Algorithms for Sparse Approximation

Philip Breen

Year 4 Project
School of Mathematics
University of Edinburgh
2009

Abstract

In recent years there has been a growing interest in sparse approximations. This is due to their vast amount of applications. The task of finding sparse approximations can be very difficult this is because there is no general method guaranteed to work in every situation. In fact, in certain cases there are no efficient methods for finding sparse approximations. This project considers the problems of finding sparse approximations and then examines the two most commonly used algorithms the *Lasso* and *Orthogonal matching pursuit*. Then goes on to discuss some practical applications of sparse approximations.

This project report is submitted in partial fulfilment of the requirements for the degree of *BSc Mathematics*.

Contents

Chapter 1

Introduction

What is a sparse approximation? Before this question can be answered let us first consider the term sparse. The term sparse refers to a measurable property of a vector. It means that the vector is in a sense small but it is not the length of the vector that is important. Instead sparsity concerns the number of non-zero entries in the vector. Here the L_0 is used to measure the sparsity of a vector.

There are a lot of advantages working with sparse vectors. For example calculations involving multiplying a vector by a matrix take less time to compute in general if the vector is sparse. Also sparse vectors require less space when being stored on a computer as only the position and value of the entries need to be recorded.

Definition 1. The L_0 norm of an n-dimensional vector $\mathbf{x}$ is
$$\|\mathbf{x}\|_0 = \#\{k\colon x_k \neq 0, k = 1, \ldots, n\}.$$

Informally the L_0 norm is simply the number of non-zero entries in a vector. The notation $\|\mathbf{x}\|_0$ will be used to represent the L_0 of $\mathbf{x}$. For example L_0 norm of $(0,2,0,0)^T$ is 1 and $(0,2,2,5)^T$ is 3. If there are two n-dimensional vectors $\mathbf{a}$ and $\mathbf{b}$ then $\mathbf{a}$ is sparser then $\mathbf{b}$ if and only if $\|\mathbf{a}\|_0 < \|\mathbf{b}\|_0$.

The L_1 and L_2 norms will be also used in later sections of this report. The notation $\|\cdot\|_1$ and $\|\cdot\|_2$ will be used for the L_1 and L_2 norms respectively. The L_1 norm is the sum of the absolute value of the entries in a vector. The L_2 is the Euclidean length of a vector. Below is the formal definition of the norms.

Definition 2. Let $\mathbf{x}$ be of an n-dimensional vector,
$$\text{than } \|\mathbf{x}\|_2 = \sum_{i=1}^{n} |x_i| \text{ and } \|\mathbf{x}\|_2 = \Big(\sum_{i=1}^{n} |x_i|^2\Big)^{1/2}$$

1.1 Sparse Approximation

Consider the following problem :

Problem 1.1 Given a vector $\mathbf{b} \in \mathbb{C}^m$ and a matrix $\mathbf{A} \in \mathbb{C}^{m+n}$ find a vector $\mathbf{x} \in \mathbb{C}^n$ such that

$$\mathbf{Ax} = \mathbf{b}$$

It is generally the case that the matrix $\mathbf{A}$ is over complete ($m \ll n$). Therefore the above problem does not have a unique solution. Since there exists a choice of solution a sparse vector is more desirable because of its advantages. In the above problem if a sparse vector $\mathbf{x}$ is found it is called a sparse representation of $\mathbf{b}$. This is because $\mathbf{x}$ can be used to reproduce $\mathbf{b}$ exactly. The original question "what is a sparse approximation" can now be answered by a more general version of Problem 1.1.

Problem 1.2 With $\mathbf{A}$ and $\mathbf{b}$ define as above, $\epsilon \in \mathbb{R}$ *epsilon* > 0 find $\mathbf{x}$ such that

$$\|\mathbf{Ax} - \mathbf{b}\| < \epsilon$$

If a sparse vector $\mathbf{x}$ is a solution to the above problem $\mathbf{x}$ is a sparse approximation of $\mathbf{b}$. In this problem there is more flexibility of choice for x since it is no longer required to reproduce $\mathbf{b}$ exactly. For example if $\mathbf{x}$ is a representation the smaller entries in $\mathbf{x}$ be set to 0 to produce a sparser approximation. Also there is a wider range of matrices $\mathbf{A}$ that can be used in the above problem. This is due to the fact that it is no longer required that $\mathbf{b} \in span(\mathbf{A})$ for solutions to exist.

Example of sparse representation and sparse approximation using a randomly generated vector and matrix.

$$\mathbf{1)} \begin{pmatrix} 0.9593 & 0.2575 & 0.2435 & 0.2511 & 0.8308 \\ 0.5472 & 0.8407 & 0.9293 & 0.6160 & 0.5853 \\ 0.1386 & 0.2543 & 0.3500 & 0.4733 & 0.5497 \\ 0.1493 & 0.8143 & 0.1966 & 0.3517 & 0.9172 \end{pmatrix} \begin{pmatrix} 0 \\ 0.7537 \\ 0.1285 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.2254 \\ 0.7531 \\ 0.2366 \\ 0.6390 \end{pmatrix}$$

$$\textbf{2)}\begin{pmatrix} 0.9593 & 0.2575 & 0.2435 & 0.2511 & 0.8308 \\ 0.5472 & 0.8407 & 0.9293 & 0.6160 & 0.5853 \\ 0.1386 & 0.2543 & 0.3500 & 0.4733 & 0.5497 \\ 0.1493 & 0.8143 & 0.1966 & 0.3517 & 0.9172 \end{pmatrix}\begin{pmatrix} 0 \\ 0.7537 \\ 0 \\ 0 \\ 0 \end{pmatrix} \approx \begin{pmatrix} 0.2254 \\ 0.7531 \\ 0.2366 \\ 0.6390 \end{pmatrix}$$

The error in the second example is just 0.1338, which is small in comparison to the length of the original vector 1.0403.

1.2 Uniqueness of sparsest approximation

In general sparsest approximations are not unique but there are conditions in which they can be. Examples of these conditions will be given later. Here the condition for uniqueness of sparse representation will be given.

Theorem 3. *A sparse representation* $\boldsymbol{x}$ *of* $\boldsymbol{b}$ *is unique if* $\|\boldsymbol{x}\|_0 < \dfrac{Spark(\boldsymbol{A})}{2}$

Where $\mathbf{A}, \mathbf{x}$ and $\mathbf{b}$ are defined as in section 1.2 . $Spark(\mathbf{A})$ is defined the size of the smallest set of linearly dependant vectors of $\mathbf{A}$. The $Spark(\mathbf{A})$ can also be defined as the value of the L_0 of sparsest non-zero vector such that $\mathbf{Ac} = \mathbf{0}$.

This theorem can easily be proved as follows :

Proof. Assume $\exists\ \mathbf{x_1}, \mathbf{x_2}$ sparse representation of $\mathbf{b}$ where $\mathbf{x_1} \neq \mathbf{x_2}$ and $\|\mathbf{x_1}\|_0, \|\mathbf{x_2}\|_0 < \dfrac{Spark(\mathbf{A})}{2}$.

$$\begin{aligned} \mathbf{Ax_1} &= \mathbf{Ax_2} = \mathbf{b} \\ \Rightarrow \mathbf{Ax_1} - \mathbf{Ax_2} &= \mathbf{0} \\ \Rightarrow \mathbf{A}(\mathbf{x_1} - \mathbf{x_2}) &= \mathbf{0} \\ \Rightarrow \|\mathbf{x_1} - \mathbf{x_2}\| &< Spark(\mathbf{A}) \end{aligned}$$

which contradicts the minimality of $Spark(\mathbf{A})$

□

1.3 Finding sparse approximation

Finding a sparse approximation is far from being straight forward. This is because there are an infinite amount of solutions. For the case of sparse representation all possible solutions can be built as follows.

Theorem 4. *All possible representation are of the form*

$$\mathbf{x} + \sum_i \lambda_i \mathbf{y_i}$$

,

where x is a chosen solution, $\lambda_i \in \mathbb{R}$ and the vectors $\mathbf{y_i}$ form a basis for $kernel(A)$.

Proof. For any vector $\mathbf{x}^*$ such that $\mathbf{A}\mathbf{x}^* = \mathbf{b}$

$$\begin{aligned} &\Rightarrow \mathbf{A}\mathbf{x}^* = \mathbf{A}\mathbf{x} \\ &\Rightarrow \mathbf{A}(\mathbf{x}^* - \mathbf{x}) = \mathbf{0} \\ &\Rightarrow \mathbf{x}^* = \mathbf{x} + (\mathbf{x}^* - \mathbf{x}) \end{aligned}$$

which is of the above form as $\mathbf{x}^* - \mathbf{x} \in Kenrel(\mathbf{A})$

□

In the case of sparse approximation it is very similar. If a solution x of $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 < \epsilon$ is chosen then different solutions can be generated by using the above equation. The important difference for approximations is that the above method will not generate all possible solutions.

Finding the sparsest solution out of this infinite set is very difficult. To clarify this consider the zero subset sum problem. Given a set of integers does there exist a subset where the elements of the subset add up to zero? If the set only has four elements then it would be easy to find a solution or whether one exits by checking all possible subsets. If the set is large, for example consisting of a thousand elements, then there are approximately 10^{301} possible subsets. It would take a lot of effort to find a solution. To make things more difficult lets add the condition that not only do the elements of the subset need to add up to zero but the subset needs to be the of smallest possible size. The sparse problem can be thought of as a multidimensional version of the zero subset problem. There is no method that will work in general for these types of problems.

1.4 Importance of Sparse Approximation

Finding a sparse approximation is more than just an abstract mathematical problem. Sparse approximations have a wide range of practical applications. Vectors are often used to represent large amounts of data which can be difficult to store or transmit. By using a sparse approximation the amount of space needed to store the vector would be reduced to a fraction of what was originally needed. Sparse approximations can also be used to analyze data by showing how column vectors in a given basis come together to produce the data.

There are many areas of science and technology which have greatly benefited from advances involving sparse approximations. In the application section of this report it will be shown how sparse approximations are used in inpainting, denoising, feature extraction and gene micro array analysis. The very useful nature of sparse approximations have prompted much interest and research in recent years and there is no doubt that sparse approximations will be of great interest in the coming years also.

Chapter 2

Algorithms

2.1 Introduction

There are many different methods used to solve sparse approximation problems but by far the two most common methods in use are the lasso and orthogonal matching pursuit. The lasso replaces the sparse approximation problem by a convex problem. One of the motivations for change to a convex problems is there are algorithms with can effectively find solutions. Orthogonal matching pursuit is a greedy method for solving the sparse approximation problem. This method is very straight forward as the approximation is generated by going through an iteration process. During each iteration the column vectors which most closely resemble the required vectors are chosen. These vectors are then used to build the solution.

2.2 The Lasso and Basis Pursuit

First a method known as basis pursuit with is related to the Lasso will be considered. The idea of Basis Pursuit is to replace the difficult sparse problem with an easier optimization problem. Below the formal definition of the sparse problem is given:

$$\textbf{Sparse problem} : min\|\mathbf{x}\|_0 \text{ subject to } \mathbf{Ax} = \mathbf{b} \tag{2.1}$$

The difficultly with the above problem is the L_0 norm. Basis Pursuit replaces the L_0 norm with the L_1 to make the problem easier to work with.

$$\textbf{Basis Pursuit}: min\|\mathbf{x}\|_1 \text{ subject to } \mathbf{Ax} = \mathbf{b} \tag{2.2}$$

The solution to the above problem can be found with relative ease. There are methods that will find the solution to the BP problem but does it lead to a sparse solution? The answer in general is no but under the right conditions it can be guaranteed that B will find a sparse solution or even the sparsest solution. This is because L_1 norm is only concerned with the value of entries not the quantity. A vector with a small L_1 could have very small valued non zero entries in every position which would give it a large L_0 norm.

The Lasso is similar to BP and is in fact known as Basis Pursuit De-Noising (BPDN) in some areas. The Lasso rather then trying to minimize the L_1 norm like BP places a restriction on its value.

$$\textbf{The Lasso} : \ \min\|\mathbf{Ax} - \mathbf{b}\|_2 \text{ subject to } \|\mathbf{x}\|_1 < \lambda$$

The Lasso allow us to find approximations rather then just representations and like BP can be guaranteed to find the sparsest solution under the right conditions.

2.2.1 Algorithms

The Lasso is an optimization principle rather than an algorithm. There are numerous algorithms to solve the problems above involving the L_1 norm. In this report the Matlab software package `CVX`, a package for solving convex problems [1, 2], is used to solve the optimization problem. According to S. S. Chen, D. L. Donoho, and M. A. Saunders [3] the simplex and interior-point methods offer an interesting insight into these optimization problems and will be introduced here.

The simplex method is based on a variant of Dantzig's simplex algorithm. Created by the American mathematician George Dantzig in 1947, the Dantzig simplex algorithm is so useful that in 2000 the journal *Computing in Science and Engineering* [4] listed it as one of the top 10 algorithms of the century. The standard simplex method starts by forming a new matrix $\mathbf{A}^*$ consisting of linearly independent columns of $\mathbf{A}$. Since all the columns of $\mathbf{A}^*$ are independent $\mathbf{b}$ can be uniquely represented (or approximated) with respect to $\mathbf{A}^*$. Then an iteration process takes place, where at each iteration a column vector of $\mathbf{A}^*$ is swapped with a column vector of $\mathbf{A}$. Each swap improves the desired property of the solution. In this case a reduction of the value of the L_1 norm.

The interior method starts with a solution $\mathbf{x_0}$ where $\mathbf{Ax_0} = \mathbf{b}$. Then goes through an iteration process changing the entries in $\mathbf{x_{k-1}}$ to form a new solution $\mathbf{x_k}$ while maintaining the condition $Ax_k = b$. A transformation is then applied to $\mathbf{x_k}$ which effectively sparsifies $\mathbf{x_k}$. Eventually a vector is reached that meets the preset stopping conditions and by forcing all extreme small entries to zero the

final solution is obtained. It should be noted that there are some more general interior point methods which do not require the condition $\mathbf{Ax_k} = \mathbf{b}$ to be met during the iteration process as long as the final solution meets the condition.

2.2.2 Properties of the Lasso and Basis Pursuit

The following graph 2.1 produced by the Matlab code *GLasso* with input $s = [0\colon 0.01\colon 1], n = 20$, shows how the L_0 norm of the approximation changes with changes in value of λ.

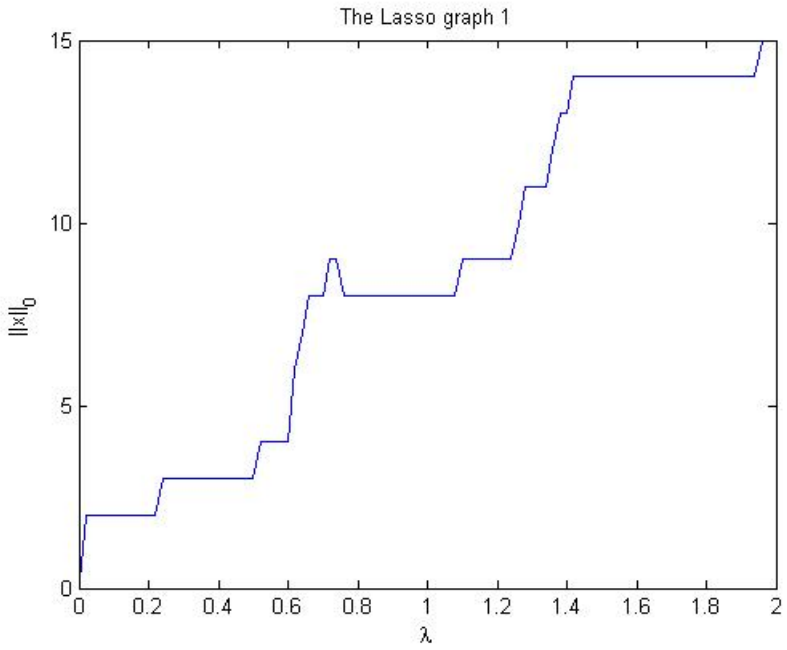

Figure 2.1: GLasso.m output 1

The most interesting feature in Figure 2.1 is the step like pattern that appears. The value of the L_0 norm corresponds to the number of column vectors of $\mathbf{A}$ which are used to construct the approximation. As the value of λ increases the entries of the approximation increase in size until an optimum position is reached and more column vectors are needed to improve the approximation. It would make sense to add the column vector which would improve the approximation the most. It is usually the case that several vectors would equally improve the approximation or that adding a group of vectors would improve the approximation better then any single vector. These steps are very important when trying to find sparse approximations as can be seen from Figure 2.2.

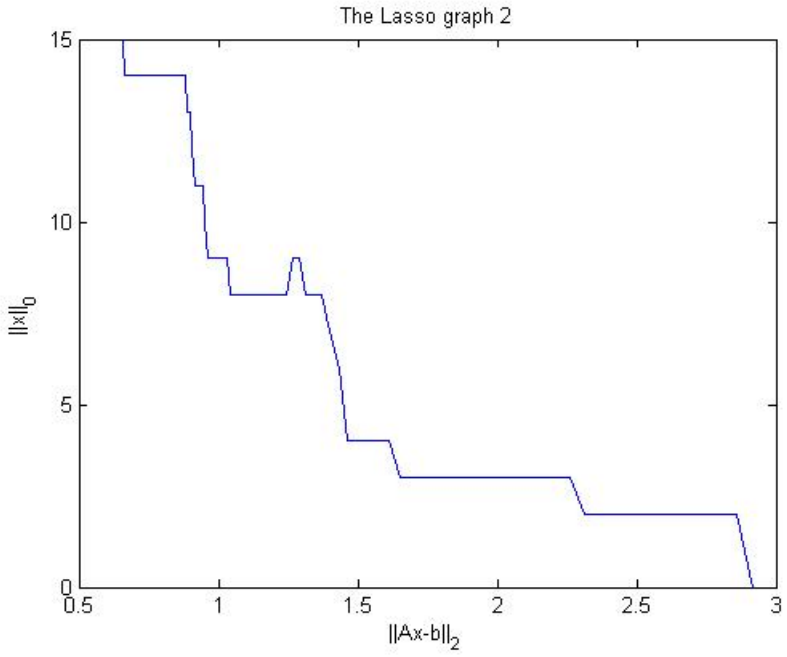

Figure 2.2: GLasso.m output 2

Figure 2.2 compares the sparsity of the approximation with it's accuracy measured using the L_2 norm. If the L_0 norm of an approximation $\mathbf{x}$ is required to be not more than 7, then it would be a good idea to take the optimum approximation where $\|\mathbf{x}\|_0 = 4$ because there is little accuracy gained by adding additional entries. This is why the step like feature makes the Lasso a very desirable method to use when dealing with sparse approximations.

Another interesting feature is that when the value of λ is around 0.7 the value of the L_0 decreases. This is because the L_1 is being used not the L_0 and an approximation is found with fewer but larger entries. Therefore in general an approximation would have to be calculated for every possible value of λ in order to confirm that the best approximation has been found for a given level of sparsity. It is possible because of the differences betweens the L_0 and L_1 norm that the Lasso will not find the sparsest possible approximation.

Can the BP find the sparsest representation? There is a result developed by Donoho and Huo [5] which guarantees under certain conditions that BP will find the sparsest possible representation. First the coherence μ will be define as it is need for the theorem.

Definition 5. The coherence of a matrix μ is the maximum absolute value of the inner product between any two different column vectors $\mathbf{a_i}$ of the matrix $\mathbf{A}$.

$$\mu = \max_{i \neq j}(< \mathbf{a_i}, \mathbf{a_j} >)$$

The coherence of a matrix offers insight into the structure of a matrix. The next theorem uses it to give the conditions for uniqueness of a sparse representation.

Theorem 6. *Let* $\mathbf{A}$ *consist of the columns of two orthonormal basis with coherence* μ*. Then if a representation* $\mathbf{x}$ *where* $\|\mathbf{x}\|_0 < \frac{1}{2}(\mu^{-1} + 1)$*. Then* $\mathbf{x}$ *is the unique solution to both problems 2.1 and 2.2.*

What this theorem means is that if the BP calculated a representation $\mathbf{x}$ and if $\|\mathbf{x}\|_0 < \frac{1}{2}(\mu^{-1} + 1)$ then $\mathbf{x}$ is the sparsest possible solution. This theorem gives a way of checking our solutions.

2.3 Orthogonal Matching Pursuit

Orthogonal matching pursuit (OMP) constructs an approximation by going through an iteration process. At each iteration the locally optimum solution is calculated. This is done by finding the column vector in A which most closely resembles a residual vector $\mathbf{r}$. The residual vector starts out being equal to the vector that is required to be approximated *i.e* $\mathbf{r} = \mathbf{b}$ and is adjusted at each iteration to take into account the vector previously chosen. It is the hope that this sequence of locally optimum solutions will lead to the global optimum solution. As usual this is not the case in general although there are conditions under which the result will be the optimum solution. OMP is based on a variation of an earlier algorithm called Matching Pursuit (MP). MP simply removes the selected column vector from the residual vector at each iteration.

$$\mathbf{r_t} = \mathbf{r_{t-1}} - < \mathbf{a_{OP}}, \mathbf{r_{t-1}} > \mathbf{r_{t-1}}$$

.

Where $\mathbf{a_{OP}}$ is the column vector in $\mathbf{A}$ which most closely resembles $\mathbf{r_{r-1}}$. OMP uses a least-squares step at each iteration to update the residual vector in order to improve the approximation.

2.3.1 Algorithm

Input:

- signal $\mathbf{b}$ and matrix $\mathbf{A}$.
- stopping criterion e.g. until a level of accuracy is reached

Output:

- Approximation vector $\mathbf{c}$.

Algorithm:

1. Start by setting the residual $\mathbf{r_0} = \mathbf{b}$, the time $t = 0$ and index set $V_0 = \emptyset$
2. Let $v_t = i$, where a_i gives the solution of max $< \mathbf{r_t}, \mathbf{a_k} >$
   where $\mathbf{a_k}$ are the row vectors of $\mathbf{A}$
3. Update the set V_t with v_t: $V_t = V_{t-1} \bigcup \{v_t\}$

4. Solve the least-squares problem

$$\min_{c \in \mathbb{C}^{V_t}} \|\mathbf{b} - \sum_{j=1}^{t} c(v_j)\mathbf{a_{v_j}}\|_2$$

5. Calculate the new residual using c

$$\mathbf{r_t} = \mathbf{r_{t-1}} - \sum_{j=1}^{t} c(v_j)\mathbf{a_{v_j}}$$

6. Set $t \leftarrow t + 1$

7. Check stoping criterion if the criterion has not been satisfied then return to step 2.

2.3.2 Properties of OMP

Figure 2.3 shows the average time it takes OMP to calculate solutions using different amounts of iterations.The calculations used a randomly generated 100×300 matrix. Since the OMP algorithm has to go through n iterations to calculate a solution with $\|\mathbf{x}\|_0 = n$ therefore as can be seen from the graph it has taken OMP on average 0.5 seconds to go through 80 iterations.

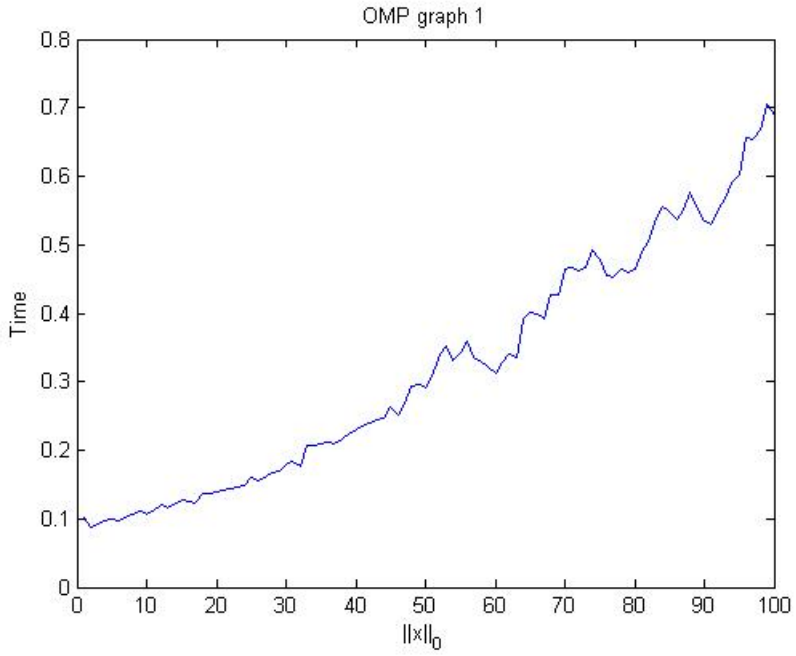

Figure 2.3: GOMP1.m output 1

Figure 2.4 shows the average error of the solutions found for figure 2.3. It can be seen that OMP can produce fairly accurate approximations at least in calculations where the matrix involved is an over-complete randomly generated one. This graph is very smooth showing that a small increase in the number of iterations leads to a small increase in accuracy.

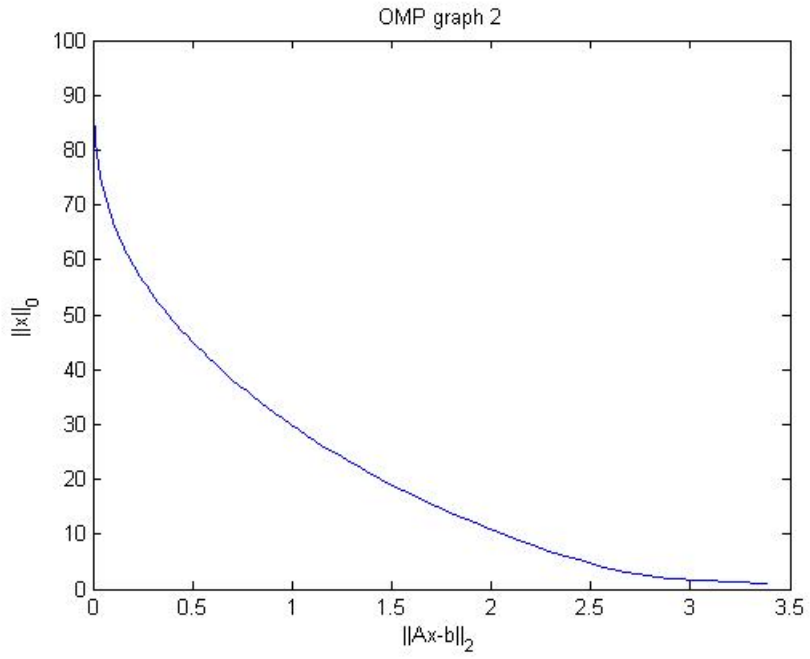

Figure 2.4: GOMP1.m output 2

Figure 2.4 shows that OMP is able to find approximations when the calculation involves over-complete random matrices but are those approximations optimum? The optimum approximation should be the most accurate for its level of sparsity (its value of L_0). There is an interesting theorem developed by Tropp [6] which gives the necessary condition for OMP to find the optimum solution. This theorem uses the matrix $\mathbf{B}$ defined below.

Definition 7. The matrix $\mathbf{B}$ consists of the columns of the matrix $\mathbf{A}$ which are used to construct the optimum representation of $\mathbf{b}$.

Theorem 8. *Orthogonal Matching Pursuit will recover the optimum representation if*

$$\max_{\mathbf{a_i}} \|\mathbf{B}^{+}\mathbf{a_i}\|_1 < 1$$

where $\mathbf{a_i}$ *is the column vectors of* $\mathbf{A}$ *that are not in* $\mathbf{B}$ *and* $\mathbf{B}^{+}$ *is the pseudo-inverse of* $\mathbf{B}$.

This condition is in fact also enough for BP to recover the optimum solution. This is very unusual because of the very different nature of the two methods.

2.4 Other Methods

2.4.1 StOMP

Stagewise Orthogonal Matching Pursuit (StOMP) [7] is an algorithm which is based on OMP.The difference between the two is that where OMP builds the solution by adding one vector at a time StOMP uses several vectors. StOMP compares the values of the dot product of $\mathbf{b}$ with the columns of $\mathbf{A}$. Then selects all vectors above a preset threshold value and uses a least squares method to find an approximation. This is then repeated with a residue vector. An advantage of using a method like this is that it can produce a good approximation with a small number of iterations. A disadvantage is in determining an appropriate value for the threshold as different threshold values could lead to different results. The algorithm is outlined below:

Algorithm
Input : Matrix $\mathbf{A}$, vector $\mathbf{b}$ and threshold n

Output : Approximation vector $\mathbf{x}$ or index set V

1. **1.** Start by setting the residual $\mathbf{r_0} = \mathbf{b}$, the iteration count $t = 0$ and index set $V = \emptyset$
2. **2.** Cerate a set J consisting of the location of all entries in the vector $\mathbf{u} = \mathbf{A^T r_t}$ above a preset value n
3. **4.** The index set by $V = V \cup J$ and residual by

$$\mathbf{x} = \min_{c \in \mathbb{R}^V} \|\mathbf{b} - \mathbf{Ac}\|_2$$

$$\mathbf{r_{t+1}} = \mathbf{b} - \mathbf{Ax}$$

4. **4.** Check stoping criterion if the criterion has not been satisfied then return to step 2.

2.4.2 ROMP

Regularized Orthogonal Matching Pursuit (ROMP) [8] is again based on OMP and like StOMP uses several vectors at each iteration to build the solution. The difference between ROMP and StOMP is that ROMP does not use a preset threshold value instead it uses the vectors which have a similar dot product with the required vector. It uses all vectors which have a dot product above half the size of the largest dot product. The advantage of this is that vectors are used which would make a similar contribution to the required vector. This process is repeated using a residue vector updated at each iteration the same as OMP. The algorithm is outlined below:

Algorithm
Input : Matrix $\mathbf{A}$, vector $\mathbf{b}$ and sparsity level n

Output : Approximation vector $\mathbf{x}$ or index set V

1. **Initialize** Start by setting the residual $\mathbf{r_0} = \mathbf{b}$, the time $t = 0$ and index set $V = \emptyset$

2. **Identify** Choose a set J of the n biggest coordinates in the vector $\mathbf{u} = \mathbf{A^T r_t}$, (or all of its nonzero coordinates if this set is smaller)

3. **Regularize** Find among all subsets $J_0 \subset J$ the maximum $\|\mathbf{u}|_{\mathbf{J_0}}\|_2$.
   Where J_0 is define $i, j \in J_0$ if $\|u_i\| \leq 2\|u_j\|$and $u_i \in \mathbf{u}|_{\mathbf{j_0}}$ if $k \in J_0$

4. **Update** The index set by $V = V \cup J_0$ and residual by

$$\mathbf{x} = \min_{c \in \mathbb{R}^V} \|\mathbf{b} - \mathbf{Ac}\|_2$$

$$\mathbf{r_{t+1}} = \mathbf{b} - \mathbf{Ax}$$

   Check stoping criterion if the criterion has not been satisfied then return to step 2.

The identification and regularization steps of ROMP can be computed efficiently and therefore the ROMP has similar running time to OMP. Since this method dose not require a threshold value it has an advantage over StOMP.

2.4.3 CoSaMP

Compressive Sampling Matching Pursuit (CoSaMP) [9] like ROMP and StOMP is based on OMP and also takes in a number of vectors to build the approximation at each iteration. CoSaMP selects a preset number of vectors from $\mathbf{A}$. The vectors which produce the largest dot product with $\mathbf{b}$. CoSaMP then restricts the approximation to the required level of sparsity by removing all but the required amount of entries. CoSaMP is one of the latest algorithms developed based on OMP. Tropp and Needell [9] claim that CoSaMP has the following very impressive properties.

1. Accepts samples from a variety of sampling schemes.
2. Succeeds using a minimal number of samples.
3. Be robust when samples are contaminated with noise.
4. Provides optimal error guarantees for every target signal.
5. Offers provably efficient resource usage.

The following definitions are need for the CoSaMP algorithm.

Definition 9. Let V be a set positive integers and $\mathbf{x}$ a vector. Then the notation $\mathbf{x}|_{\mathbf{V}}$ is $\mathbf{x}$ restricted to V

$$\mathbf{x}|_{\mathbf{V}} = \begin{cases} x_i & i \in V \\ 0 & \text{otherwise} \end{cases}$$

Definition 10. The support of a vector $\mathbf{x}$ notation $supp(\mathbf{x})$ is

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

Algorithm

Input : Matrix $\mathbf{A}$, vector $\mathbf{b}$ and sparsity level n

Output : Approximation vector $\mathbf{x}$

1. **Initialize** Start by setting current sample $v = b$, the iteration number $t = 0$, and initial approximation $\mathbf{a_0} = \mathbf{0}$

2. Set $t = t + 1$, form a proxy vector $\mathbf{y} = \mathbf{A^T v}$ and identify the largest components $J = supp(\mathbf{y_{2n}})$

3. Set $V = J \cup supp(\mathbf{a_t})$ and create a new vector b defined as

$$\mathbf{b}|_{\mathbf{V}} = \mathbf{A}^{+}|_{\mathbf{T}}\mathbf{v}$$

$$\mathbf{b}|_{\mathbf{V^c}} = \mathbf{0}$$

4. Update the approximation $a_{t+1} = b_n$ and samples vector $v = b - Aa$ Check stoping criterion if the criterion has not been satisfied then return to step 2.

2.4.4 IHT

Iterative Hard Thresholding Algorithm (IHT) [10] is a very simple iteration algorithm but is different from the previous algorithms considered. This algorithm is not based on OMP. It uses a non-linear operator to reduce the value of the L_0 norm at each iteration. The operator $H_n()$ achieves this by changing all but the largest n entries to 0. The algorithm is guaranteed to work whenever $\|\mathbf{A}\|_2 < 1$. IHT gives similar performance guarantees to those of CoSaMP. T. Blumensath, M. E. Davies [11] showed that this algorithm has the following properties

1. It gives near-optimal error guarantees.
2. It is robust to observation noise.
3. It succeeds with a minimum number of observations.
4. It can be used with any sampling operator for which the operator and its adjoint can be computed.
5. The memory requirement is linear in the problem size

The algorithm is as follows.
Algorithm
Input : Matrix $\mathbf{A}$, vector $\mathbf{b}$ and sparsity level n

Output : Approximation vector $\mathbf{x}$

$$\mathbf{x_0} = \mathbf{0}$$

$$\mathbf{x_{t+1}} = H_n(\mathbf{x_t} + \mathbf{A^T}(\mathbf{b} - \mathbf{A}\mathbf{x_t}))$$

2.5 Discussion

In this section the properties of OMP and Lasso are compared. The table below gives the average results of a 100 runs of the OMP and the Lasso. The inputs were a randomly generated matrix and vector with dimensions 10×30 matrix and 10×1. This table was generated by the Matlab code *timetable*.

Lasso				OMP		
time	λ	L_0	error	time	iterations	error
0.5366	0.1	2	1.5981	0.0895	1	0.8582
0.5587	0.2	2	1.4381	0.0954	2	0.6113
0.5853	0.3	3	1.2904	0.1003	3	0.4827
0.5978	0.4	3	1.0527	0.1014	4	0.3258
0.5971	0.5	4	0.9727	0.0957	5	0.2360
0.5994	0.6	5	0.8454	0.0970	6	0.1571
0.6065	0.7	5	0.7330	0.0959	7	0.0961
0.6107	0.8	6	0.6191	0.0967	8	0.0412
0.6148	0.9	7	0.5413	0.1025	9	0.0125
0.6236	1.0	7	0.4886	0.0971	10	0

As can be seen from the table OMP is in general a faster running algorithm than the Lasso and this is because of its simplicity. The main computational cost is in solving the least squares problem in step .4 of the algorithm. The Lasso takes more time to compute as it is a more complicated optimization problem.

Figure 2.3 generated by OMP is very different than the version generated by the Lasso (figure 2.2). This is because the Lasso builds an approximation by taking groups of vectors at a time from $\mathbf{A}$ unlike OMP. This gives more information about the structure of the vector that is being approximated. The smooth graph produced by OMP makes it difficult to decide on the sparsity level of an approximation. It is for this reason that the Lasso is generally the preferred algorithm.

It is the aim of the algorithms in the **section 2.4** to build an approximation using groups of vectors at a time like the lasso while maintaining a relatively fast run time like OMP.

Chapter 3

Applications

3.1 Feature Extraction

Feature extraction is the removal or separating out of information. This can be very useful when analyzing data if it is only one particular feature of that data that is of interest. Matrices with columns which have similar properties to the desired features can be used to approximate those features. It is also necessary that the matrix chosen can only be used to badly approximate or not approximate at all the undesired features.

For example take a data vector where the entries gradually change in value in some parts and dramatically change in others. It would be difficult to construct a sparse approximation for this vector with respect to a matrix with column vectors that only change gradually. When trying to build a sparse approximation with this matrix the smooth features would be first approximated to give an extraction. The images below taken from a paper by Elad, Starck, Querre and Donoho [12], shows an original image and its decomposition.

Figure 3.1: Original image

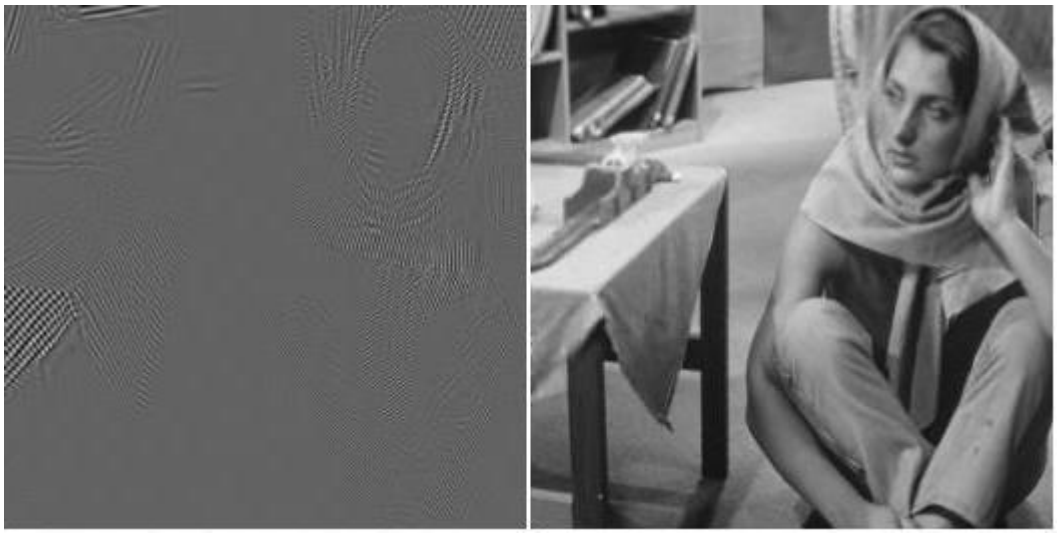

Figure 3.2: Texture layer(left) and cartoon layer(right)

Wavelets and curvelets are commonly used to produce the cartoon layers of images. The cartoon layer consists of all the smooth areas in an image. The wavelets are constructed by taking scaled and translated copies of a predefined "mother wavelet". Curvelets are extensions of the wavelets and are constructed in a similar way. To produce the cartoon layer in figure 3.2 (right) a curvelet matrix was used. The texture layer consists of all lines and edges in the image. The matrices which can be used to extract texture are the discrete cosine transform (DCT) and the Gabor wavelet (a certain type of wavelet). In the texture layer in the above example a DCT matrix was used to decompose the image.

3.2 Denoising

In the modern world it is very common to work with data that is not perfect. The imperfections in the data are referred to as noise. This could be because the data has become distorted during transfer or has become corrupt whilst being stored. In many areas of science it is difficult to obtain accurate measurements and this can result in noisy data being produced. Sparse approximations can be used to remove noise from data. This is related to feature extraction thinking of the noise as the undesired feature.

In the example below an image had randomly generated noise add to it and then was approximated using a wavelet dictionary

Figure 3.3: Original image (left) and noisy image (rigth)

Figure 3.4: Denoised image (left) and residual image (right)

Because the noise is random it is not approximated well by smooth curves and all most not approximated at all when restricting the level of sparsity. In the above example the noisy image is approximated using wavelets which better represent the smoother features. However the wavelets matrix is also poor at reproducing the texture of the image. This is why the residual image contains much of the texture as well as the noise. If column vectors were added to the wavelet matrix. This could reproduce some of the more consistent textures such as the DCT and this could improve the denoised image.

3.3 Inpainting

Inpainting is a technique used to restore information in incomplete data. This can be done by ignoring the missing information and building an approximation on the remaining information. This is usually done by adding a diagonal matrix **M** to the equation. The entries on the diagonal of **M** are of value 1 except at the locations where the information is to be ignored. Here the values of the entries are 0. The matrix **M** is known as a mask. The missing information is replaced with something which is approximately the same as the surrounding information. Therefore if there was some distinct information originally in the area to be inpainted it will not be reproduced by this technique. Below is the example of inpainting generated by *inpint*()

Figure 3.5: Original damaged recovered

Figure 3.6: Damaged image (Left) and Inpainted Image (right)

The above example of inpainting generated by *inpint*() used the sparsity toolbox [13] to produce the inpainted image. This worked really well in some areas, for example the square in the bottom right corner and not so well in others for example the top right hand square. The reason for this is because the area in the bottom right corner of the image is very similar unlike the top corner.

This technique can also be used to remove unwanted information. For example the subtitles or channel logo in video footage. Below is an example of this taken from a paper on inpainting by Fadili Starck and Murtadh [14]. The cage from the original image had been removed by treating it as missing information. The white areas in the mask show exactly what pixels have been ignored. A more advanced algorithm has been used to produce this result than for the previous image.

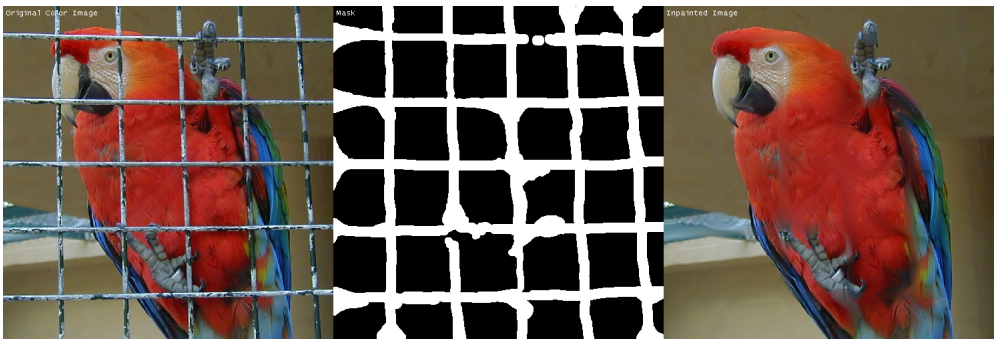

Figure 3.7: Original image (Left), Mask (center) and Inpainted Image (right)

3.4 Gene array

A gene array is a way of representing data from an experiment. A gene array is sometimes referred to as an expression array as each gene has an expression which results in a protein being produced. Below is an example of a gene micro-array where each dot represents a gene. The colour intensity of each dot corresponds to the expression levels. (the amount of protein). The measure of the expression level is only approximate.

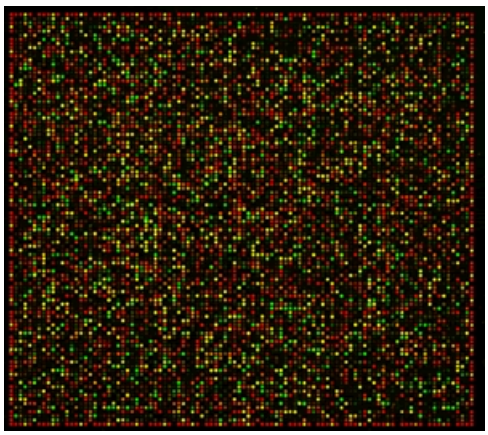

Figure 3.8: Gene array

These arrays have large numbers of genes often in the order of thousands. There is many different experiments normally in the order of hundreds of thousands. This vast amount is represented in a $M \times N$ matrix.

To analyze this data the $M \times N$ matrix is broken up by Singular Value Decomposition SVD. Let $\mathbf{G}$ be the $M \times N$ matrix

$$\mathbf{G} = \mathbf{UDV^T}$$

where $\mathbf{U}$ is referred to as the eigen-genes, $\mathbf{V}$ the loading or eigen-array and $\mathbf{D}$ is a diagonal matrix known as the eigen-expression. Now Principal Component Analysis can be used on the data. The columns of $\mathbf{U}$ are the principal components (PC) and the corresponding rows of $\mathbf{V^T}$ is the loading of each PC. The figure below shows how the PCs contribute to the columns of $\mathbf{G}$.

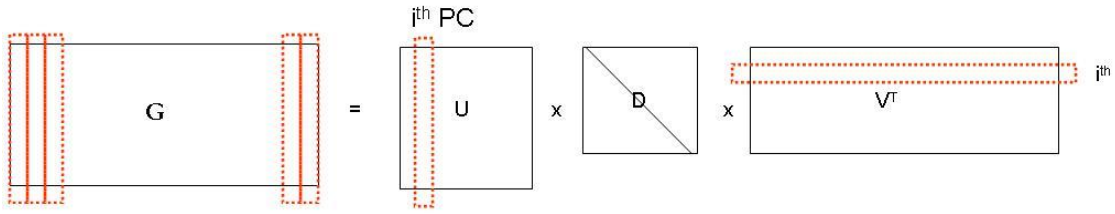

Figure 3.9: Singular Value Decomposition

If the loading is a large amount of entries this makes the result difficult to interpret. A new method was developed by H. Zou, T. Hastie, R. Tibshirani [16] called sparse principal component analysis to sparsify the loading. SPCA uses a more generalized version of the Lasso to reduce the number of entries in each row of $\mathbf{V^T}$. This makes it much clearer how the eign-genes build up the array. This in turn results in more information about the genes being determined.

Chapter 4

Matlab Code

4.1 Orthogonal Matching Pursuit Algorithm

```
function [y] = AOmp(H,A,b)
! H:number of iterations, A: matrix, b: signal
n=size(A);
A1=zeros(n);
R=b;

if(H ≤ 0)
    error('The number of iterations needs to be greater then 0')
end;

for k=1:1:H,

    [c,d] = max(abs(A'*R));
    A1(:,d)=A(:,d);
    A(:,d)=0;

    y = A1 \ b;

    R = b - A1*y;

end;
```

4.1.1 Graphs Generating Code

```
function [] = GOMP1(s,n)

t=zeros(s,2);

for kk=1:100
    A=rand(n,3*n);
    b=rand(n,1);

    for k=1:s,
        tic
        a=AOmp(k,A,b);
        t(k,1) = t(k,1) + toc;
        t(k,2) = t(k,2) + norm(A*a-b);

    end;
end;
t=t/100;

figure('Name','OMP Graph 1')
plot(1:s,t(:,1))
ylabel('Time')
xlabel('||x||_0')
title('OMP graph 1')

figure('Name','OMP Graph 2')
plot(t(:,2),1:s)
xlabel('||Ax-b||_2')
ylabel('||x||_0')
title('OMP graph 2')
```

4.2 The Lasso Algorithm

```
function [x] = Lasso(H,A,b)

cvx_begin;
    l=size(A);
    variable x(l(2));
    minimize( norm(A*x-b) );
    subject to;
```

```
    norm(x,1) ≤ H;
cvx_end;
```

4.2.1 Graphs Generating Code

```
function [t] = GLasso(s,n)

t=zeros(size(s,2),2);
A=rand(n,3*n);
b=rand(n,1);

for k=1:size(s,2),
    a=Lasso(s(k),A,b);
    t(k,2) = norm(A*a-b);
    c=0;
    for i=1:size(a,1)
        if(abs(a(i))≥0.0001)
            c=c+1;
        end;
    end;
    t(k,1)=c;
end;

figure('Name','Lasso graph 1')
plot(s,t(:,1))
ylabel('||x||_0')
xlabel('\lambda')
title('The Lasso graph 1')

figure('Name','Lasso graph 2')
plot(t(:,2),t(:,1))
xlabel('||Ax-b||_2')
ylabel('||x||_0')
title('The Lasso graph 2')
```

4.3 Timetable

```
function [t] = timetable1(n)
! Lasso t(k,1)= time, t(k,2)= lambda, t(k,3)= L-0, t(k,4)= error,
! OMP t(k,5)=  time, t(k,6)= iteration, t(k,7)= error

t=zeros(n,7);

for k=1:n,
    for kk=1:100
        A=rand(n,3*n);
        b=rand(n,1);
        tic
        x1 = Lasso(k/10,A,b);
        t(k,1)=t(k,1)+toc;
        t(k,4)=t(k,4)+norm(A*x1-b);
        for i=1:size(x1,1)
            if(abs(x1(i)) >=0.00000001)
              t(k,3)=t(k,3) + 1;
            end;
        end;
    end;
    t(k,1)=t(k,1)/100;
    t(k,2)=k/10;
    t(k,3)=round(t(k,3)/100);
    t(k,4)=t(k,4)/100;

    for ii=1:100;
        A=rand(n,3*n);
        b=rand(n,1);
        tic
        x2 = AOmp(k,A,b);
        t(k,5)=t(k,5)+toc;
        t(k,7)=t(k,7)+norm(A*x2-b);
    end;
    t(k,5)=t(k,5)/100;
    t(k,6)=k;
    t(k,7)=t(k,7)/100;
end;
```

4.4 Inpainting

```
function []=inpaint()

clear options;

%Loads and resizes the image
n = 300;
M = load_image('clo2');
M = rescale( crop(M,n) );

%Constrcution of the mask
mask=zeros(n,n);
for kk = [50,150, 250]
    for jj = [50,150, 250]
        mask(kk:kk+15,jj:jj+15)=1;
    end
end
options.mask = mask;
y = callback_inpainting(M, +1, options);

% Set up conditions for the iterative threshold method
options.thresh = 'soft';
options.Tmax = 0.1;
options.Tmin = 0;
options.niter = 400;
options.tau = 1;
options.drawiter = 0;

% Set up the wavelet matrix used ro inpaint the image
options.Jmin = 3;
options.wavelet_type = 'biorthogonal';
options.wavelet_vm = 3;
options.D = @callback_atrou;

% Retrieve the image from its coefficients
[MW,err,lun,Tlist]
= perform_iterative_thresholding(@callback_inpainting, y, options);

Mlun = callback_atrou(MW, +1, options);

%Save each image to a file and produces figure containing the images
imwrite(M,'inp1.jpg','jpg')
imwrite(y,'inp2.jpg','jpg')
```

```
imwrite(clamp(Mlun),'inp3.jpg','jpg')
imageplot({M y clamp(Mlun)},{'Original' 'Damaged' 'Recovered'});
```

4.5 Denoising

```
X=imread('image1.jpg');
X=im2double(X);
x = X + 0.2*rand(size(X));

[thr,sorh,keepapp] = ddencmp('den','wv',x);
thr

xd = wdencmp('gbl',x,'sym4',2,thr,sorh,keepapp);

figure('color','white')

subplot(221)
colormap(gray(255)), sm = 255;
image(wcodemat(X,sm)),
title('Original Image')

subplot(222)
colormap(gray(255))
image(wcodemat(x,sm)),
title('De-Noised Image')

subplot(223)
colormap(gray(255))
image(wcodemat(xd,sm)),
title('Noisy Image')

subplot(224)
colormap(gray(255))
image(wcodemat(X-xd,sm)),
title('Residual of Image')

imwrite(X,'image1a.jpg','jpg')
imwrite(x,'image2a.jpg','jpg')
imwrite(xd,'image3a.jpg','jpg')
imwrite(X-xd,'image4a.jpg','jpg')
```

Bibliography

[1] M. Grant and S. Boyd. (2008) CVX: Matlab software for disciplined convex programming (web page and software). http://stanford.edu/∼boyd/cvx.

[2] M. Grant and S. Boyd. (2008) Graph implementations for nonsmooth convex programs, Recent Advances in Learning and Control (a tribute to M. Vidyasagar), V. Blondel, S. Boyd, and H. Kimura, editors, pages 95-110, Lecture Notes in Control and Information Sciences. http://stanford.edu/∼boyd/graph_dcp.html.

[3] S. S. Chen, D. L. Donoho, and M. A. Saunders (1999) Atomic decomposition by basis pursuit. SIAM J.Sci. Comp.

[4] (2000) Computing in Science and Engineering, volume 2, no. 1,

[5] D. L. Donoho and X. Huo. (2001) Uncertainty principles and ideal atomic decomposition. *IEEE Trans. Inform.* Th., 47:2845-2862.

[6] Tropp, J. A. (2004) Greed is good: Algorithmic results for sparse aprroximation, *IEEE Trans. Inform. Theory,* vol 50, no 10, pp. 2231-2242

[7] D. Donoho, Y. Tsaig, I. Drori, J.L. Starck(2006) Sparse Solution of Underdetermined Linear Equations by Stagewise Orthogonal Matching Pursuit

[8] Needell and R. Vershynin Uniform Uncertainty Principle and signal recovery via Regularized Orthogonal Matching Pursuit. http://www-personal.umich.edu/∼romanv/papers/papers-by-topic.html

[9] D. Needell and J. A. Tropp (2008) CoSaMP: Iterative signal recovery from incomplete and inaccurate samples" http://www.acm.caltech.edu/∼jtropp/

[10] T. Blumensath, M. E. Davies (2008) Iterative Thresholding for Sparse Approximations. The Journal of Fourier Analysis and Applications, vol.14, no 5, pp. 629-654

[11] T. Blumensath, M. E. Davies (2009) Normalised Iterative Hard Thresholding; guaranteed stability and performance.

[12] M.Elad, j. L. Strack P.Querre and D.L Donoho (2005)Simultaneous cartoon and texture image inpainting usin morphological component analysis

[13] Gabriel Peyre (2006) http://www.ceremade.dauphine.fr/∼peyre/

[14] M. Fadili, J. L. Starck and F. Murtagh (2007) Inpainting and Zooming Using Sparse Representations

[15] Groningen Biomolecular Sciences and Biotechnology Institute website http://molgen.biol.rug.nl/molgen/index.php

[16] H. Zou, T. Hastie, R. Tibshirani (2004) Sparse Principal Component Analysis