# SMALLbox version 2.0

An Evaluation Framework for Sparse Representations and Dictionary Learning Algorithms

Daniele Barchiesi,
Ivan Damjanovic,
Matthew E. P. Davies,
Luís Figueira,
Arias Grestsistas,
Maria Jafari,
Boris Mailhé,
Ken O'Hanlon,
Mark D. Plumbley

Centre for Digital Music, Queen Mary University of London

April 26, 2012

## 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 and 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 to develop 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 through 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, SMALLbox can be seen as a tool for reproducible research in the sparse representations research community.

# Contents

# 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 [6–8]. 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 SMALLbox—a MATLAB toolbox that mainly 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 representations and dictionary learning,
- to aid the reproducible research effort in sparse signal representations and dictionary learning.

However, before explaining in detail the structure of the software, we first introduce the type of problem that we are addressing: sparse representations and dictionary learning.

# 2 Sparse Representations and Dictionary Learning

Sparse signal representations allow the salient information within a signal to be conveyed with only a few elementary components, known as atoms drawn from a given redundant matrix. For this reason, they have acquired great popularity over the years, and they have been successfully applied to a variety of problems.

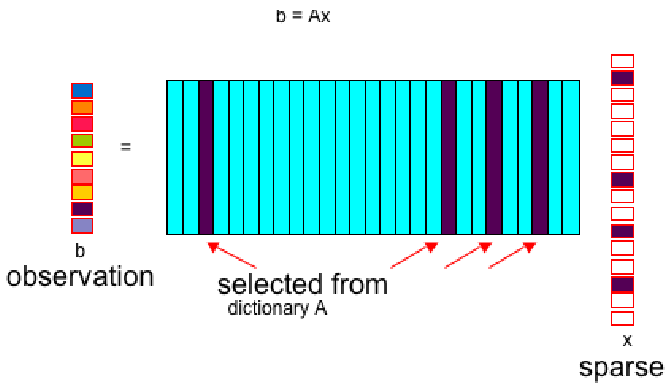

**Figure 1:** Sparse representation of a signal **b**

Figure 1 illustrates how the observed data b can be represented as a linear combination of only a few columns of the matrix A. This matrix is also known as a dictionary, and its columns are called atoms. The sparse model assumes that the vector b is generated by the superimposition of only a few atoms weighted appropriately by the corresponding non-zero entries of the sparse vector x. Therefore, one can attempt to decompose any signal of interest in a given dictionary. In the case when the resulting representation is s-sparse i.e. only s entries of x are non-zero, the signal of interest is said to be exact sparse. The sparse model is very simple, yet very powerful. For instance, in the presence of noise and interfering signals, which in contrast to the signal of interest do not have a sparse representation in the dictionary A, keeping only the largest coefficients while discarding the rest will result in the removal of a good proportion of the undesired noise or interference. However, to do this we first need to solve the above inverse problem and find a sparse representation.

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:

$$b = Ax \tag{1}$$

or

$$b = Ax + n \tag{2}$$

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 $\mathbf{x}$. 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 $\mathbf{x}$. Since $l_0$ norm minimisation is known to be combinatorial i.e. NP-hard, one can attempt to find an approximate solution using greedy algorithms such as Matching Pursuit (MP) [9] or Orthogonal Matching Pursuit (OMP) [10]. 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 $\mathbf{b}$, an operator $\mathbf{A}$ and the other components of the test problem. The operator $\mathbf{A}$ is given in the following form:

$$\mathbf{A} = \mathbf{MB} \tag{3}$$

The measurement operator $\mathbf{M}$ describes how the signal was sampled and operator $\mathbf{B}$ 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.

Successful application of a sparse decomposition depends on the dictionary used and whether it matches the signal features [11]. Two main methods have emerged to determine a dictionary within a sparse decomposition: dictionary selection and dictionary learning. Dictionary selection entails choosing a pre-existing dictionary, such as the Fourier basis, a wavelet basis or the modified discrete cosine basis, or constructing a redundant or overcomplete dictionary by forming a union of bases (for example the Fourier and wavelet bases) so that different properties of the signal can be represented [12]. Dictionary learning, on the other hand, aims at deducing the dictionary from the training data, so that the atoms directly capture the specific features of the signal or set of signals [13]. This is key to finding a sparse representation of new classes of data. Dictionary learning for a sparse representation can be formulated as a problem of the following type:

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

where $\mathbf{Y}$ 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 $textbfA$ 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.

Dictionary learning methods are often based on an alternating optimization strategy, in which the dictionary is fixed, and a sparse signal decomposition is found; then the dictionary elements are learned, while the signal representation is fixed. More recently, dictionary learning methods for exact sparse representation based on $l_1$ minimization [11, 14], and online learning algorithms [8], have been proposed. For more details on sparse decompositions and dictionary learning, see [15].

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 fulfill 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 development of new algorithms, 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. 2.

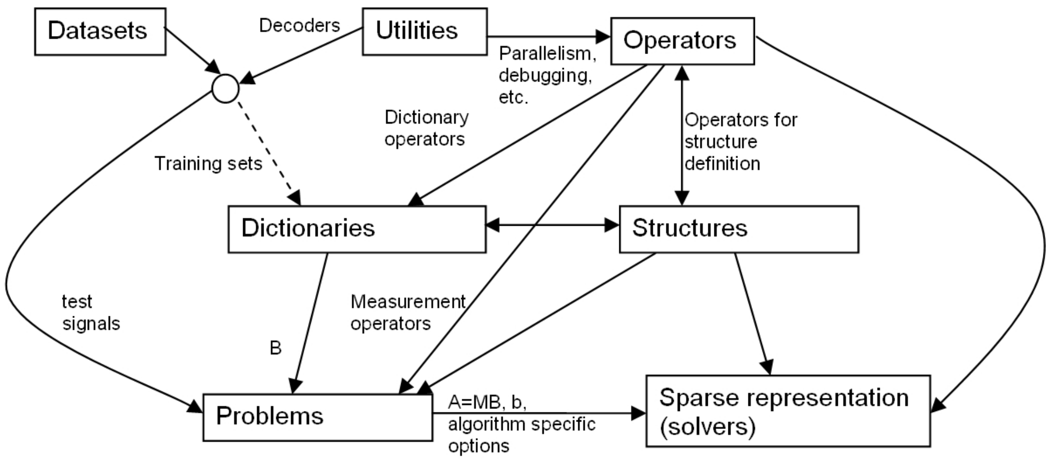

**Figure 2:** Design of the Evaluation Framework

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. When 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, using a combination of the given operators and structures, or explicitly in the form of a dictionary matrix. In the latter case, they are learned from the training data, as described in chapter 2. 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 fulfill 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, representation and reconstruction)
(b) dictionary learning (using a training set to learn the natural structures in the data)
(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's `generateProblem` function or the SMALLbox problem functions. The new problems implemented in the SMALLbox version 2.0 are: Image Denoising, Automatic Music Transcription, Audio Inpainting 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 in MATLAB. The latest release of SMALLbox can be downloaded from `http://small-project.eu` and it is supplied 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 found at `https://code.soundsoftware.ac.uk/hg/smallbox`. For the development efforts we are using Mercurial distributed version control system. If you are new to Mercurial, we recommend you to use the EasyMercurial tool provided at `http://easyhg.org/`.

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`:

**Documentation and presentations:**
`https://code.soundsoftware.ac.uk/projects/smallbox/documents`

**Frequently Asked Questions:**
`https://code.soundsoftware.ac.uk/projects/smallbox/wiki/FAQ`

**Releases:**
`https://code.soundsoftware.ac.uk/projects/smallbox/files`

**Code repository:**
`https://code.soundsoftware.ac.uk/projects/smallbox/repository`

**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 several MEX components included 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. `SMALLboxInit` provides the SMALLbox path initialisation and should be run after relaunching MATLAB or SMALLbox.

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

## 4.2 SMALLbox directory structure

In version 2.0, SMALLbox has the following directory structure:

**SMALLbox top directory** – contains `SMALLboxSetup.m`, `SMALLboxInit.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** — third party toolboxes (sparco, sparselab, sparsify, ksvd, sksvd, etc)
**examples** — all example and demonstration scripts (`SMALL_solver_test.m`, etc)
**config** — configuration files that allow users to extend SMALLbox
**doc** — SMALLbox documentation

## 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. It then automatically downloads and installs all toolboxes that are not in the MATLAB path, 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)** KSVD dictionary learning method [6]
**KSVDS-box (v.11) and OMPS-box (v.1)** sparse K-SVD dictionary learning method [7]
**ALPS** Algebraic Pursuit algorithms [16]
**CVX** MATLAB Software for Disciplined Convex Programming [17][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

Besides 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) is the following:

**AMT** – dictionary learning for Automatic Music Transcription:gets an audio file and creates its spectrogram, so that 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 adds 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 and clips all values above a given clipping threshold, 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 [18].

**ImageDenoise** – gets an Image file and adds 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 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` folder. This example is based on the experiment suggested by Professor Pierre Vandergheynst at the SMALL meeting in Villars, hence the name.

To run this example simply run the following from the Matlab console:

```
>> SMALLboxInit
>> Pierre_Villiars_Example
```

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 ((`{SMALLbox root}/Examples/Pierre Villars/Pierre_Villars_Example↩ .m`) firstly calls the `Pierre_Problem` function:

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

The `generatePierreProblem` 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=generatePierreProblem(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 is 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.

```
%% create dictionary data
S=im2col(src,blocksize,'sliding');

for j= 1:size(S,2)
    S(:,j)=S(:,j)./norm(S(:,j));
end

%% observed patches matrix
%    that are going to be decomposed on the  dictionary
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 as 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.

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

(a) `peppers.png` (b) `barbara.png`

**Figure 3:** Two images used in the experiment (`{SMALLbox root}/data/images/`): `peppers.png` and `barbara.png`

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 4 and 5 confirm this assumption. In the first experiment (Figure 4), 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.

## 4.5 Implementation of sparse solvers in SMALLbox

As SMALLbox is designed in a way that allows comparison of different solvers, apart from solvers from the third 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 [18]
**ALPS toolbox** – Volkan Cevher's accelerated hard thresholding methods [16]
**mm1** – Iterative Soft Thresholding implemented as a part of Majorization Minimization toolbox [19]

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

```
help {solver Name}
```

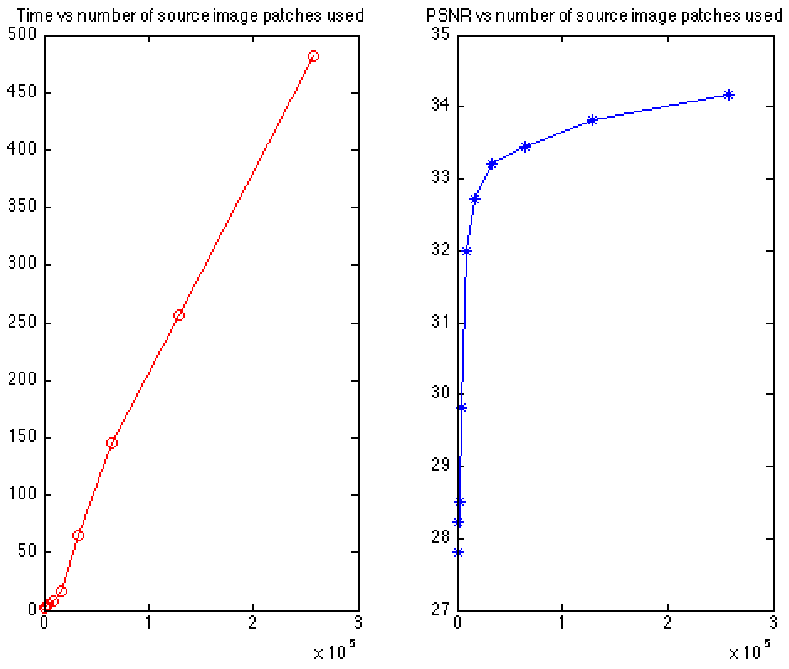

**Figure 4:** Representing `pepper.png` with variable number of patches from `barbara.png` image: Time and PSNR values.

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

### 4.5.1 Testing a new solver on Sparco problems example

In this section, we explain how to introduce a new solver to SMALLbox and test it against the provided algorithms.

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.

The SMALL_solve function then 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):

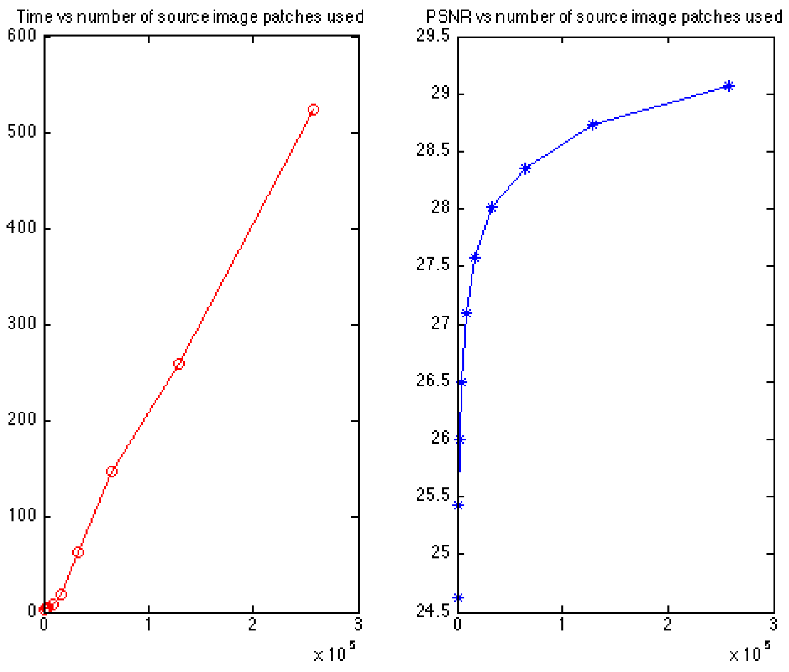

**Figure 5:** Representing barbara.png with variable number of patches from pepper.png image: Time and PSNR values.

```
    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 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 code listed in Listing 1 needs to be added to the `SMALL_solve` local configuration file (located in `{SMALLbox Root}/config/↩ SMALL_solve_config_local.m` – refer to Sec. 5.1).

**Listing 1:** Code that needs to be added to the local configuration file in order add a new solver

```
    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 6. Here, `SMALL_chol` function from `{SMALLbox_root}/`↩ `solvers` directory was used instead of `My_dummy_OMP`.

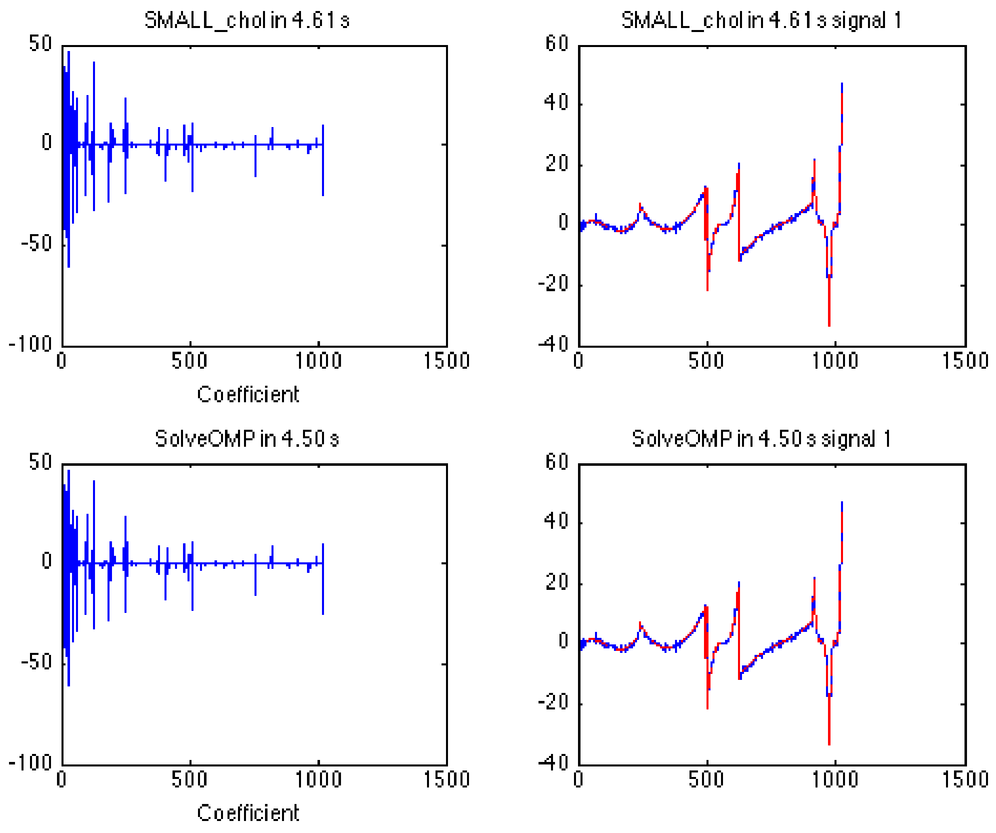

**Figure 6:** `SMALL_solver_test.m`: Comparing `SMALL_chol` and `SolveOMP` from SparseLab on SPARCO problem 6

## 4.6 Implementation of Dictionary Learning algorithms in SMALLbox

As already mentioned, the main driving force behind SMALLbox was to develop a framework for dictionary learning (DL) evaluation. Hence, we incorporated a number of standard 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 learning that were provided through the third party toolboxes `ksvd` [6] and `ksvds` [7].

In version 2.0, 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 [20]. 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 the dictionary after desired number of iterations (using the show_dict parameter).

**Majorization Minimization Dictionary Learning (toolbox name: MMbox)**
Mehrdad Yaghoobi toolbox that was used to generate the figures in [19] 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 [19]
MM_fn – Regularized DL with Frobenius norm constrain [19]
mod_cn – Method of Optimized Direction [21]
map_cn – Maximum A Posteriori dictionary update [22]
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 [21], fixed step gradient descent algorithm (Olshausen and Field algorithm) [23], optimal step gradient descent [24] and LGD (Large step Gradient Descent) [25]. Again, you can define any of solvers provided in SMALLbox for the 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 the dictionary will be decorrelated after every iteration [26]. Using the show-dict parameter, you can visualise the dictionary after a 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 [9], sparse-KSVD [7], SPAMS [8][2]) 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 compare 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 dictionary) 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. *This example can be run by calling the function* `SMALL_ImgDenoise_DL_test_KSVDvsSKSVD`.

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

A `solver` stucture is then initialised, and parameters inserted before `SMALL_denoise` is called, with the `problem` and `solver` structures as fields, to perform the denoising:

[2] An API for SPAMS [8] is included in SMALLbox together with examples using SPAMS, but due to licensing issues this toolbox needs to be installed separately by the user.

```
%%
%   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='omp2';
SMALL.solver(1).param=struct(...
    'epsilon',Edata,...
    'maxatoms', maxatoms);

%   Denoising the image - find the sparse solution in the learned
%   dictionary for all patches in the image and the end it uses
%   reconstruction function to reconstruct the patches and put them into a
%   denoised image

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

%   Initialising solver structure
%   Setting solver structure fields (toolbox, name, param, solution,
%   reconstructed and time) to zero values

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` structures, is called to perform the denoising:

```
%   Initialising solver structure
SMALL.solver(2)=SMALL_init_solver;
SMALL.solver(2).toolbox='ompsbox';
SMALL.solver(2).name='omps2';
SMALL.solver(2).param=struct(...
    'epsilon',Edata,...
    'maxatoms', maxatoms);

%   Denoising the image - find the sparse solution in the learned
%   dictionary for all patches in the image and the end it uses
%   reconstruction function to reconstruct the patches and put them into a
%   denoised image

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 and denoising and the PSNR values (Figure 7). The results of this experiment support the claim given in [7]. De-noising in the S-KSVD is faster while the PSNR is only 0.08 dB lower.

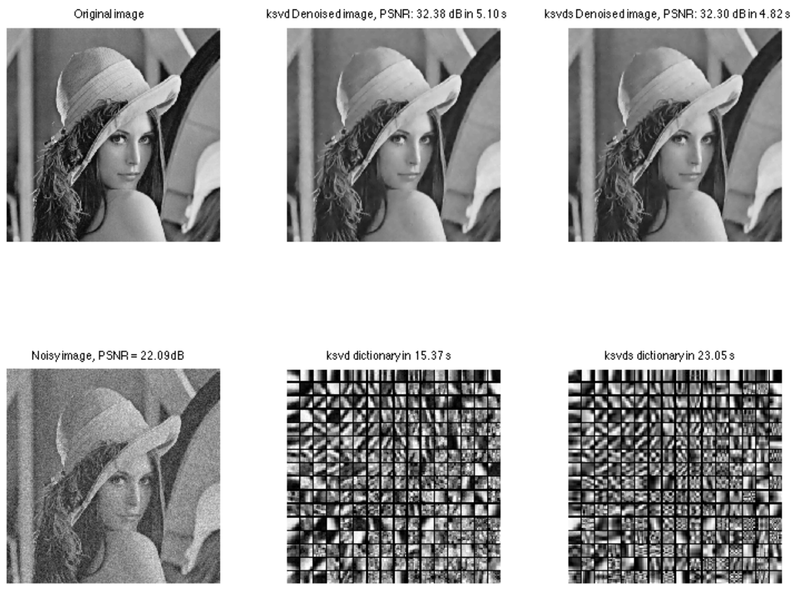
Original image
ksvd Denoised image, PSNR: 32.38 dB in 5.10 s
ksvds Denoised image, PSNR: 32.30 dB in 4.82 s
Noisy image, PSNR = 22.09dB
ksvd dictionary in 15.37 s
ksvds dictionary in 23.05 s

**Figure 7:** SMALLbox example results - KSVD [6] versus S-KSVD [7] in image de-noising

### 4.6.2 A Dictionary Learning "toy example"

The file `{SMALLbox root}/examples/SMALL_DL_test.m` contains a very simple use-case of the KSVD dictionary learning algorithm [6], which illustrates the advantages of learning an over-complete set of atoms for sparse approximation over the classic principal component analysis (PCA) transform.

First, a training set consisting of two-dimensional signals is generated, such that most of the points follow three distinct directions when plotted on an x/y plane. This is illustrated in Figure 8.

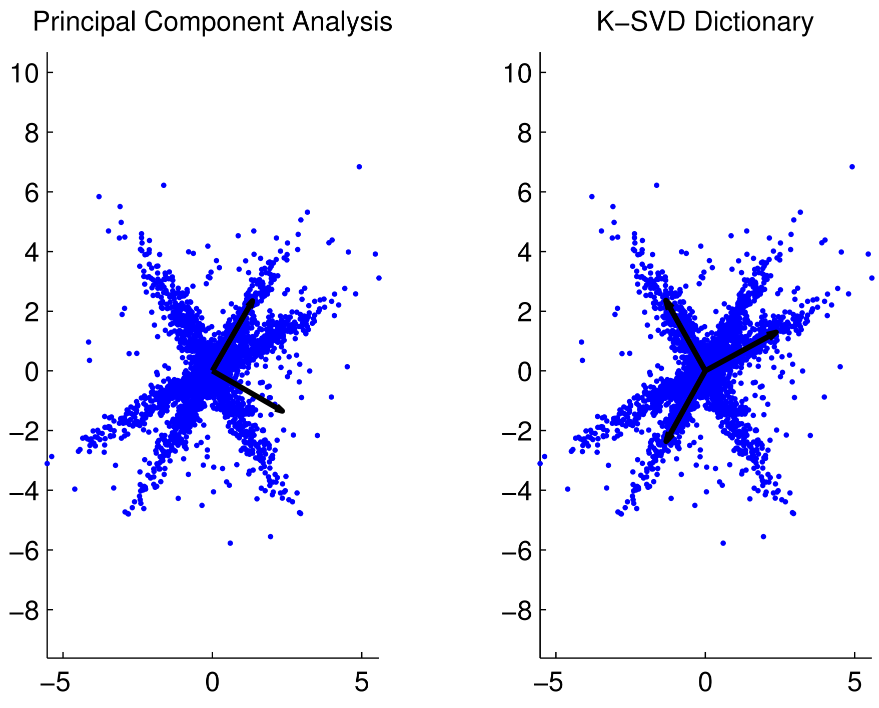

**Figure 8:** A training set consisting of 2-dimensional signals that follow three distinct directions, and representation atoms learned using either PCA or over-complete KSVD.

In the plot on the left side of Figure 8, the arrows depict the orthonormal basis learned using the PCA algorithm. The vector pointing to the upper-right corner (the principal component) identifies the direction that contains most of the variance of the dataset, whereas the other vector (minor component) is orthogonal to the first. In this case, the dictionary is complete, orthonormal and adapted to the dataset, but only the data that are lying along the principal component direction can be efficiently approximated with one coefficient using the corresponding atom.

On the other hand, the plot on the right of the figure shows an over-complete dictionary whose atoms are learned using the KSVD algorithm. In this case, we set a sparsity constraint `Tdata` which forces the data to be represented using only one coefficient:

```
SMALL.DL=SMALL_init_DL();
SMALL.DL.toolbox = 'KSVDS';
SMALL.DL.name = 'ksvds';
SMALL.DL.params = struct('data',X,'Tdata',1,'dictSize',nAtoms);
```

The initial dictionary is defined randomly, leading to vectors that point to arbitrary directions. After the algorithm runs for a sufficient number of iterations, the vectors are updated so that their directions lock to the three directions followed by the points in the dataset. In this case, most of the training samples can be efficiently represented using only one atom of the dictionary.

This simple example shows that moving from a complete, orthonormal transform to an over-complete dictionary learned using a sparsity criterion leads to a set of basis functions that are better adapted to the training set.

# 5 SMALLbox add–ons

It is possible to add functionalities to SMALLbox without affecting its core code by designing add-on modules. These will be typically implementation of algorithms published in papers and technical reports that use the SMALLbox framework, but that are not essential to the toolbox itself.

## 5.1 Configuration files structure

The two main wrapper functions `solve` and `learn` have two separate configuration files in the folder `{SMALLbox Root}/config`. In order to add a new entry to the solver or dictionary learning configuration files you should create a copy the original configuration file with `_local` in the name (for instance: the local version of the `SMALL_learn_config.m` file will be called `SMALL_learn_config_local.m`). The caller function (either `SMALL_learn` or `SMALL_solve`), at first looks for the existence of the local configuration file: if this file exists, it will run it; otherwise it will run the default configuration file. The main advantages of using this approach are as follows:

- if you are using a cloned repository version of SMALLbox, since the local versions of the configuration files are not under version control, changing any of these will not prompt you for uncommited changes in the repository;
- if you update SMALLbox to a later version, your local configuration file will not be overwritten, since it is not distributed with SMALLbox, and thus preventing you from losing any local changes you may have made.

A listing of the wrapper functions (and their configuration files) can be found in Appendix A. Please refer to Sec. 4.5.1 for a detailed example on how to add a new solver to SMALLbox.

## 5.2 Add–on Example: Incoherent Dictionary Learning

*This section refers to the* ***`incoherentdl`*** *algorithm code tagged as ver_ 1.1 in the project's Mercurial repository. To download this version please go to the project's homepage in* *`http://code.soundsoftware.ac.uk/projects/incoherentdl`.*

The incoherent dictionary learning add-on includes algorithms for learning dictionaries that are both adapted to the training data and that exhibit a low mutual coherence (defined as the maximum absolute inner product between any two different atoms) [27]. It extends the functionalities of the `SMALL_two_step_DL` toolbox which is included in the SMALLBox distribution, and is an example showing the capabilities of SMALLbox add–ons.

These are the steps needed to interface SMALLbox add-ons to the core SMALLbox distribution:

- Download and install the latest SMALLBox distribution from: `https://code.soundsoftware.ac.uk/projects/smallbox`
- Download the add-on distribution. In this case, the incoherent dictionary learning source is available from: `https://code.soundsoftware.ac.uk/projects/incoherentdl`
- Place the add-on code in any convenient location of your file system, and add this to your MATLAB path.
- Modify the relevant files in the folder `{SMALLBox Root}/config/` (see below)
- Run the `SMALLboxInit.m` script to set environmental variables.
- Run any function or script contained in the add-on.

In our case, incoherent dictionary learning extends `SMALL_two_step_DL`. In order to extend this function, a local copy of this file was created, and named `SMALL_incoherentDL.m`. As can be seen in Listing 2, a new dictionary update algorithm (`mocod`) is introduced. By having an add–on specific version of this file we guarantee that the new algorithm works correctly with the SMALLbox default settings, without having to make any changes to the core. The reasons for this approach are the same as the ones supporting the functioning of the configuration files (see Sec. 5.1).

Here the `switch` statement contains different decorrelation algorithms that are contained in the add–on and are called by the function `SMALL_two_step_DL`. Whenever another add-on was designed with a new decorrelation method, it could be simply interfaced to SMALLbox by adding an additional case to the `switch` block.

In order to run the test functions included in the `incoherentdl` add–on, we need to extend the list of available dictionary learning tools; this is done by adding a new case to the SMALL learn wrapper function. To do so the user needs to copy the code snippet that defines the new method (as shown in Listing 3), by copying the code snippet of Listing 3 to the local `SMALL_learn` configuration file (located in `SMALLBOX_PATH/config/`↩ `SMALL_learn_config_local.m`).

**Listing 2:** Snippet from file `SMALL_incoherentDL.m`

```
    % main loop %

    for i = 1:iternum
        Problem.A = dico;
        solver = SMALL_solve(Problem, solver);

          %DICTIONARY UPDATE STEP
          if strcmpi(typeUpdate,'mocod')          %if update is MOCOD create parameters structure
              mocodParams = struct('zeta',DL.param.zeta,...   %coherence regularization factor
                  'eta',DL.param.eta,...      %atoms norm regularization factor
                  'Dprev',dico);              %previous dictionary
              % dico = dico_update(dico,sig,solver.solution,typeUpdate,flow,learningRate,mocodParams);
              if ~isfield(DL.param,'decFcn'), DL.param.decFcn = 'none'; end

              dico = dico_update_mocod(dico,sig,solver.solution,typeUpdate,flow,learningRate,mocodParams);

          else
              [dico, solver.solution] = dico_update(dico, sig, solver.solution, ...
                  typeUpdate, flow, learningRate);
              dico = normcols(dico);
          end

          switch lower(DL.param.decFcn)
              case 'ink-svd'
                  dico = dico_decorr_symetric(dico,mu,solver.solution);
              case 'grassmannian'
                  [n m] = size(dico);
                  dico = grassmannian(n,m,[],0.9,0.99,dico);
              case 'shrinkgram'
                  dico = shrinkgram(dico,mu);
              case 'iterproj'
                  dico = iterativeprojections(dico,mu,Problem.b1,solver.solution);
              otherwise
          end

    %     [dico, solver.solution] = dico_update(dico, sig, solver.solution, ...
    %         typeUpdate, flow, learningRate);
    %     if (decorrelate)
    %         dico = dico_decorr(dico, mu, solver.solution);
    %     end

        if ((show_dictionary)&&(mod(i,show_iter)==0))
            dictimg = SMALL_showdict(dico,[8 8],...
                 round(sqrt(size(dico,2))),round(sqrt(size(dico,2))),'lines','highcontrast');
            figure(2); imagesc(dictimg);colormap(gray);axis off; axis image;
            pause(0.02);
        end
    end
```

**Listing 3:** Code that needs to be added to the local configuration file in order to run `incoherentdl`

```
elseif strcmpi(DL.toolbox, 'SMALL_incoherentDL')
    DL=SMALL_incoherentDL(Problem, DL);

    %   we need to make sure that columns are normalised to
    %     unit lenght.
    for i = 1: size(DL.D,2)
        DL.D(:,i)=DL.D(:,i)/norm(DL.D(:,i));
    end
    D = DL.D;
```

# 6 Acknowledgments

## 6.1 SMALLbox

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

## 6.2 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.

# Bibliography

[1] David L. Donoho, Victoria Stodden, and Yaakov Tsaig. Sparselab, 2007.

[2] Thomas Blumensath and Michael E. Davies. Gradient pursuits. *IEEE Transactions on Signal Processing*, 56(6):2370–2382, 2008.

[3] E. van den Berg and M. P. Friedlander. Probing the pareto frontier for basis pursuit solutions. *SIAM Journal on Scientific Computing*, 31(2):890–912, 2008.

[4] Mário A. T. Figueiredo, Robert D. Nowak, and Stephen J. Wright. Gradient projection for sparse reconstruction: Application to compressed sensing and other inverse problems, 2007.

[5] Ewout van den Berg, Michael P. Friedlander, Gilles Hennenfent, Felix J. Herrmann, Rayan Saab, and Özgür Yilmaz. Algorithm 890: Sparco: A testing framework for sparse reconstruction. *ACM Transactions on Mathematical Software*, 35(4):1–16, February 2009.

[6] M Aharon, M Elad, and A Bruckstein. K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on Signal Processing*, 54(11):4311–4322, 2006.

[7] R Rubinstein, M Zibulevsky, and M Elad. Double Sparsity: Learning Sparse Dictionaries for Sparse Signal Approximation, 2010.

[8] Julien Mairal, Francis Bach, Jean Ponce, and Guillermo Sapiro. Online dictionary learning for sparse coding. *Proceedings of the 26th Annual International Conference on Machine Learning ICML 09*, 1(June):1–8, 2009.

[9] S G Mallat. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 41(12):3397–3415, 1993.

[10] Y C Pati, R Rezaiifar, and P S Krishnaprasad. Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition. *Proceedings of 27th Asilomar Conference on Signals Systems and Computers*, 1:40–44, 1993.

[11] Remi Gribonval and Karin Schnass. Some Recovery Conditions for Basis Learning by L1-Minimization. *ISCCSP*, 2008.

[12] L Rebollo-Neira. Dictionary redundancy elimination. *Image Rochester NY*, 151(1):0–3, 2004.

[13] M Elad and M Aharon. Image Denoising Via Learned Dictionaries and Sparse representation. *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Volume 1 CVPR06*, 1(c):895–900, 2006.

[14] M. D. Plumbley. Dictionary learning for L1-exact sparse coding. In M. E. Davies, C. J. James, S. A. Abdallah, and M. D. Plumbley, editors, *Proceedings of the 7th International Conference on Independent Component Analysis and Signal Separation (ICA)*, pages 406–413, London, UK, 2007. Springer, Berlin.

[15] Michael Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer, 2010.

[16] Volkan Cevher. On Accelerated Hard Thresholding Methods for Sparse Approximation. Technical report, 2011.

[17] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 1.21. `http://cvxr.com/cvx`, April 2011.

[18] A Adler, V Emiya, M G Jafari, M Elad, R Gribonval, and M D Plumbley. Audio Inpainting. *Audio Speech and Language Processing IEEE Transactions on*, 20(3):1, 2011.

[19] M Yaghoobi, T Blumensath, and M E Davies. Dictionary Learning for Sparse Approximations With the Majorization Method. *IEEE Transactions on Signal Processing*, 57(6):2178–2191, 2009.

[20] K. Skretting and K. Engang. Recursive Least Squares Dictionary Learning Algorithm. *IEEE Transaction on Signal Processing*, 58(4):2121–2130, 2010.

[21] K Engan, S O Aase, and J Hakon Husoy. Method of optimal directions for frame design. *1999 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings ICASSP99 Cat No99CH36258*, 5:2443–2446 vol.5, 1999.

[22] Kenneth Kreutz-Delgado, Joseph F Murray, Bhaskar D Rao, Kjersti Engan, Te-Won Lee, and Terrence J Sejnowski. Dictionary learning algorithms for sparse representation. *Neural Computation*, 15(2):349–396, 2003.

[23] B A Olshausen and D J Field. Sparse coding with an overcomplete basis set: A strategy employed by V1? *Vision Research*, 37(23):3311–3325, 1997.

[24] Boris Mailhé, Sylvain Lesage, Rémi Gribonval, Pierre Vandergheynst, and Frédéric Bimbot. Shift-invariant dictionary learning for sparse representations: extending K-SVD. In *European Conference on Signal Processing 2008*, 2008.

[25] Boris Mailhé and Mark D. Plumbley. Dictionary learning with large step gradient descent for sparse representations. In *LVA/ICA*, pages 231–238, 2012.

[26] Boris Mailhé, Daniele Barchiesi, and Mark D. Plumbley. Ink-svd: Learning incoherent dictionaries for sparse representations. In *ICASSP'2012*, pages 231–238, 2012.

[27] D. Barchiesi and M. D. Plumbley. Learning incoherent dictionaries for sparse approximation using iterative projections and rotations. *IEEE Transactions on Audio Speech and Language Processing (submitted)*, February 2012.

# A Appendix: SMALLbox Wrappers

There are two main wrapper functions in SMALLbox: `SMALL_solve` and `SMALL_learn`. These provide the necessary interface to add new sparse approximation and dictionary learning tools without the need of modifying the core. To see an example please refer to 5.

Each wrapper function has its corespondent configuration file, as detailed in Section 5.1. The configuration files are located in the `config` folder.

## A.1 Solvers wrapper function

```
function solver = SMALL_solve(Problem, solver)
%% SMALL sparse solver caller function
%
%   Function gets as input SMALL structure that contains SPARCO problem to
%   be solved, name of the toolbox and solver, and parameters file for
%   particular solver.
%
%   Outputs are solution, reconstructed signal and time spent

%   Centre for Digital Music, Queen Mary, University of London.
%   This file copyright 2009 Ivan Damnjanovic.
%
%   This program is free software; you can redistribute it and/or
%   modify it under the terms of the GNU General Public License as
%   published by the Free Software Foundation; either version 2 of the
%   License, or (at your option) any later version.  See the file
%   COPYING included with this distribution for more information.
%
%%

SMALLboxInit

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
% if signal that needs to be represented is different then training set for
% dictionary learning it should be stored in Problem.b1 matix
if isfield(Problem, 'b1')
```

```
    b = Problem.b1;
else
    b = Problem.b;             % The right-hand-side vector.
end
%%
if (solver.profile)
    fprintf('\nStarting solver %s... \n', solver.name);
end

start=cputime;
tStart=tic;

%% solvers configuration
% test if there is a locally modified version of the config
% otherwise reads the "default" config file
if exist(fullfile(SMALL_path, 'config/SMALL_solve_config_local.m'), 'file') ==↩
     2
    run(fullfile(SMALL_path, 'config/SMALL_solve_config_local.m'));
else
    run(fullfile(SMALL_path, 'config/SMALL_solve_config.m'));
end

%%
%   Sparse representation time
tElapsed=toc(tStart);
solver.time = cputime - start;
if (solver.profile)
    fprintf('Solver %s finished task in %2f seconds (cpu time). \n', solver.↩
        name, solver.time);
    fprintf('Solver %s finished task in %2f seconds (tic-toc time). \n', ↩
        solver.name, tElapsed);
end
solver.time=tElapsed;
% geting around out of memory problem when converting big matrix from
% sparse to full...

if isfield(Problem, 'sparse')&&(Problem.sparse==1)
    solver.solution = y;
else
    solver.solution = full(y);
end
if isfield(Problem,'reconstruct')
    %   Reconstruct the signal from the solution
    solver.reconstructed  = Problem.reconstruct(solver.solution);
end
end
```

```
%% Configuration file used in SMALL_solve
%
%   Please DO NOT use this file to change the solvers used in SMALLBox
%   If you want to change the solvers create a copy
%     of this file named 'SMALL_learn_config_local.m'
%
%   Please refer to the documentation for further information

%   Centre for Digital Music, Queen Mary, University of London.
%   This file copyright 2009 Ivan Damnjanovic.
%
%   This program is free software; you can redistribute it and/or
%   modify it under the terms of the GNU General Public License as
%   published by the Free Software Foundation; either version 2 of the
%   License, or (at your option) any later version.  See the file
%   COPYING included with this distribution for more information.
%
```

```
%%


if strcmpi(solver.toolbox,'sparselab')
    y = eval([solver.name,'(SparseLab_A, b, n,',solver.param,');']);
elseif strcmpi(solver.toolbox,'sparsify')
    if isa(Problem.A,'float')
        y = eval([solver.name,'(b, A, n,',solver.param,');']);
    else
        y = eval([solver.name,'(b, A, n, ''P_trans'', AT,',solver.param,');'])↩
            ;
    end
elseif (strcmpi(solver.toolbox,'spgl1')||strcmpi(solver.toolbox,'gpsr'))
    y = eval([solver.name,'(b, A,',solver.param,');']);
elseif (strcmpi(solver.toolbox,'SPAMS'))
    y = eval([solver.name,'(b, A, solver.param);']);
elseif (strcmpi(solver.toolbox,'SMALL'))
    if isa(Problem.A,'float')
        y = eval([solver.name,'(A, b, n,',solver.param,');']);
    else
        y = eval([solver.name,'(A, b, n,',solver.param,',AT);']);
    end
elseif (strcmpi(solver.toolbox, 'ompbox'))
    G=A'*A;
    epsilon=solver.param.epsilon;
    maxatoms=solver.param.maxatoms;
    y = eval([solver.name,'(A, b, G,epsilon,''maxatoms'',maxatoms,''checkdict'↩
        ',''off'');']);
    % danieleb: added call to omp functions with fast implementation.
elseif (strcmpi(solver.toolbox, 'ompbox_fast'))
    DtX=A'*b;
    XtX = sum(b.*b);
    G=A'*A;
    epsilon=solver.param.epsilon;
    maxatoms=solver.param.maxatoms;
    y = eval([solver.name,'(DtX, XtX, G,epsilon,''maxatoms'',maxatoms,''↩
        checkdict'',''off'');']);
elseif (strcmpi(solver.toolbox, 'ompsbox'))
    basedict = Problem.basedict;
    if issparse(Problem.A)
        A = Problem.A;
    else
        A = sparse(Problem.A);
    end
    G = dicttsep(basedict,A,dictsep(basedict,A,speye(size(A,2))));
    epsilon=solver.param.epsilon;
    maxatoms=solver.param.maxatoms;
    y = eval([solver.name,'(basedict, A, b, G,epsilon,''maxatoms'',maxatoms,''↩
        checkdict'',''off'');']);
    Problem.sparse=1;
elseif (strcmpi(solver.toolbox, 'ALPS'))
    if ~isa(Problem.A,'float')
        % ALPS does not accept implicit dictionary definition
        A = opToMatrix(Problem.A, 1);
    end
    [y, numiter, time, y_path] = wrapper_ALPS_toolbox(b, A, solver.param);
elseif (strcmpi(solver.toolbox, 'MMbox'))
    if ~isa(Problem.A,'float')
        % MMbox does not accept implicit dictionary definition
        A = opToMatrix(Problem.A, 1);
    end

    [y, cost] = wrapper_mm_solver(b, A, solver.param);

    %%
    %   Please do not make any changes to the 'SMALL_solve_config.m' file
    %   All the changes should be done to your local configuration file
    %    named 'SMALL_solve_config_local.m'
    %
    %   To introduce new sparse representation algorithm put the files in
    %   your Matlab path. Next, unique name <TolboxID> for your toolbox and
    %   prefferd API <Preffered_API> needs to be defined.
    %
    % elseif strcmpi(solver.toolbox,'<ToolboxID>')
    %
    %     % - Evaluate the function (solver.name - defined in the main) with
    %     %   parameters given above
    %
    %     y = eval([solver.name,'(<Preffered_API>);']);

else
    printf('\nToolbox has not been registered. Please change SMALL_solver file↩
        .\n');
    return
end
```

## A.2 Dictionary Learning wrapper function

```
function DL = SMALL_learn(Problem,DL)
%% SMALL Dictionary Learning
%
%   Function gets as input Problem and Dictionary Learning (DL) structures
%   In Problem structure field b with the training set needs to be defined
%   In DL fields with name of the toolbox and solver, and parameters file
%   for particular dictionary learning technique needs to be present.
%
%   Outputs are Learned dictionary and time spent as a part of DL structure

%
%   Centre for Digital Music, Queen Mary, University of London.
%   This file copyright 2009 Ivan Damnjanovic.
%
%   This program is free software; you can redistribute it and/or
%   modify it under the terms of the GNU General Public License as
%   published by the Free Software Foundation; either version 2 of the
%   License, or (at your option) any later version.  See the file
%   COPYING included with this distribution for more information.
%%

SMALLboxInit

if (DL.profile)
    fprintf('\nStarting Dictionary Learning %s... \n', DL.name);
end

start=cputime;
tStart=tic;

%% toolbox configuration
% test if there is a locally modified version of the config
% otherwise reads the "default" config file
if exist(fullfile(SMALL_path, 'config/SMALL_learn_config_local.m'), 'file') ==↩
     2
    printf('\n\nSMALL_learn: Using local configuration file.\n\n');
    run(fullfile(SMALL_path, 'config/SMALL_learn_config_local.m'));
else
    printf('\n\nSMALL_learn: Using default configuration file.\n\n');
    run(fullfile(SMALL_path, 'config/SMALL_learn_config.m'));
```

```
end

%%
%   Dictionary Learning time
tElapsed=toc(tStart);
DL.time = cputime - start;
if (DL.profile)
    fprintf('\n%s finished task in %2f seconds (cpu time). \n', DL.name, DL.↩
        time);
    fprintf('\n%s finished task in %2f seconds (tic-toc time). \n', DL.name, ↩
        tElapsed);
end
DL.time=tElapsed;
%   If dictionary is given as a sparse matrix change it to full

DL.D = full(D);

end
```

```
%% Configuration file used in SMALL_learn
%
%   Please DO NOT use this file to change the dictionary learning algorithms ↩
    in SMALLBox
%   If you want to change the dictionary learning algorithms
%     create a copy of this file named 'SMALL_learn_config_local.m'
%
%   Please refer to the documentation for further information

%   Centre for Digital Music, Queen Mary, University of London.
%   This file copyright 2009 Ivan Damnjanovic.
%
%   This program is free software; you can redistribute it and/or
%   modify it under the terms of the GNU General Public License as
%   published by the Free Software Foundation; either version 2 of the
%   License, or (at your option) any later version.  See the file
%   COPYING included with this distribution for more information.
%
%%

if strcmpi(DL.toolbox,'KSVD')
    param=DL.param;
    param.data=Problem.b;

    D = eval([DL.name,'(param)']);%, ''t'', 5);']);
elseif strcmpi(DL.toolbox,'KSVDS')
    param=DL.param;
    param.data=Problem.b;

    D = eval([DL.name,'(param, ''t'', 5);']);
elseif strcmpi(DL.toolbox,'SPAMS')

    X  = Problem.b;
    param=DL.param;

    D = eval([DL.name,'(X, param);']);
    %   As some versions of SPAMS does not produce unit norm column
    %   dictionaries, we need to make sure that columns are normalised to
    %   unit lenght.

    for i = 1: size(D,2)
        D(:,i)=D(:,i)/norm(D(:,i));
    end
elseif strcmpi(DL.toolbox,'SMALL')
```

```
    X   = Problem.b;
    param=DL.param;

    D = eval([DL.name,'(X, param);']);
    %   we need to make sure that columns are normalised to
    %   unit lenght.

    for i = 1: size(D,2)
        D(:,i)=D(:,i)/norm(D(:,i));
    end

elseif strcmpi(DL.toolbox,'TwoStepDL')

    DL=SMALL_two_step_DL(Problem, DL);

    %   we need to make sure that columns are normalised to
    %   unit lenght.

    for i = 1: size(DL.D,2)
        DL.D(:,i)=DL.D(:,i)/norm(DL.D(:,i));
    end
    D = DL.D;

elseif strcmpi(DL.toolbox,'MMbox')

    DL = wrapper_mm_DL(Problem, DL);

    %   we need to make sure that columns are normalised to
    %   unit lenght.

    for i = 1: size(DL.D,2)
        DL.D(:,i)=DL.D(:,i)/norm(DL.D(:,i));
    end
    D = DL.D;

%%
%   Please do not make any changes to the 'SMALL_learn_config.m' file
%   All the changes should be done to your local configuration file
%    named 'SMALL_learn_config_local.m'
%
%   To introduce new dictionary learning technique put the files in
%   your Matlab path. Next, unique name <TolboxID> for your toolbox needs
%   to be defined and also prefferd API for toolbox functions <Preffered_API>
%
% elseif strcmpi(DL.toolbox,'<ToolboxID>')
%     % This is an example of API that can be used:
%     % - get training set from Problem part of structure
%     % - assign parameters defined in the main program
%
%     X   = Problem.b;
%     param=DL.param;
%
%     % - Evaluate the function (DL.name - defined in the main) with
%     %   parameters given above
%
%     D = eval([DL.name,'(<Preffered_API>);']);

else
    printf('\nToolbox has not been registered. Please change SMALL_learn file↩
        .\n');
    return
end
```