = eye(N^2);
E = [I -I; -I -I];
d = zeros(2*N^2,1);

f = [zeros(N^2,1); ones(N^2,1)];

% lower and upper bounds
for i=1:2*N^2
    lb(i) = -Inf;
    ub(i) = Inf;
end
lb = lb';
ub = ub';

disp(sprintf('Linprog: Interior point method'))
options = optimset('MaxIter',3000,'TolFun',1e-3);
tic
[x,fval,exitflag,output,lambda] = linprog(f,E,d,C,y,
   lb,ub,x0,options)
toc
x = x(1:N^2);

% now having x, computing back the image
X = vectomat(x',N,N);
frec = DWT2Impl97transpose(X,m);
%frec = DWT2Impl53transpose(X,m);
%frec = IDWT2HaarImpl(X,m);
imwrite(uint8(frec),'image1616.jpg','jpg');
```