SMALLbox version 1.9

An Evaluation Framework for Sparse Representations and Dictionary Learning Algorithms

Ivan Damjanovic,
Matthew E. P. Davies,
Ken O'Hanlon and
Mark D. Plumbley

Centre for Digital Music
Queen Mary University of London

Abstract.

SMALLbox is a new foundational framework for processing signals using adaptive sparse structured representations. The main aim of SMALLbox is to become a test ground for exploration of new provably good methods to obtain inherently data-driven sparse models, which are able to cope with large-scale and complicated data. The main focus of research in the area of sparse representations is in developing reliable algorithms with provable performance and bounded complexity. Yet, such approaches are simply inapplicable in many scenarios for which no suitable sparse model is known. Moreover, the success of sparse models heavily depends on the choice of a “dictionary” to reflect the natural structures of a class of data. Inferring a dictionary from training data is a key to the extension of sparse models for new exotic types of data.

SMALLbox provides an easy way to evaluate these methods against state-of-the art alternatives in a variety of standard signal processing problems. This is achieved trough a unifying interface that enables a seamless connection between the three types of modules: problems, dictionary learning algorithms and sparse solvers. In addition, it provides interoperability between existing state-of-the-art toolboxes. As an open source MATLAB toolbox, the SMALLbox can be seen as a tool for reproducible research in the sparse representations research community.

Table of Contents

Abstract. 2
1 Introduction 4
2 Sparse Representations and Dictionary Learning 5
3 Design approach to SMALLbox Overview 6
3.1 Generating Problems (Problem structure) 7
3.2 Dictionary Learning (DL structure) 7
3.3 Sparse Representation (solver structure) 8
4 SMALLbox Implementation 9
4.1 SMALLbox Installation 9
4.2 SMALLbox directory structure 9
4.3 Toolboxes 10
4.4 Implementation of Sparse Representation and Dictionary Learning Problems in SMALLbox 11
4.4.1 Implementing a sparse representation problem example 12
4.5 Implementation of sparse solvers in SMALLbox 18
4.5.1 Testing a new solver on Sparco problems example 18
4.6 Implementation of Dictionary Learning algorithms in SMALLbox 22
4.6.1 Testing Dictionary Learning Algorithms on an Image denoising Problem 23
5 Acknowledgments 26
5.1 SMALLbox 26
5.2 Included Toolboxes 26
6 References 26

1 Introduction

The field of Sparse representations has become a very active research area in recent years, and many toolboxes implementing a variety of greedy or other types of sparse algorithms have become freely available in the community [1-4]. As the number of algorithms has grown, there has become a need for a proper testing and benchmarking environment. This need was partially addressed with the SPARCO framework [5], which provides a large collection of imaging, signal processing, compressed sensing, and geophysics sparse reconstruction problems. It also includes a large library of operators that can be used to create new test problems. However, using SPARCO with other sparse representations toolboxes, such as SparseLab [1] is tedious and non-trivial because of the inconsistency in the APIs of the different toolboxes.

Many algorithms exist that aim to solve the sparse representation dictionary learning problem [6-7, 10]. However, no comprehensive means of testing and benchmarking these algorithms exist, in contrast to the problem of sparse representation with a known dictionary. The main driving force for this work is the lack of a toolbox such as SPARCO for dictionary learning problems. Recognising the need of the community for such a toolbox, we set out to design the SMALLbox - a MATLAB toolbox with three main aims:

- to enable an easy way of comparing dictionary learning algorithms,
- to provide a unifying API that will enable interoperability and re-use of already available toolboxes for sparse representation and dictionary learning,
- to aid the reproducible research effort in sparse signal representations and dictionary learning.

2 Sparse Representations and Dictionary Learning

One of the main requirements in many signal processing applications is to represent the signal in a transformed domain where it can be expressed as a linear combination of a small number of coefficients. In many research areas such as compressed sensing, image de-noising and source separation, these sparse structured signal representations are sought-after signal models. Depending on the application, we seek either an exact solution for a noise-free model or an approximate sparse reconstruction of the signal in the presence of noise:

$$\mathbf{b} = \mathbf{A}\mathbf{x} \tag{1}$$

or

$$\mathbf{b} = \mathbf{A}\mathbf{x} + \mathbf{n} \tag{2}$$

where $\mathbf{b} \in R^m$ is the signal of interest, $\mathbf{A} \in R^{m \times n}$ is a transformation matrix (or dictionary), $\mathbf{x} \in R^n$ is the sparse coefficients vector and $\mathbf{n} \in R^m$ is a noise vector. When $m<n$ the problem is *underdetermined* and there is no unique solution. Evidently, additional constraints need to be imposed on the signal model to find the solution of interest. In this sense, probably the most studied and well-understood constraint is to assume a Gaussian distribution on the coefficients and to minimise the l_2 norm of the vector **x**. However, in applications such as compression for example, it is more appropriate to impose the sparsity assumption on the coefficients, i.e. to minimise the l_0 norm of the vector **x**. Since l_0 norm minimisation is known to be an NP-hard problem, one can try finding an approximate solution using greedy algorithms such as Matching Pursuit (MP) [8] or Orthogonal Matching Pursuit (OMP) [9]. Alternatively, the sparsity assumption can be relaxed by imposing a Laplace distribution on the coefficients and minimising the l_1 norm, which can be solved using different convex optimisation methods.

In the SPARCO toolbox, the problems to be solved are given through a consistent interface represented in the form of a problem structure that contains a measurement vector **b**, an operator **A** and the other components of the test problem. Operator **A** is given in the following form:

$$\mathbf{A} = \mathbf{M}\mathbf{B} \tag{3}$$

The measurement operator **M** describes how the signal was sampled and operator **B** represents a basis with which the signal can be sparsely represented [5]. It is assumed that a basis that can give a sparse solution is known in advance. The success of the sparse representation heavily depends on the choice of the basis and the transform dictionary **A** and how well the dictionary reflects the structure present in the signal. Learning the matrix **A** from the data itself is a key to finding a sparse representation of the new classes of data. Dictionary learning for a sparse representation can be formulated as a problem of the following type:

$$\min_{\mathbf{A},\mathbf{X}} \|\mathbf{Y} - \mathbf{A}\mathbf{X}\|_F^2 \quad \text{subject to} \quad \forall i \quad \|\mathbf{x_i}\|_0^0 \le s \tag{4}$$

where **Y** is a matrix with vectors of training data and $\mathbf{x_i}$ are sparse representations of the training vectors. We want to choose a transform matrix **A** that will minimise the residual, given that the training data representation vectors $\mathbf{x_i}$ are sparse with a maximum of s non-zero coefficients.

Reflecting high activity in the research area, many dictionary learning algorithms are available, but currently no evaluation framework exists for testing them.

3 Design approach to SMALLbox Overview

The SMALLbox framework has been designed to fulfil two main goals: (1) to provide a set of test problems that permit formative evaluation of the techniques and algorithms to be developed elsewhere, and (2) to be a framework within which to build demonstrator applications. The design of the SMALLbox toolbox was constructed to allow easy portability of existing algorithms and new algorithms to be developed, taking into account the experiences in using toolboxes such as SPARCO [5] and SparseLab [1]. A graphical overview of the design of SMALLbox is shown in Fig 1.

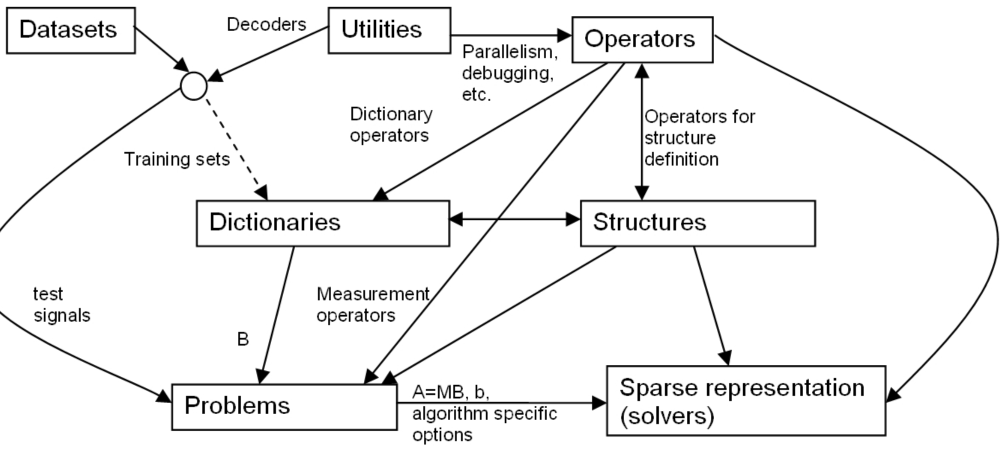


Fig. 1. Design of the Evaluation Framework.

The main interoperability of the design is given through the “Problems” part which can be defined either as a sparse representation or dictionary learning problem. In generating a problem, some of the utilities can be used to decode a dataset and prepare a test signal or a training set for dictionary learning. The dictionaries can be either defined or learned using dictionary learning algorithms. In the former case, they can be given as implicit dictionaries, as a combination of the given operators and structures, or explicitly in the form of a dictionary matix. In the latter case, they are learned from the training data. Once the dictionary is set in the problem, the problem is ready to be solved by one of the sparse representation algorithms.

SMALLbox has been designed to enable an easy exchange of information and a comparison of different modules developed through a unified API structure. The structure was made to fulfil two main goals. The first goal is to separate a typical sparse signal processing problem into three meaningful units:

a) problem specification (preparing data for learning the structures, representation and reconstruction),

b) dictionary learning (using a prepared training set to learn the natural structures in the data) and

c) sparse representation (representing the signal with a pre-specified or learned dictionary).

The second goal is to provide a seamless connection between the three types of modules and ease of communication of data between the problem, dictionary learning and sparse representation parts of the structure. To achieve these goals, SMALLbox provides a “glue” structure to allow algorithms from different toolboxes to be used with a common API.

The structure consists of three main sub-structures: ***Problem*** structure, ***DL*** (dictionary learning) structure and ***solver*** structure. Since the ***Problem*** structure is designed to be backward compatible with the SPARCO problem structure [5], it can be filled with SPARCO generateProblem or one of the dictionary learning problems provided in SMALLbox. When the specific problem is dictionary learning, one or more ***DL*** structures can be specified, so [6-7] or any other dictionary learning techniques can be compared using specified sets of parameters. Finally, to sparsely represent the signal in a dictionary (either defined in the ***Problem*** structure or learned in the previous step), one or more ***solver*** structures can be used to specify any solver from [1-4] or any of the solvers provided in SMALLbox.

3.1 Generating Problems (Problem structure)

The ***Problem*** structure defines all necessary aspects of a problem to be solved. To be compatible with SPARCO, it needs to have five fields defined prior to any sparse representation of the data:

- *A* – a matrix or operator representing a dictionary in which the signal is sparse
- *b* – a vector or matrix representing the signal or signals to be represented
- *reconstruct* – a function handle to reconstruct the signal from coefficients
- *signalSize* – the dimension of the signal
- *sizeA* – if matrix **A** is given as an operator the size of the dictionary needs to be defined in advance.

Other parameter fields can be used to further describe the problem and are useful for either reconstruction of the signal or representation of the results. These parameter fields can be generated by the SPARCO generateProblem function or the SMALLbox problem functions. The new problems implemented in the SMALLbox version 1.0 are: Image De-noising, Automatic Music Transcription and Image Representation using another image as a dictionary.

In the case of a dictionary learning problem, fields *A* and *reconstruct* are not defined while generating the problem. Instead these are defined after the dictionary is learned and prior to the sparse representation stage. In this case, field *b* needs to be given in matrix form to represent the training data and another field *p* defining the number of dictionary elements to be learned needs to be specified.

3.2 Dictionary Learning (DL structure)

The structure for dictionary learning - ***DL*** is a structure that defines the dictionary learning algorithm to be used. It is initialised with a utility function SMALL_init_DL, which will define five mandatory fields:

- *toolbox* - a field used to discriminate the API
- *name* - the name of dictionary learning function from the particular toolbox
- *param* - a field containing parameters for the particular DL technique and in the form given by the toolbox API
- *D* - a field where the learned dictionary will be stored
- *time* - a field to store the time elapsed during the learning stage.

After the *toolbox*, *name* and *param* fields are set, the function SMALL_learn is called with ***Problem*** and ***DL*** structures as inputs. According to the *DL.toolbox* field, the function calls the *DL.name* algorithm with its API and outputs learned dictionary *D* and *time* spent. The *DL.param* field contains parameters such as dictionary size, the number of iterations, the error goal or similar depending on the particular algorithm used. To compare a new dictionary learning algorithm against existing ones, the algorithm needs to be in the MATLAB path and introduced to SMALLbox by defining two parameters <*Toolbox ID*> and <*Preferred API*> in the SMALL_learn function, where examples and a simple explanation are provided. Once the new dictionary is learned, field *A* of the

Problem structure is defined to be equal to *DL.D* and also the reconstruction function is instructed to use this particular dictionary. In this way, a SPARCO compatible Problem structure is defined and ready to be used by any of the supported sparse representation algorithms for use.

3.3 Sparse Representation (solver structure)

Similar to dictionary learning, a structure containing the information required to run the sparse representation problem needs to be initialised. In this case the SMALL_init_solver function is called, which initialises and defines mandatory fields of the ***solver*** structure:

- *toolbox* - a field with toolbox name (e.g. sparselab)
- *name* - the name of solver from the particular toolbox (e.g. SolveOMP)
- *param* - the parameters in the form given by the toolbox API
- *solution* - the output representation
- *reconstructed* - the signal reconstructed from the solution
- *time* - the time taken in calculating the sparse representation.

With the input parameters of the *solver* structure set, the SMALL_solve function is called with the ***Problem*** and ***solver*** structures as inputs. The function calls the *solver.name* algorithm with the API specified by *solver.toolbox* and outputs the *solution, reconstructed* and *time* fields.

To introduce a new sparse representation algorithm, the file containing its implementation needs to be in the MATLAB path and the <*Toolbox ID*> and <*Preferred API*> parameters need to be defined for the algorithm in the SMALL_solve function, as demonstrated through the example in the section 4.5.1. As already mentioned, three solvers that can find a sparse representation of the whole training set matrix in one go are included in SMALLbox (SMALL_MP, SMALL_chol and SMALL_cgp).

4 SMALLbox Implementation

The evaluation framework is implemented as a Matlab toolbox called SMALLbox. The latest release of SMALLbox can be downloaded from http://small-project.eu and is in the form of an archive containing the SMALLbox directory structure and necessary MATLAB scripts.

The complete code base with the latest changes can be checkout at https://code.soundsoftware.ac.uk/hg/smallbox. For the development efforts we are using mecurial distributed version control system. If you are new to mecurial, we recommend you to use the easyMercurial tool provided at https://code.soundsoftware.ac.uk/projects/easyhg .

We also recommend you to check the SMALLbox project development pages were you can find all relevant information about the SMALLbox:

https://code.soundsoftware.ac.uk/projects/smallbox

- SMALLbox documentation and presentations:
  https://code.soundsoftware.ac.uk/projects/smallbox/documents
- Frequently Asked Questions:
  https://code.soundsoftware.ac.uk/projects/smallbox/wiki/FAQ
- SMALLbox releases:
  https://code.soundsoftware.ac.uk/projects/smallbox/files
- SMALLbox code repository:
  https://code.soundsoftware.ac.uk/projects/smallbox/repository
- Automatically extracted code documentation:
  https://code.soundsoftware.ac.uk/embedded/smallbox/indexlgi.html

Once opened, the installation script can be found in the root directory. To enable easy comparison with the existing state-of-the-art algorithms, installation scripts will download third party toolboxes as required.

4.1 SMALLbox Installation

The SMALLbox installation involves the automatic download of several existing toolboxes. These are described in section 4.3. Due to the automatic download of toolboxes you must have an active internet connection.

Please note that within the toolboxes are several MEX components that must be compiled. If you do not already have MEX setup, run "mex -setup" or type "help mex" in the MATLAB command prompt.

To install the toolbox run the command "SMALLboxsetup" from the MATLAB command prompt and follow the instructions.

Once installed, there are two optional demo functions that can be run. These are described in section 7. Further information can be found in the README.txt in the main SMALLbox directory.

4.2 SMALLbox directory structure

In version 1.0, SMALLbox has the following directory structure:

- SMALLbox top directory – contains SMALLboxSetup.m and README.txt
  - data – here all datasets used should be stored
    - image

        - video
        - audio
        - other
    - Problems - test problems for Dictionary Learning and Sparse representation
    - DL - all dictionary Learning algorithms developed inside of SMALLbox
    - solvers – all new solvers developed inside of SMALLbox
    - utilities – utilities such as decoders, players, GUI etc
    - toolboxes – 3rd party toolboxes (sparco, sparselab, sparsify, ksvd, sksvd, etc)
    - results - directory for storing results
    - Papers - reproducible research papers and scripts to generate results using SMALLbox
    - examples – all example and demonstration scripts (SMALL_solver_test.m, etc.)

4.3 Toolboxes

To enable easy comparison with the existing state-of-the-art algorithms, during the installation procedure SMALLbox checks the Matlab path for existence of the following freely available toolboxes and will automatically download and install them, as required:

- SPARCO (v.1.2) - set of sparse representation problems[5]
- SparseLab (v.2.1) - set of sparse solvers [1]
- Sparsify (v.0.4) - set of greedy and hard thresholding algorithms [2]
- SPGL1 (v.1.7) - large-scale sparse reconstruction solver [3]
- GPSR (v.6.0) - Gradient projection for sparse reconstruction [4]
- KSVD-box (v.13) and OMP-box (v.10) - dictionary learning [6]
- KSVDS-box (v.11) and OMPS-box (v.1) - sparse dictionary learning [7].
- ALPS - Algebraic Pursuit algorithms [11]
- CVX - Matlab Software for Disciplined Convex Programming [12] [1]

[1] The list of 3rd party toolboxes included in SMALLbox version 2.0 beta

4.4 *Implementation of Sparse Representation and Dictionary Learning Problems in SMALLbox*

Beside typical sparse representation and approximation problems provided by SPARCO toolbox, in *{SMALLbox root}/Problems/* it is also possible to find various sparse representation and dictionary learning problems implemented in the scope of SMALLbox. All implemented problems have the same API and are compatible with SPARCO API. There are two functions related to every problem:

```
data = generate{ProblemName}Problem(varagin);
```

and

```
reconstructed = {ProblemName}_reconstruct(y, problemData);
```

The first function is used to read the input data and generate the problem structure according to the provided parameters. For the list of input parameters that can be specified and the structure of output data please type:

```
help generate{ProblemName}Problem
```

The second function takes as input parameters sparse coefficients returned by sparse solver and the problem data generated by generate*{ProblemName}* function and gives as an output structure with usually two fields - reconstructed signal and psnr or similar quality measure. Again, for the structure of the output type:

```
help {ProblemName}_reconstruct
```

The list of new problems implemented (*ProblemName*, description):

-	**AMT -**	dictionary learning for Automatic Music Transcription - gets an audio file creates spectrogram, so dictionary learning can be used to learn the dictionary of 88 atoms (i.e. piano notes) and activation matrix (sparse matrix of pressed piano keys).
-	**AudioDenoise -**	gets an audio file and add White Gaussian Noise with given noise level, creates the training set for dictionary learning with a given frame size and overlapping.
-	**AudioDeclipping -**	gets an audio file clip all the values above the given clipping threshold and creates the training matrix for dictionary learning with a given frame size and overlapping. The experiment is using Audio Inpainting Toolbox that is provided with SMALLbox [13].
-	**ImageDenoise -**	gets an Image file and add White Gaussian Noise with given noise level, creates the training set for dictionary learning with a given block size [6].
-	**Pierre -**	given the source image represent the target image patches with combination of the patches of the source image (see the section 4.4.1).

4.4.1 Implementing a sparse representation problem example

In this section we give an example of how to use the Problem structure in order to enable seamless communication with other parts of SMALLbox and to allow testing of different algorithms included in SMALLbox on your problem. Implementation is of course dependent on the particular sparse representation or dictionary learning problem that you want to solve. The example *Pierre_Villars_Example.m* can be found in *{SMALLbox root}/examples/Pierre Villars* directory. This example is based on the experiment suggested by Professor Pierre Vandergheynst at the SMALL meeting in Villars, hence the name.

The idea behind is to use patches from a source image as a dictionary in which we represent a target image using the matching pursuit algorithm. This simple experiment is assembled to give some idea about: How many patches from the source image are needed to form a good dictionary, and whether the number of patches (dictionary elements) depends on the source image used.

The example function first calls the *Pierre_Problem* function (*{SMALLbox root}/Problems/*) .

```
SMALL.Problem = Pierre_Problem();
```

The *Pierre_Problem* function will first prompt the user to select the source (dictionary) and target images, and will set the blocksize (image patch size) and dictsize (number of source patches used for the representation) parameters.

The code below shows that if the blocksize and dictsize parameters are specified, an array of indices of equidistant patches to be taken from the source image is calculated. Otherwise, a default setting is used, with the block size set as 5x5 and dictionary size is specified by using all 5x5 sliding patches from the source image.

```
function data=Pierre_Problem(src, trg, blocksize, dictsize);

%% set parameters %%
maxval = 255;
if ~exist( 'blocksize', 'var' ) || isempty(blocksize),blocksize = 5;end
if ~exist( 'dictsize', 'var' ) || isempty(dictsize),
  dictsize = (size(src,1)-blocksize+1)*(size(src,2)-blocksize+1);
  patch_idx=1:dictsize;
else
  num_blocks_src=(size(src,1)-blocksize+1)*(size(src,2)-blocksize+1);
  patch_idx=1:floor(num_blocks_src/dictsize):dictsize*floor(num_blocks_src/dictsize);
end
p = ndims(src);
if (p==2 && any(size(src)==1) && length(blocksize)==1)
  p = 1;
end
% blocksize %
if (numel(blocksize)==1)
  blocksize = ones(1,p)*blocksize;
end
```

The dictionary data are formed by representing the source image in matrix form with each column representing one of the normalised sliding patches from the source image, and similarly the columns of the measurement matrix contain all distinct patches of the target image.

```
S=im2col(src,blocksize,'sliding');
for j= 1:size(S,2)
  S(:,j)=S(:,j)./norm(S(:,j));
end
%% create measurement matrix %%
T=im2col(trg,blocksize, 'distinct');
```

Finally, the output structure is formed with all mandatory fields (except *reconstruct* - Section 3.1) and some additional fields, such as original images:

```
%% output structure %%
data.A = S(:,patch_idx);
data.b = T;
data.m = size(T,1);
data.n = size(T,2);
data.p = size(data.A,2);
data.blocksize=blocksize;
data.maxval=maxval;
data.sparse=1;
data.imageSrc = src;
data.imageTrg = trg;
```

The *reconstruct* field of the problem structure is not defined in the *Pierre_Problem* function, as the dictionary changes later in the *Pierre_Villars_Example* function. Instead, it is defined in the Pierre_reconstruct function (*{SMALLbox root}/util/Pierre_recostruct.m/*), which takes the sparse coefficients and *Problem* structure as inputs and outputs the reconstructed structure with reconstructed image and PSNR value:

```
function reconstructed=Pierre_reconstruct(y, Problem)
imout=Problem.A*y;
%   combine the patches into reconstructed image

im=col2im(imout,Problem.blocksize,size(Problem.imageTrg),'disctint');

%   bound the pixel values to [0,255] range
im(im<0)=0;
im(im>255)=255;

%% output structure image+psnr %%
reconstructed.image=im;
reconstructed.psnr = 20*log10(Problem.maxval * sqrt(numel(Problem.imageTrg(:))) /
norm(Problem.imageTrg(:)-im(:)));
end
```

In the *Pierre_Villars_Example* function, when the *SMALL.Problem* has been defined, the image is represented with ten different dictionary sizes using the matching pursuit algorithm to find the three most correlated patches (*{SMALLbox root}/solvers/SMALL_MP.m)*: the variables dictsize, time and PSNR are then initialised to a size to contain a value from each iteration.

```
n =10;
dictsize=zeros(1,n);
time = zeros(1,n);
psnr = zeros(1,n);
```

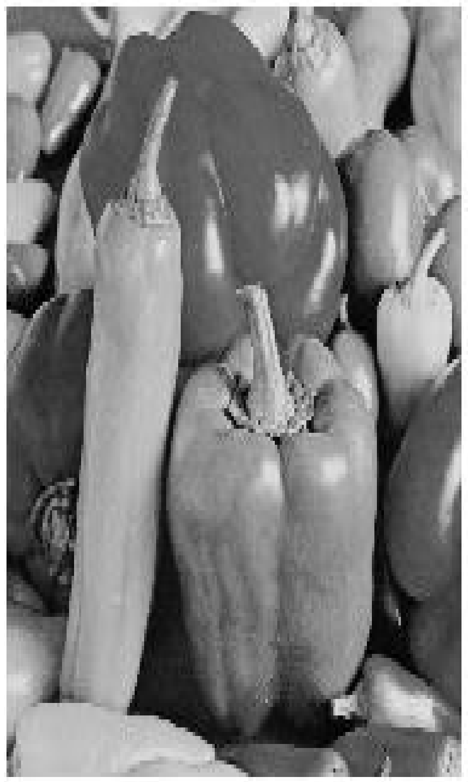
In the main loop, for each value of dictionary size the *problem.reconstruct* parameter is given a function handle to the *Pierre_reconstruct* function discussed above, then the solver parameters are set and the *SMALL_solve* function is called with the problem and solver structures. With the output from the solver structure, the relevant indexed variables above are set.

```
for i=1:n

      %   Set reconstruction function
      SMALL.Problem.reconstruct=@(x) Pierre_reconstruct(x, SMALL.Problem);

      %   Defining the parameters sparse representation
      SMALL.solver(i)=SMALL_init_solver;
      SMALL.solver(i).toolbox='SMALL';
      SMALL.solver(i).name='SMALL_MP';

      %   Parameters needed for matching pursuit (max number of atoms is 3
      %   and residual error goal is 1e-14
          SMALL.solver(i).param=sprintf('%d, 1e-14',3);

      % Represent the image using the source image patches as dictionary
      SMALL.solver(i)=SMALL_solve(SMALL.Problem, SMALL.solver(i));

      dictsize(1,i) = size(SMALL.Problem.A,2);
      time(1,i) = SMALL.solver(i).time;
      psnr(1,i) = SMALL.solver(i).reconstructed.psnr;

      %   Set new SMALL.Problem.A dictionary taking every second patch from
      %   previous dictionary
      SMALL.Problem.A=SMALL.Problem.A(:,1:2:dictsize(1,i));

      %%  show reconstructed image %%
      figure('Name', sprintf('dictsize=%d', dictsize(1,i)));
      imshow(SMALL.solver(i).reconstructed.image/SMALL.Problem.maxval);
      title(sprintf('Reconstructed image, PSNR: %.2f dB in %.2f s',...
              SMALL.solver(i).reconstructed.psnr, SMALL.solver(i).time ));
end
```

Finally, the time and PSNR values are plotted as functions of the number of source patches used

```
%%  plot time and psnr given dictionary size %%
figure('Name', 'time and psnr');
subplot(1,2,1); plot(dictsize(1,:), time(1,:), 'ro-');
  title('Time vs number of source image patches used');
subplot(1,2,2); plot(dictsize(1,:), psnr(1,:), 'b*-');
  title('PSNR vs number of source image patches used');
```

From the above example, it is possible to see how the solver structure is initialised and defined before calling *SMALL_solve* function (*{SMALLbox root}/util/*). The function will find sparse coefficients of *Problem.b* in the dictionary *Problem.A*, measure the time spent for calculating the representation and reconstruct the signal using the *Problem.reconstruct* function handle. More about the *SMALL_solve* function will be discussed in the next section.



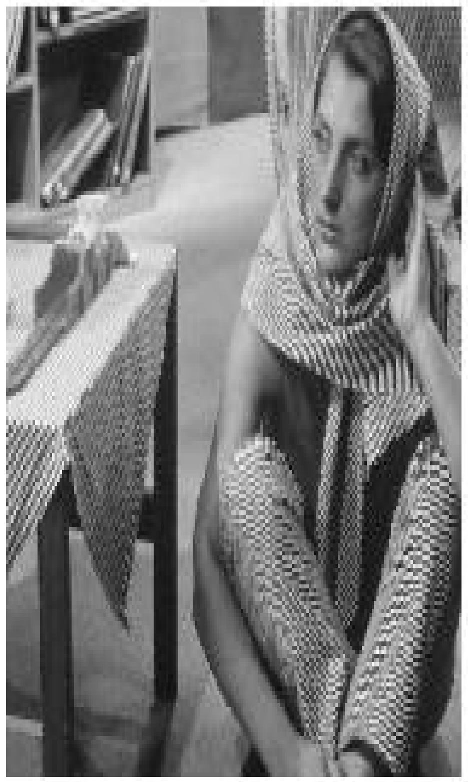


Fig 2. Two images used in the experiment *({SMALLbox root}/data/images/)*: peppers.png and barbara.png

Once all calculations are finished, the time and PSNR values for different dictionary sizes are plotted. We ran this experiment twice, first using barbara.png as a source for a dictionary to represent peppers.png. In the second instance we used peppers.png as the dictionary source to represent barbara.png image. Since the image barbara.png is much more detailed than peppers.png, it is expected that less dictionary patches will be needed in the first experiment than in the second. Figures 3 and 4 confirm this assumption. In the first experiment (Fig.3), a dictionary of 8065 equidistant patches was enough to obtain the reconstructed peppers image with PSNR of 32dB in 28.17 seconds. In the second experiment, even when we used the whole peppers.png image as a dictionary (~258000 patches), a maximum PSNR of 29.07dB in reconstructing barbara.png took more than 20 minutes to achieve.

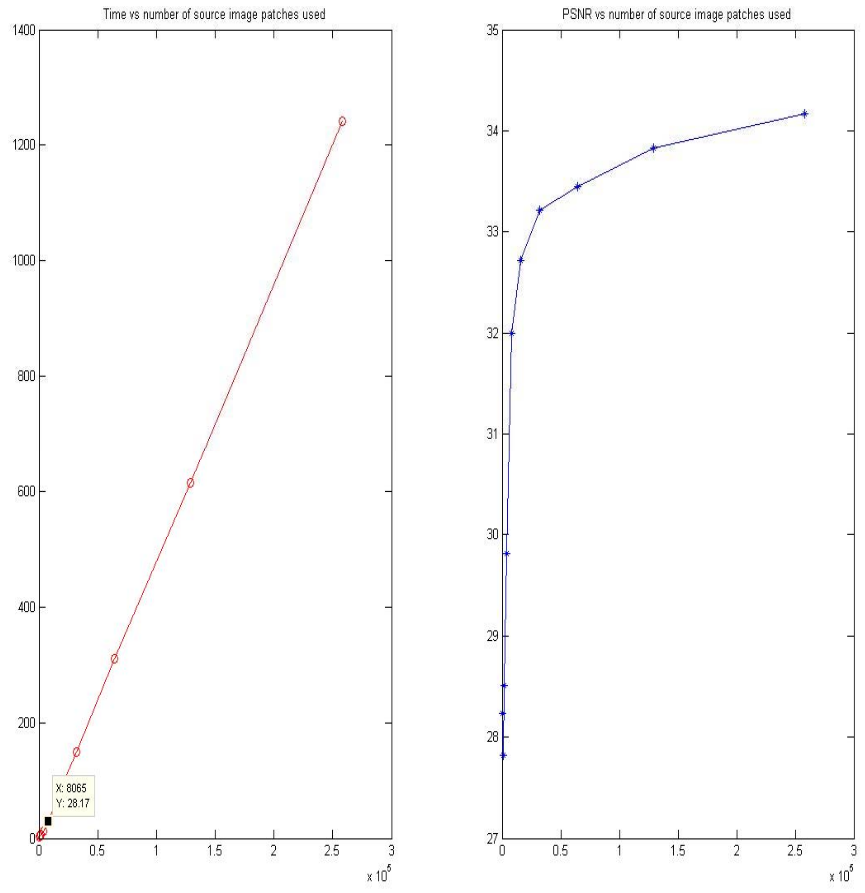


Fig 3. Representing pepper.png with variable number of patches from barbara.png image: Time and PSNR values

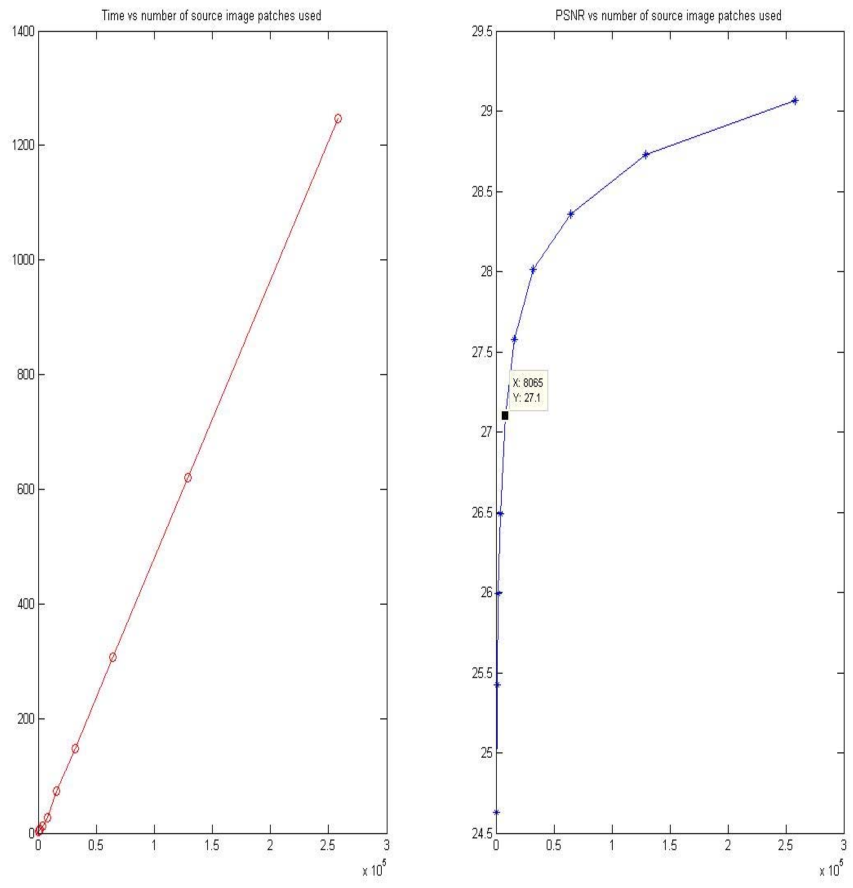


Fig 4. Representing barbara.png with variable number of patches from pepper.png image: Time and PSNR values

4.5 *Implementation of sparse solvers in SMALLbox*

As SMALLbox is made to allow comparison of different solvers, beside solvers form 3rd party toolboxes, given in section 4.3, we also provide a number of solvers developed within the SMALL project:

- SMALL_MP - Matching Pursuit
- SMALL_chol - Orthogonal Matching Pursuit with Cholesky updates
- SMALL_pcgp - Partial Conjugate Gradient Pursuit
- ompGabor, omp2Gabor - fast omp solvers for Gabor dictionary implemented as DCT+DST [13]
- ALPS toolbox - Volkan Cevher's accelerated hard thresholding methods [11]
- mm1 - Iterative Soft Thresholding implemented as a part of Majorization toolbox [14]

For more information about implemented solvers, their usage and their parameters, please type:

```
help {solver Name}
```

You can also consult examples given in *{SMALLbox root}*/examples/ directory to see how solvers are used within the SMALbox.

4.5.1 Testing a new solver on Sparco problems example

In this section, we show how to introduce a new solver to the SMALLbox results from different solvers.

First, a new solver is introduced. A new solver with its own API needs to be defined in *SMALL_solve.m* (*{SMALLbox root}/util/*). This function takes the *problem* and *solver* structures as inputs, and outputs the updated *solver* structure as the solution.

```
function solver = SMALL_solve(Problem, solver)
```

First, in this function, the *problem* structure is parsed and it is checked whether the dictionary matrix is given in implicit or explicit form:

```
if isa(Problem.A,'float')
    A = Problem.A;
    SparseLab_A=Problem.A;
    m = size(Problem.A,1);      % m is the no. of rows.
    n = size(Problem.A,2);      % n is the no. of columns.
else
    A  = @(x) Problem.A(x,1);  % The operator
    AT = @(y) Problem.A(y,2);  % and its transpose.
    SparseLab_A =@(mode, m, n, x, I, dim) SL_A(Problem.A, mode, m, n, x, I, dim);
    m = Problem.sizeA(1);      % m is the no. of rows.
    n = Problem.sizeA(2);      % n is the no. of columns.

end
b  = Problem.b;            % The right-hand-side vector.
```

Here one can notice the slightly different function handles used with different implicit dictionaries, due to the different APIs of Sparselab and Sparco. The additional function *{SMALLbox root}/util/SL_A.m* is provided as a bridge between the different implicit dictionary matrices used in SPARCO and Sparselab. Once the signal and dictionary are prepared, the solver given by the *name* part of *solver structure* is called with the appropriate API as defined by the *toolbox* field of the solver structure. Here is an example using the Sparselab and sparsify toolboxes:

```
if strcmpi(solver.toolbox,'sparselab')
  y =eval([solver.name,'(SparseLab_A, b, n,',solver.param,');']);
elseif strcmpi(solver.toolbox,'sparsify')
  y =eval([solver.name,'(b,A,n,''P_trans'',AT,',solver.param,');']);
```

The *param* field of the *solver* structure can be either a string or a structure depending on the API used for the particular toolbox.

Then the *SMALL_solve* function updates the *solver* structure with the sparse coefficients (*solver.solution)*, the reconstructed signal (*solver.reconstructed*) and the time spent constructing the sparse representation (*solver.time*):

```
solver.time = cputime - start;
fprintf('Solver %s finished task in %2f seconds. \n', solver.name, solver.time);
if isfield(Problem, 'sparse')&&(Problem.sparse==1)
    solver.solution = y;
else
    solver.solution = full(y);
end
solver.reconstructed  = Problem.reconstruct(solver.solution);
```

To introduce a new sparse representation algorithm to the SMALLbox, the file containing the code for the algorithm needs to be put into the MATLAB path. For example, one has a function called *My_dummy_OMP*, to be used in the SMALLbox, with the following API call:

```
y=My_dummy_OMP(size_y, dictionary, signal, error_goal, iter_num);
```

A name needs to be defined for your toolbox in order to differentiate your API from other toolboxes. Using the example name *My_toolbox* the following line needs to be inserted to the *if* statement in the *SMALL_solve.m* script:

```
elseif strcmpi(solver.toolbox,'My_toolbox')
  y =eval([solver.name,'(n,A,n,',solver.param,');']);
```

To test the function, the *SMALL_solver_test.m* script from the *{SMALLbox_root}*/*examples* directory can be modified as follows:

```
SMALL.Problem = generateProblem(6, 'P', 6, 'm', 270,'n',1024, 'show');
```

```
i=1;
%%
% My_OMP test test
SMALL.solver(i)=SMALL_init_solver;
SMALL.solver(i).toolbox='My_toolbox';
SMALL.solver(i).name='My_dummy_OMP';

% In the following string all parameters except matrix, measurement vector
% and size of solution need to be specified. If you are not sure which
% parameters are needed for particular solver type "help <Solver name>" in
% MATLAB command line

SMALL.solver(i).param='1e-14, 200';
SMALL.solver(i)=SMALL_solve(SMALL.Problem, SMALL.solver(i));
```

For comparison, another call is made to Sparselab's *SolveOMP* function.

```
i=i+1;
% SolveOMP from SparseLab test

SMALL.solver(i)=SMALL_init_solver;
SMALL.solver(i).toolbox='SparseLab';
SMALL.solver(i).name='SolveOMP';

SMALL.solver(i).param='200, 0, 0, 0, 1e-14';
SMALL.solver(i)=SMALL_solve(SMALL.Problem, SMALL.solver(i));
SMALL_plot(SMALL);
end % function SMALL_solver_test
```

Finally, the script can be run, and the plots of the coefficients and the reconstructed signal output for two solvers as in Figure 5. Here, *SMALL_chol* function from *{SMALLbox_root}/solvers* directory was used instead of *My_dummy_OMP*.

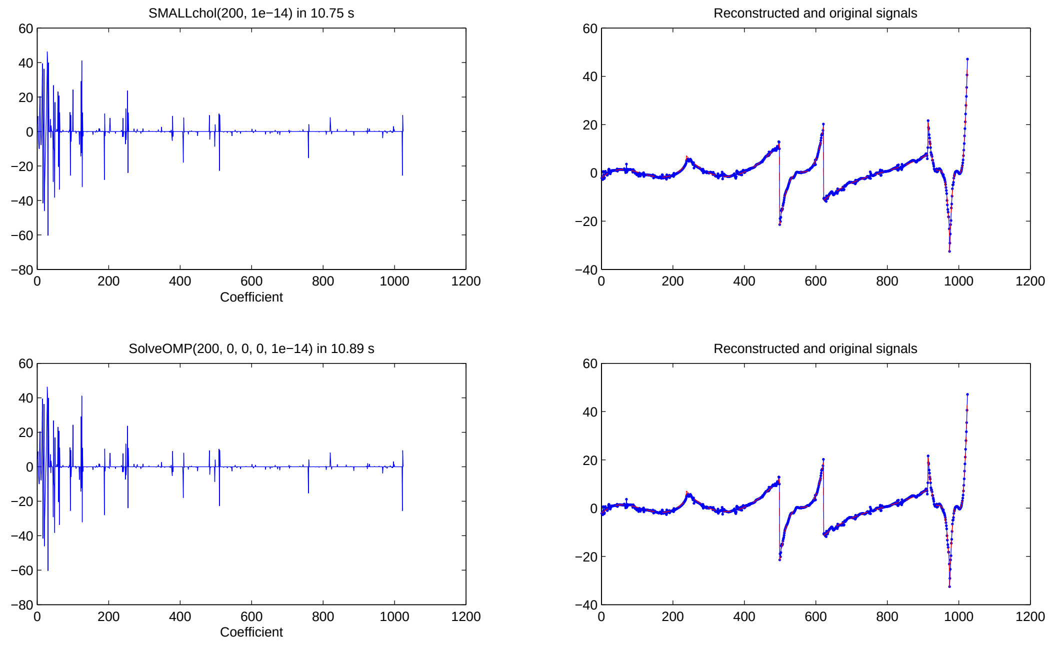


Fig 5. SMALL_solver_test.m - Comparing *SMALL_chol* and *SolveOMP* from Sparselab on SPARCO problem 6

4.6 Implementation of Dictionary Learning algorithms in SMALLbox

The main driving force for SMALLbox is making an framework for Dictionary Learning (DL) evaluation. Hence, we incorporated a number of standards methods for sparse dictionary learning. In the SMALLbox version 1.0, it was possible to evaluate your DL algorithms against KSVD and sparse KSVD dictionary learnin gthat were provided through the 3rd party toolboxes ksvd [6] and ksvds [7].

In the version 1.9, you can find implementations of some of the major state-of-the-art DL algorithms:

- **RLS-DLA** (tollbox name: “SMALL”, algorithm name: “SMALL_rlsdla”) - Recursive Least Square Dictionary Learning Algorithm is the algorithm proposed by Engang and Skretting in [15]. It is our implementation of the algorithm that uses Orthogonal Matching Pursuit (as implemented in KSVD toolbox) for coefficients updates and dictionary update as presented in the paper. If you are using it to learn the dictionary of image patches you can visualise dictionary after desired number of iterations (using ‘show_dict’ parameter).
- **Majorization Minimization Dictionary Learning** (toolbox name: “MMbox”) - Mehrdad Yaghoobi toolbox that was used to generate the figures in [14] adapted for SMALLbox. By passing a SMALLbox solver structure as one of the parameters to Dictionary learning you can use any of the solvers provided for coefficients update step (see some of the provided examples in “*{SMALL_root}*/examples/MajorizationMinimization tests”). Following dictionary updates are provided as a part of the MMbox (mod, map and ksvd are provided for comparison purposes in MMbox and by no means represent the optimised versions of the algorithms):
    - **MM_cn** - Regularized DL with column norm constrain [14]
    - **MM_fn** - Regularized DL with Frobenius norm constrain [14]
    - **mod_cn -** Method of Optimized Direction [16]
    - **map_cn -** Maximum A Posteriori dictionar update [17]
    - **ksvd_cn** - KSVD update [7]
- **Two Step Dictionary Learning** (toolbox name: “TwoStepDL”) - Similar as with MMbox, Boris Mailhe provided dictionary update functions that he used for his comparisons of convergence of KSVD [7], MOD [16] and fixed step gradient descent algorithm (Olshausen and Field algorithm) [18]. Again, you can define any of solvers provided in SMALbox for coefficients update step and use any of dictionary updates provided by Boris function. In addition, you can also choose maximal mutual coherence (‘coherence’ parameter) of dictionary atoms and dictionary will be decoralated after every iteration. Using ‘show-dict’ parameter, you can visualise dictionary after desired number of iterations.

Please consult the example scripts in the *{SMALL_root}*/examples/ directory to learn more how these algorithms are used within SMALLbox.

4.6.1 Testing Dictionary Learning Algorithms on an Image denoising Problem

The image denoising examples in SMALLbox are based on the KSVD image denoising example forwarded in [6]. They show a modular and more flexible view to the problem enabling easy comparison of dictionary learning algorithms and easy parameters testing. The image denoising problem is performed through three separate modules:

- Problem Statement (*{SMALLbox root}/Problems/GenerateImageDenoiseProblem.m*)
- Dictionary Learning (*{SMALLbox root}/util/SMALL_learn.m*)
- Image representation in the learned dictionary and denoising (*{SMALLbox root} /util/SMALL_denoise.m*)

There are three example scripts in *{SMALLbox root}/examples/Image Denoising/* directory which use the above modules to:

- Compare dictionary learning techniques (KSVD [8], sparse-KSVD [7], SPAMS[2] [10]) in terms of PSNR and the time required to perform learning and denoising.
- Compare the PSNR and the time requirements for learning using the KSVD and SPAMS algorithms, with relation to the size of the training set (number of image patches) used.
- Plot PSNR and the time taken for different values of the 'lambda' parameter in SPAMS dictionary learning.

In the first example, we compared the KSVD algorithm [6] with S-KSVD [7]. The main idea presented in [7] is that if an implicit dictionary (in this case an overcomplete DCT) is used as the base dictionary over which the sparse dictionary is learned, much better computational time can be achieved while still keeping adaptability and the performance characteristics of explicit dictionaries:

The function *generateImageDenoiseProblem* is used to fill the fields of the SMALL.Problem structure. It will prompt the user for an image, then add the noise and generate a training set of 40000 image patches with the default 8x8 blocksize before initialising a dictionary of size 256 with an overcomplete DCT.

```
SMALL.Problem = generateImageDenoiseProblem('', 40000);
```

A DL (dictionary learning) structure is initialised and the parameters required for the KSVD inserted before calling *SMALL_learn* to start the dictionary learning.

```
%%
%   Use KSVD Dictionary Learning Algorithm to Learn overcomplete dictionary


SMALL.DL(1)=SMALL_init_DL();
SMALL.DL(1).toolbox = 'KSVD';
SMALL.DL(1).name = 'ksvd';
%   KSVD PARAMETERS (Type help ksvd in MATLAB prompt for more about parameters).
Edata=sqrt(prod(SMALL.Problem.blocksize)) * SMALL.Problem.sigma * SMALL.Problem.gain;
SMALL.DL(1).param=struct('Edata', Edata, 'initdict', SMALL.Problem.initdict,...
    'dictsize', SMALL.Problem.p, 'iternum', 20,'memusage', 'high');


%   Learn the dictionary
```

[2] An API for SPAMS [10] is included in SMALLbox together with examples using SPAMS, but due to licensing issues this toolbox needs to be installed by the user.

```
SMALL.DL(1) = SMALL_learn(SMALL.Problem, SMALL.DL(1));
```

The dictionary in the DL structure, which has been learned following the call to *SMALL_learn* is inserted into the *SMALL.Problem* structure.

```
SMALL.Problem.A = SMALL.DL(1).D;
```

A solver struct is then initialised, and parameters inserted before *SMALL_denoise* is called, with the problem and solver structs as fields, to perform the denoising.

```
%%
%   Initialising solver structure for denoising
%   Setting solver structure fields (toolbox, name, param, solution,
%   reconstructed and time) to zero values
SMALL.solver(1)=SMALL_init_solver;

% Defining the parameters needed for image denoising
SMALL.solver(1).toolbox='ompbox';
SMALL.solver(1).name='ompdenoise';

%   Denoising the image - SMALL_denoise function is similar to SMALL_solve,
%   but backward compatible with KSVD, doing sparse representation and denoising in one
%   go. Should be changed in the next version

SMALL.solver(1)=SMALL_denoise(SMALL.Problem, SMALL.solver(1));
```

The following code creates another DL structure, this time for the KSVDS algorithm, and inserts the relevant parameters.

```
% Use KSVDS Dictionary Learning Algorithm to denoise image

SMALL.DL(2)=SMALL_init_DL();
SMALL.DL(2).toolbox = 'KSVDS';
SMALL.DL(2).name = 'ksvds';

%   Defining the parameters for KSVDS
%   In this example we are learning 256 atoms in 20 iterations, so that
%   every patch in the training set can be represented with target error in
%   L2-norm (EDataS). We also impose "double sparsity" - dictionary itself
%   has to be sparse in the given base dictionary (Tdict - number of
%   nonzero elements per atom).
%   Type help ksvds in MATLAB prompt for more options.

EdataS=sqrt(prod(SMALL.Problem.blocksize)) * SMALL.Problem.sigma * SMALL.Problem.gain;
SMALL.DL(2).param=struct('Edata', EdataS, 'Tdict', 6, 'stepsize', 1,...
    'dictsize', SMALL.Problem.p, 'iternum', 20, 'memusage', 'high');

SMALL.DL(2).param.initA = speye(SMALL.Problem.p);
SMALL.DL(2).param.basedict{1} = odctdict(8,16);
SMALL.DL(2).param.basedict{2} = odctdict(8,16);
```

When the parameters are set, a call is made to learn the dictionary. The dictionary learned is returned in the *DL* structure. This is then assigned as the dictionary in the *SMALL.Problem* structure, along with other parameters related to the *DL* structure.

```
% Learn the dictionary
SMALL.DL(2) = SMALL_learn(SMALL.Problem, SMALL.DL(2));

SMALL.Problem.A = SMALL.DL(2).D;
SMALL.Problem.basedict{1} = SMALL.DL(2).param.basedict{1};
SMALL.Problem.basedict{2} = SMALL.DL(2).param.basedict{2};
```

A solver structure is then initialised and the parameters for this structure are added before the *small_denoise* function, with the problem and solver structs, is called to perform the denoising

```
%   Initialising solver structure
SMALL.solver(2)=SMALL_init_solver;
SMALL.solver(2).toolbox='ompsbox';
SMALL.solver(2).name='ompsdenoise';

SMALL.solver(2)=SMALL_denoise(SMALL.Problem, SMALL.solver(2));
```

Finally, the *SMALL_ImgDeNoiseResult* function is called to display the results from the two dictionary learning algorithms.

```
SMALL_ImgDeNoiseResult(SMALL);
```

The *SMALL_ImgDeNoiseResult* function will show the original, noisy, and denoised images, the learned dictionaries, the amounts of time required for learning for learning and denoising and the PSNR values (Figure 6). The results of this experiment support the claim given in [7]. De-noising in the S-KSVD is almost 3 times faster while the PSNR is only 0.09 dB lower.

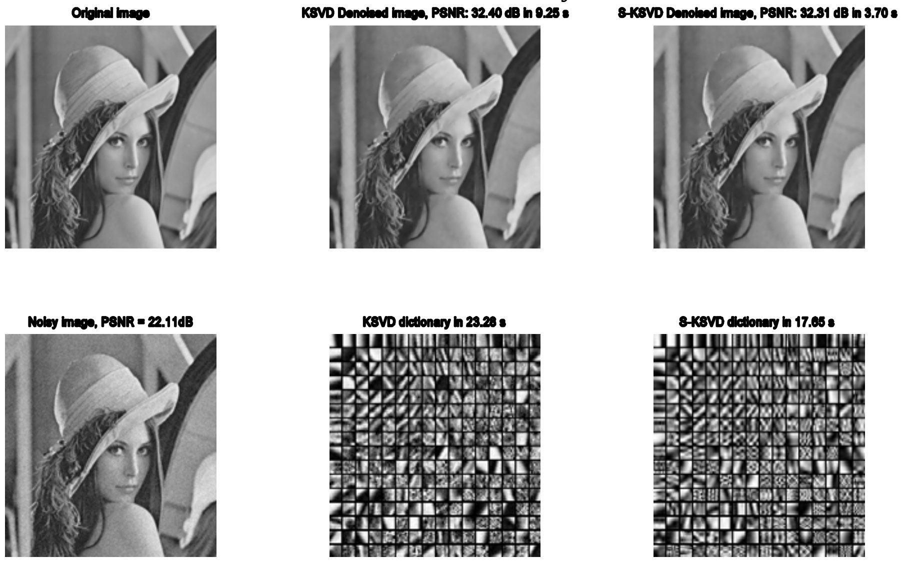

Original image
KSVD Denoised image, PSNR: 32.40 dB in 9.25 s
S-KSVD Denoised image, PSNR: 32.31 dB in 3.70 s
Noisy image, PSNR = 22.11dB
KSVD dictionary in 23.28 s
S-KSVD dictionary in 17.65 s

Fig. 6. SMALLbox example results - KSVD [6] versus S-KSVD [7] in image de-noising

5 Acknowledgments

5.1 SMALLbox

- This research has been funded by the EU FET-OPEN Project, no: 225913: Sparse Models, Algorithms, and Learning for Large-scale data (SMALL).

5.2 Included Toolboxes

- SPARCO and SPGL1 are supported in part by an NSERC Discovery Grant and by an NSERC Collaborative Research and Development Grant (334810-05) that funds the DNOISE project.

- SPARSELAB is partially supported in part by NSF DMS-05-05303.

- Part of the SPARSIFY toolbox was developed with the support of EPSRC Grant D000246/1.

- GPSR work was partially supported by the USA National Science Foundation (NSF), under grants CCF-0430504 and CNS-0540147 and the Portuguese Fundacao para a Ciencia e Tecnologia (FCT), under project POSC/EEA-CPS/61271/2004.

- KSVDbox and OMPbox are supported by United States - Israel Binational Science Foundation (BSF) grant no. 2004199 and the Israel Science Foundation grant no. 796/05.

6 References

1. Donoho, D., Stodden, V., Tsaig, Y.: Sparselab. 2007, http://sparselab.stanford.edu/
2. Blumensath, T., Davies, M. E.: Gradient pursuits. In IEEE Transactions on Signal Processing, vol. 56, no. 6, pp. 2370–2382, June 2008.
3. Berg, E. v., Friedlander, M. P.: Probing the Pareto frontier for basis pursuit solutions. In SIAM Journal on Scientific Computing, vol. 31, no. 2, pp. 890–912, 2008.
4. Figueiredo, M. A. T., Nowak, R. D., Wright, S. J.: Gradient projection for sparse reconstruction: Application to compressed sensing and other inverse problems. In Journal of Selected Topics in Signal Processing: Special Issue on Convex Optimization for Signal Processing, December 2007.
5. Berg, E. v., Friedlander, M. P., Hennenfent, G., Herrmann, F., Saab, R., Yılmaz, O.: SPARCO: A testing framework for sparse reconstruction. In ACM Trans. on Mathematical Software, 35(4):1-16, February 2009.
6. Aharon, M., Elad, M., Bruckstein, A. M.: The K-SVD: An algorithm for designing of overcomplete dictionaries for sparse representation. In IEEE Transactions on Signal Processing, vol. 54, no. 11, pp. 4311–4322, 2006.
7. Rubinstein, R., Zibulevsky, M. and Elad, M.: Double Sparsity: Learning Sparse Dictionaries for Sparse Signal Approximation. In IEEE Transactions on Signal Processing, Vol. 58, No. 3, Pages 1553-1564, March 2010.
8. Mallat, S. G., Zhang, Z.: Matching Pursuits with Time-Frequency Dictionaries. In: IEEE Transactions on Signal Processing, December 1993, pp. 3397-3415
9. Pati, Y. C., Rezaiifar, R., Krishnaprasad, P. S.: Orthogonal matching pursuit: Recursive function approximation with applications to wavelet decomposition. In Proc. 27th Asilomar Conference on Signals, Systems and Computers, A. Singh, ed., Los Alamitos, CA, USA, IEEE Computer Society Press, (1993)
10. Mairal, J., Bach, F., Ponce J., Sapiro, G.: Online Learning for Matrix Factorization and Sparse Coding. In Journal of Machine Learning Research, volume 11, pages 19-60. 2010.
11. Chever, V.: On Accelerated Hard Thresholding Methods for Sparse Approximation. Technical Report, 2011.
12. Grant, M., Boyd, S.: Matlab Software for Disciplined Convex Programming, version 1.21. http://cvxr.com/cvx, April 2011.
13. Adler, A., Emiya, V., Jafari, M. G., Elad, M., Gribnoval, R., Plumbley, M. D.: Audio Inpainting. Submitted to IEEE Transactions on Audio, Speech and Language Processing, http://hal.inria.fr/inria-00577079/en/, 2011.
14. Yaghoobi, M., Blumensath, T., Davies, M.: "Dictionary Learning for Sparse Approximation with Majorization Method", IEEE Transactions on Signal Processing, vol. 57, no. 6, pp. 2178-2191, 2009.
15. Skretting, K., Engang, K.: "Recursive Least Squares Dictionary Learning Algorithm", IEEE Transaction on Signal Processing, vol. 58, no. 4, pp. 2121-2130, 2010.
16. Engan, K. , Aase, S. O. , Husøy, J. H. : "Method of optimal directions for frame design," in *Proc. ICASSP'99*, Phoenix, AZ, pp. 2443–2446, Mar. 1999.
17. Kreutz-Delgado, K. , Murray, J. , Rao, B., Engan, K., Lee, T., Sejnowski, T.: "Dictionary learning algorithms for sparse representation", *Neural Comp.*, vol. 15, pp. 349–396, 2003.
18. Olshausen, B. , Field, D.: "Sparse coding with an overcomplete basis set: a strategy employed by V1?" *Vision Research*, vol. 37, no. 23, pp. 3311–3325, 1997.