UNLocBoX: USER'S GUIDE

MATLAB CONVEX OPTIMIZATION TOOLBOX

Lausanne - February 2014

PERRAUDIN Nathanaël, KALOFOLIAS Vassilis

LTS2 - EPFL

Abstract

Nowadays the trend to solve optimization problems is to use specific algorithms rather than very general ones. The UNLocBoX provides a general framework allowing the user to design his own algorithms. To do so, the framework try to stay as close from the mathematical problem as possible. More precisely, the UNLocBoX is a Matlab toolbox designed to solve convex optimization problem of the form

$$\min_{x\in\mathscr{C}} \sum_{n=1}^{K} f_n(x),$$

using proximal splitting techniques. It is mainly composed of solvers, proximal operators and demonstration files allowing the user to quickly implement a problem.

Contents

1 Introduction **1**

2 Literature **2**

3 Installation and initialization **2**

3.1 Dependencies 2

3.2 GPU computation 2

4 Structure of the UNLocBoX **2**

5 Problems of interest **4**

6 Solvers **4**

6.1 Defining functions 5

6.2 Selecting a solver 5

6.3 Optional parameters 6

6.4 Plug-ins: how to tune your algorithm 6

6.5 Returned arguments 8

7 Proximal operators **9**

7.1 Constraints 9

8 Example **10**

References **16**

1 Introduction

During your research, you have most likely faced the situation where you need to solve a convex optimization problem. Maybe you were trying to find an optimal combination of parameters, you were transforming some process into something automatic or, even more simply, your algorithm was equivalent to solving convex problem. In this situation, you usually face a dilemma: you need a specific algorithm to solve your problem, but you do not want to spend hours writing it. And if you are not an expert in convex optimization and you do not want to spend weeks learning it.

If you are in this situation, you probably want to have a go with the UNLocBoX, a MATLAB convex optimization toolbox based on proximal splitting methods[1]. This tool allows you to efficiently deal with convex problems thanks to a quick and simple implementation. For instance, to solve the problem

$$\arg\min_x \|Ax-y\|_2^2 + \|x\|_1$$

where $\|\cdot\|_2$, $\|\cdot\|_1$ denote the ℓ_2 and the ℓ_1 norms, you will write

```
f1.eval = @(x) norm( A*x - y )^2;
f1.grad = @(x) 2 * A' * ( A*x -y );
f1.beta = 2 * norm(A)^2;

f2.eval = @(x) norm( x, 1 );
f2.prox = @(x,T) prox_l1( x, T );

sol = solvep( x0, {f1,f2} );
```

In this case, the UNLocBoX will automatically select a suitable solver for your problem and compute the answer with the chosen method. The UNLocBoX supports an unlimited number of convex functions and constraints. As you saw in the previous example, it only requires gradients for differentiable functions, proximal operators[2] for non-smooth functions and projections for hard constraints. For efficient implementation, many proximal operators and projections are already provided with the toolbox. This allows a great freedom for the range of solvable problems and also a quick implementation.

The UNLocBoX possesses a framework well suited for complex convex optimization problem with a extensive list of useful proximal and projection operator, solver and other utility functions. However it is not a complete tool that you can use blindly without knowing anything about the magic inside the "black" box. If this is what you're looking for, try for instance CVX. Otherwise (or if CVX can not handle your problem), the UNLocBoX provides you a very good compromise between a fast implementation and a very high customizability.

The documentation of the UNLocBoX is complete, meaning that every single function is documented. Even if it is not perfect, it has improved significantly with time and we expect it to continue to do so. You can find it online at `http://unlocbox.sourceforge.net/doc`. Everything is also periodically updated into a big PDF file.

In this document, you will find the basic concepts and the structure of the UNLocBoX along with the traditional "installation instructions". This includes some reminders about convex optimization and proximal splitting methods. For those of you, who starts with convex optimization, we will provide a sufficient amount of selected literature on the subject to help you learn about it. If you just need a quick introduction to the UNLocboX, a tutorial can be found at `https://lts2.epfl.ch/unlocbox/notes/unlocbox-note-008.pdf`. Finally, if you use the UNLocBoX, please cite [**?**].

[1]It has recently been ported to python as the PyUNLocBoX.

[2]If you do not know what a proximal operator is, do not worry. We will provide a basic introduction with some references to help you.

2 Literature

to be done

To continue further learning about the UNLocBoX you should now read the tutorial https://lts2.epfl.ch/unlocbox/notes/unlocbox-note-008.pdf.

3 Installation and initialization

First, download the latest version for your operating system on https://lts2.epfl.ch/unlocbox/download.php. Then extract the archive in the directory of your choice. Finally, to start the toolbox, just run the function *init_unlocbox.m* at the root of the UNLocBoX directory. You can also work with the development version by cloning the git repository at:

https://github.com/nperraud/unlocbox.git

3.1 Dependencies

- Some functions of the UNLocBoX depend on the LTFAT toolbox (mostly demonstration). You can download it at

  https://github.com/epfl-lts2/unlocbox.

- To solve graph related convex optimization problems, you can use the GSPBox. You can download it at

  https://lts2research.epfl.ch/gsp.

3.2 GPU computation

If your system support it, usage of the GPU processor for some of the computations can reduce considerably the running time of the optimization algorithms. For numerous applications, it is worth using the GPU to optimize speed. However, keep in mind that the UNLocBoX does not provide optimized code. For high-performance or large-scale data, you may need to use another language than MATLAB.

To use the GPU with UnLocBox, you first need a compatible NVIDIA graphic card. Then you need to install and configure CUDA, available at: http://developer.nvidia.com/cuda-downloads. Once this is done you can set the global variable *GLOBAL_useGPU* into the script init_unlocbox to 1. This will enable the permanent use of the GPU for the TV proximal operator. The toolbox will then automatically use the GPU if it is supported. GPU computation is under development.

For now, only the TV proximal operators use the GPU. However, if you know what you are doing, you might want to push your data problem onto the GPU to avoid big data transfers between the CPU and the GPU. In this case, you should be able to run the UNLocBoX. The script demo_gpu presented in the demos compares the efficiency of the different methods. Below is a sample of results obtained with Intel-i7 2.4Ghz processor and a NVIDIA GeForce GT 650M graphic card.

```
Computation time with the GPU: 15.3013
Computation time with the CPU: 26.5361
Computation time all on the GPU: 11.9943
```

4 Structure of the UNLocBoX

The toolbox is composed of the following functions

- Solvers : the core of the toolbox. It contains most of the recent techniques, like forward-backward (FISTA), Douglas-Rachford, PPXA, ADMM, SDMM and others.

- Proximal operators: many pre-defined proximal operators help the user to quickly solve many standard problems. The toolbox also includes projection operators to handle hard constraint.

- Demo files: help the user to start with the toolbox.

- Sample signals: mostly used in the demonstration files.
- Utility functions: some useful functions.

5 Problems of interest

The UNLocBoX is designed to solve convex optimization problems of the form

$$\min_{x\in\mathbb{R}^N} f_1(x)+f_2(x), \tag{1}$$

or more generally

$$\min_{x\in\mathbb{R}^N} \sum_{n=1}^{K} f_n(x), \tag{2}$$

where the f_i are lower semi-continuous convex functions from $\mathbb{R}^N$ to $(-\infty,+\infty]$. We assume $\lim_{\|x\|_2\to\infty}\left\{\sum_{n=1}^{K} f_n(x)\right\}=\infty$ and the f_i have non-empty domains, where the domain of a function f is given by

$$\operatorname{dom} f := \{x\in\mathbb{R}^n : f(x) < +\infty\}.$$

In problem (2), and when both f_1 and f_2 are smooth functions, gradient descent methods can be used to solve (1); however, gradient descent methods cannot be used to solve (1) when f_1 and/or f_2 are not smooth. In order to solve such problems more generally, we implement several algorithms including the forward-backward algorithm [**?**]- [**?**] and the Douglas-Rachford algorithm [**?**, **?**]- [**?**].[3]

Both the forward-backward and Douglas-Rachford algorithms fall into the class of *proximal splitting algorithms*. The term *proximal* refers to their use of proximity operators, which are generalizations of convex projection operators. The *proximity operator* of a lower semi-continuous convex function $f:\mathbb{R}^N\to\mathbb{R}$ is defined by

$$\operatorname{prox}_f(x) := \arg\min_{y\in\mathbb{R}^N}\left\{\frac{1}{2}\|x-y\|_2^2+f(y)\right\}. \tag{3}$$

Note that the minimization problem in (3) has a unique solution for every $x\in\mathbb{R}^N$, so $\operatorname{prox}_f:\mathbb{R}^N\to\mathbb{R}^N$ is well-defined. The proximity operator is a useful tool because (see, e.g., [**?**, **?**]) x^* is a minimizer in (1) if and only if for any $\gamma>0$,

$$x^* = \operatorname{prox}_{\gamma(f_1+f_2)}(x^*). \tag{4}$$

The term *splitting* refers to the fact that the proximal splitting algorithms do not directly evaluate the proximity operator $\operatorname{prox}_{\gamma(f_1+f_2)}(x)$, but rather try to find a solution to (4) through sequences of computations involving the proximity operators $\operatorname{prox}_{\gamma f_1}(x)$ and $\operatorname{prox}_{\gamma f_2}(x)$ separately. The recent survey [**?**] provides an excellent review of proximal splitting algorithms used to solve (1) and related convex optimization problems.

6 Solvers

The UNLocBoX toolbox is composed of many solvers. We categorize them into 2 different groups. First, there are specific solvers that minimize only two functions (Forward backward, Douglas Rachford, ADMM,...). Those are usually more efficient. Second, there are general solvers that are more general but less efficient (Generalized forward backward, PPXA, SDMM,...). Not all possible solvers are included into the UNLocBoX. However it offers a general framework, where you can add you own solvers.

In general a solver takes three kinds of inputs: an initialization point x_0, the functions to be minimized and an optional structure of parameters. As a rule into the UNLocBoX, we use the following convention. The initialization point is always the first argument (except for SDMM), then comes the functions and finally the optional structure of parameters.

[3]In fact, the toolbox implements generalized versions of these algorithms that can solve problems with sums of a general number of such functions, but for simplicity, we discuss here the simplest case of two functions.

6.1 Defining functions

Each function $f_k(x)$ is modeled as a structure containing different fields: `eval`, `prox`, `grad`, `beta`. For a function `f`, the field `f.eval` is a Matlab function handle that takes as input the optimization variables x and returns the value $f(x)$. Then the field `f.prox` is another function handle that takes as input a vector x along with a positive real number τ and returns the vector $\mathrm{prox}_{\tau f}(x)$. In MATLAB, we write:

```
      f1.prox = @(x, T) prox_f1(x, T)
```

where `prox_f1(x, T)` can be a standard MATLAB function that solves the problem $\mathrm{prox}_{T f_1}(x)$ given in eq. (3)). In the same way, `f.grad` is a handle of a function that takes as input the optimization variables x and returns the vector $\nabla f(x)$. In Matlab, write:

```
      f.grad = @(x) grad_f(x)
```

where `grad_f(x)` returns the value of $\nabla f(x)$. Finally the field `beta` contains the Lipschitz constant of the gradient. A function f has a β-Lipschitz-continuous gradient ∇f if

$$\|\nabla f(x) - \nabla f(y)\|_2 \leqslant \beta \|x-y\|_2 \qquad \forall x, y \in \mathbb{R}^N, \tag{5}$$

where $\beta > 0$.

For instance to define the function $f(x) = 5\|x-y\|_2^2$, you would write:

```
      tau = 5;
      f.eval = @(x) tau * norm(x);
      f.grad = @(x) 2 * tau * (x - y);
      f.beta = 2 * tau;
```

If the function is not smooth, or for another reason, you prefer to use the proximal operator, you would write:

```
      tau = 5;
      f.eval = @(x) tau * norm(x);
      paramf.y = y;
      f.prox = @(x,T) prox_l2( x, tau*T , paramf );
```

Here we use a proximal operator available in the UNLocBoX, but you can also provide your personal function.

6.2 Selecting a solver

The UNLocBoX contains a general solving function called `solvep`. For many problems, it can be used blindly. This function will compute the time-step for you and select a compatible algorithm with your problem. To minimize the sum of the functions `f1, f2, f3`, write in MATLAB:

```
      sol = solvep(x_0, {f1, f2, f3});
```

If your problem becomes more complex to solve, you can select manually the solver or write a specific one for your problem. In many situation, this last option may be a good choice. Thanks to the framework of the UNLocBoX, this should not be too painful. Below is a detailed example of a solver.

```
function s = demo_forward_backward_alg()
%DEMO_FORWARD_BACKWARD_ALG Demonstration to define a personal solver
%   Usage : param.algo = demo_forward_backward_alg();
%
%   This function returns a structure containing the algorithm. You can
%   lauch your personal algorithm with the following::
%
%           param.algo = demo_forward_backward_alg();
%           sol = solvep(x0, {f1, f2}, param);
%

    % This function returns a structure with 4 fields:
    % 1) The name of the solver. This is used to select the solvers.
    s.name = 'DEMO_FORWARD_BACKWARD';
    % 2) A method to initialize the solver (called at the beginning)
    s.initialize = @(x_0, fg, Fp, param) ...
      forward_backward_initialize(x_0,fg,Fp,param);
    % 3) The algorithm itself (called at each iterations)
    s.algorithm = @(x_0, fg, Fp, sol, s, param) ...
      forward_backward_algorithm(fg, Fp, sol, s, param);
    % 4) Post process method (called at the end)
    s.finalize = @(x_0, fg, Fp, sol, s, param) sol;
    % The variables here are
    %   x_0 : The starting point
    %   fg  : A single smooth function
    %         (if fg.beta == 0, no smooth function is specified)
    %   Fp  : The non smooth functions (a cell array of structure)
    %   param: The structure of optional parameter
    %   s   : Intern variables or the algorithm
    %   sol : Current solution
end

function [sol, s, param] = forward_backward_initialize(x_0,fg,Fp,param)

    % Handle optional parameter. Here we need a variable lambda.
    if ~isfield(param, 'lambda'), param.lambda=1 ; end

    % All intern variables are stored into the structure s
    s = struct;
    % *sol* is set to the initial points
    sol = x_0;

    if numel(Fp)>1
        error(['This solver can not be used to optimize',...
          ' more than one non smooth function']);
    end

    if ~fg.beta
        error('Beta = 0! This solver requires a smooth term.');
    end

end

function [sol, s] = forward_backward_algorithm(fg, Fp, sol, s, param)
    % The forward backward algorithm is done in two steps
    %  1) x_n = prox_{f, gamma} ( sol - gamma grad_fg(sol) )
    s.x_n = Fp{1}.prox( sol - param.gamma*fg.grad(sol), param.gamma);
    %  2) Updates
    %     sol = sol + lambda * (x_n -sol)
    sol = sol + param.lambda * (s.x_n - sol);
end
```

6.3 Optional parameters

The optional parameters for solvers are all contained into a structure `param`. Different functions might need different parameters, but most of them are common. Table 6.3 presents a list of recurrent optional parameters for solvers and their default values.

6.4 Plug-ins: how to tune your algorithm

It is sometimes useful to be able to visualize your current solution every iterations or to be able to change the time-step when the algorithm is already launched. To solve this problem, the user can use plug-in functions that will be evaluated at the end of each iteration.

We first present a plug-in to change the time step at each iteration. It can be used with:

Parameter	**Explanation**	**Default value**
`param.tol`	This parameter is used to define the stopping criterion of the problem. The algorithm stops if $$\frac{f(t)-f(t-1)}{f(t)} < tol$$ where $f(t)$ is the function to be minimized at iteration t. $tol \in \mathbb{R}^*_+$	10^{-2}
`param.abs_tol`	This boolean parameter activates an alternative stopping criterion. The algorithm stops if $$f(t) < tol$$ where $f(t)$ is the function to be minimized at iteration t and $tol \in \mathbb{R}^*_+$ is given by `param.tol`	0
`param.use_dual`	If activated, use the norm of the dual variable instead of the evaluation of the function itself for stopping criterion. This is used in ADMM and SDMM for instance. To use it, the algorithm needs to store the dual variable in `s.dual_var`.	0
`param.maxit`	The maximum number of iterations	200
`param.verbose`	Log parameter: 0 no log, 1 a summary at convergence, 2 print main steps	1
`param.gamma`	step-size parameter γ. This constant should satisfy: $\gamma \in [\varepsilon, 2/\beta - \varepsilon]$, for $\varepsilon \in (0, \min\{1, 1/\beta\})$. If not set, it is automatically computed.	1
`param.algo`	Algorithm to solve the problem (See the example above)	

Table 1: Optional parameter for solvers

```
param.do_ts=@(x) log_decreasing_ts(x, gamma_in, gamma_fin, nit);
sol = solvep(x0, {f1,f2}, param);
```

Here `x` is a structure that contains the following field

- `x.sol` : The current solution.
- `x.iter` : The current number of iteration.
- `x.curr_eval` : The current evaluation of the objective function.
- `x.prev_eval` : The previous evaluation of the objective function.
- `x.gamma` : The current time-step.
- `x.objective` : The objective function over all iterations

Below, you can find the code of this plug-in as an example.

```
function gamma = log_decreasing_ts(x, gamma_in, gamma_fin, nit)
%LOG_DECREASING_TS Log decreasing timestep for UNLCOBOX algorithm
%   Usage gamma = log_decreasing_ts(x, gamma_in, gamma_fin, nit);
%
%   Input parameters:
%         x         : Structure of data
%         gamma_in  : Initial timestep
%         gamma_fin : Final timestep
%         nit       : Number of iteration for the decrease
%
```

```
%   Output parameters:
%         gamma     : Timestep at iteration t
%
%   This plug-in computes a new timestep at each iteration. It makes a log
%   decreasing timestep from *gamma_in* to *gamma_fin* in *nit* iterations.
%   To use this plugin, define::
%
%       param.do_ts = @(x) log_decreasing_ts(x, gamma_in, gamma_fin, nit);
%
%   in the structure of optional argument of the solver.

    if x.iter > nit
        gamma = gamma_fin;
    else
        ts = gamma_in./linspace(1,gamma_in/gamma_fin,nit);
        gamma = ts(x.iter);
    end
end
```

As a second plug-in, we present a function that can display the objective function evolution trough iteration. This can be very useful for debugging. In MATLAB, write:

```
    fig = figure(100);
    param.do_sol = @(x) plot_objective(x, fig);
    sol = solvep(x0, {f1,f2}, param);
```

```
function [ sol ] = plot_objective(info_iter, fig)
%PLOT_OBJECTIVE Plot objective function over iterations
%   Usage [ sol ] = plot_objective( info_iter, fig );
%
%   Input parameters:
%         info_iter   : Structure of info
%         fig    : Figure
%
%   Output parameters:
%         sol    : Current solution
%
%   This plug-in displays the image every iterations of an algorithm. To use
%   the plug-in just define::
%
%       fig = figure(100);
%       param.do_sol = @(x) plot_objective(x, fig);
%
%   In the structure of optional argument of the solver.

% select the figure
if info_iter.iter<2
    figure(fig);
end

%
title(['Current it: ', num2str(info_iter.iter),'   Curr obj: ', ...
    num2str(info_iter.curr_norm)]);
semilogy(info_iter.objective); title('Objective function')
drawnow;

% return the solution
sol=info_iter.sol;

end
```

6.5 Returned arguments

The solver returns 3 arguments

- `sol` : The minimizer of the problem

- `info` : General information (summarized in Table 6.5)
- `objective` : the evolution of the objective function through iterations

Field	Explanation
`info.algo`	Algorithm used
`info.iter`	Number of iterations
`info.time`	Time of execution of the function in sec.
`info.final_eval`	Final evaluation of the objective function
`info.crit`	Stopping criterion used see table 6.5
`info.rel_norm`	Relative norm at convergence

Table 2: Information returned by the solver

Criterion	Explanation
`TOL_EPS`	Tolerance achieved
`ABS_TOL`	Objective function below the tolerance
`MAX_IT`	Maximum number of iterations
`USER`	Stop by the user "ctrl + D" in the command window.
`--`	Other

Table 3: Stopping criteria

7 Proximal operators

In the UNLocBoX, a variety of proximal operators are already implemented. All proximal operator functions take as input three parameters: `x, lambda, param`. First, `x` is the initial signal. Then `lambda` is the weight of the objective function. Finally, `param` is a Matlab structure that containing a set of optional parameters.

$$\operatorname{prox}_f(x) := \arg\min_{y\in\mathbb{R}^N} \left(\frac{1}{2}\|x-y\|_2^2 + \lambda f(y)\right).$$

The optional parameters for proximal operators are contained into a structure `param`. Every function takes its own inputs. Table 7 presents a list of recurrent optional parameters for proximal operators.

The proximal operators usually return 2 arguments

- `sol` : the minimizer of the problem
- `info` : general information, see table 6.5

7.1 Constraints

If we would like to restrict the set of admissible functions to a subset $\mathscr{C}$ of $\mathbb{R}^L$, i.e. find the optimal solution to (2) considering only solutions in $\mathscr{C}$, we can use projection operator instead of proximal operators. Indeed proximal operators are generalization of projections. For any nonempty, closed and convex set $\mathscr{C} \subset \mathbb{R}^L$, the *indicator function* [**?**] of $\mathscr{C}$ is defined as

$$i_{\mathscr{C}} : \mathbb{R}^L \to \{0, +\infty\} : x \mapsto \begin{cases} 0, & \text{if } x \in \mathscr{C} \\ +\infty & \text{otherwise.} \end{cases}, \tag{6}$$

The corresponding proximity operator is given by the projection onto the set $\mathscr{C}$:

$$\begin{aligned} P_{\mathscr{C}}(y) &= \arg\min_{x\in\mathbb{R}^L} \left\{\frac{1}{2}\|y-x\|_2^2 + i_{\mathscr{C}}(x)\right\} \\ &= \arg\min_{x\in\mathscr{C}} \left\{\|y-x\|_2^2\right\} \end{aligned}$$

Parameter	**Explanation**	**Default value**
`param.tol`	This parameter is used to define the stopping criterion of the problem. The algorithm stops if $$\frac{f(t)-f(t-1)}{f(t)} < tol$$ where $f(t)$ is the function to be minimized at iteration t. $tol \in \mathbb{R}_{+}^{*}$	10^{-2}
`param.abs_tol`	This boolean parameter activates an alternative stopping criterion. The algorithm stops if $$f(t) < tol$$ where $f(t)$ is the function to be minimized at iteration t and $tol \in \mathbb{R}_{+}^{*}$ is given by `param.tol`	0
`param.maxit`	The maximum number of iterations	200
`param.verbose`	Log parameter: 0 no log, 1 a summary at convergence, 2 print main steps	1
`param.gamma`	step-size parameter γ. This constant should satisfy: $\gamma \in [\varepsilon, 2/\beta - \varepsilon]$, for $\varepsilon \in]0, \min\{1, 1/\beta\}[$.	1
`param.A`	forward linear operator, usually used to compute the prox of f(Ax)	Id
`param.At`	adjoint linear operator, usually used to compute the prox of f(Ax)	Id
`param.tight`	Flag for A tight	1
`param.y`	shift, usually used to compute the prox of f(x-y)	0

Table 4: Optional parameter for solvers

Such restrictions are called *constraints* and can be given, e.g. by a set of linear equations that the solution is required to satisfy.

8 Example

The problem Let's suppose we have a noisy image with missing pixels. Our goal would be to find the closest image to the original one. We begin first by setting up some assumptions about the problem.

Assumptions In this particular example, we firstly assume that we know the position of the missing pixels. This can be the result of a previous process on the image or a simple assumption. Secondly, we assume that the image is not special. Thus, it is composed of well delimited patches of colors. Thirdly, we suppose that known pixels are subject to some Gaussian noise with a variance of ε.

Formulation of the problem At this point, the problem can be expressed in a mathematical form. We will simulate the masking operation by a mask A. This first assumption leads to a constraint.

$$Ax = y$$

where x is the vectorized image we want to recover, y are the observe noisy pixels and A a linear operator representing the mask. However due to the addition of noise this constraint is a little bit relaxed. We rewrite it in the following form

$$\|Ax - y\|_2 \leqslant \varepsilon$$

Note that ε can be chosen equal to 0 to satisfy exactly the constraint. In our case, as the measurements are noisy, we set ε to the standard deviation of the noise.

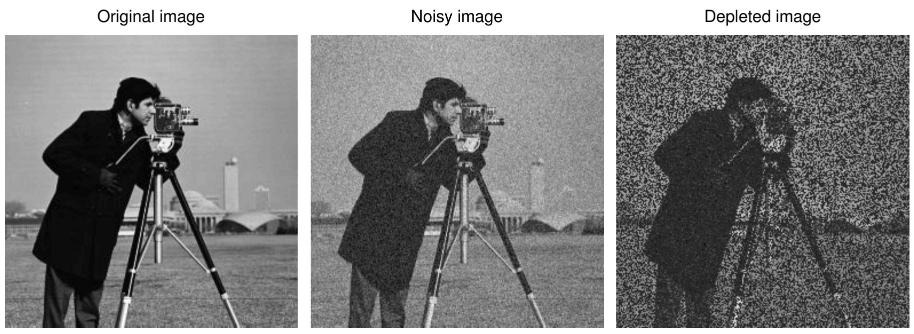

Figure 1: This figure shows the image chosen for this example: the cameraman.

Using the prior assumption that the image has a small TV-norm (image composed of patch of color and few degradee), we will express the problem as

$$\arg\min_x \|x\|_{TV} \qquad \text{subject to} \qquad \|b - Ax\|_2 \leqslant \varepsilon \qquad \text{(Problem I)}$$

where b is the degraded image and A an linear operator representing the mask. ε is a free parameter that tunes the confidence to the measurements. This is not the only way to define the problem. We could also write:

$$arg\min_x \|b - Ax\|_2 + \lambda\|x\|_{TV} \qquad \text{(Problem II)}$$

with the first function playing the role of a data fidelity term and the second a prior assumption on the signal. λ adjusts the tradeoff between measurement fidelity and prior assumption. We call it the *regularization parameter*. The smaller it is, the more we trust the measurements and vice-versa. ε play a similar role as λ. Note that there exist a bijection between the parameters λ and ε leading to the same solution. The bijection function is not trivial to determine. Choosing between one or the other problem will affect the solvers and the convergence rate.

Solving problem I The UNLocBoX solvers take as input functions with their proximity operator or with their gradient. In the toolbox, functions are modelized with structure object with at least two fields. One field contains an operator to evaluate the function and the other allows to compute either the gradient (in case of differentiable function) or the proxitity operator (in case of non differentiable functions). In this example, we need to provide two functions:

- $f_1(x) = \|x\|_{TV}$
  The proximal operator of f_1 is defined as:

  $$prox_{f1,\gamma}(z) = arg\min_x \frac{1}{2}\|x - z\|_2^2 + \gamma\|z\|_{TV}$$

  This function is defined in Matlab using:

  ```
  paramtv.verbose=1;
  paramtv.maxit=50;
  f1.prox=@(x, T) prox_tv(x, T, paramtv);
  f1.eval=@(x) tv_norm(x);
  ```

  This function is a structure with two fields. First, *f1.prox* is an operator taking as input x and T and evaluating the proximity operator of the function (T plays the role of γ is the equation above). Second, and sometime optional, *f1.eval* is also an operator evaluating the function at x.
  The proximal operator of the TV norm is already implemented in the UNLocBoX by the function `prox_tv`. We tune it by setting the maximum number of iterations and a verbosity level. Other parameters are also available (see documentation http://unlocbox.sourceforge.net/doc.php).

  - *paramtv.verbose* selects the display level (0 no log, 1 summary at convergence and 2 display all steps).
  - *paramtv.maxit* defines the maximum number of iteration.

- f_2 is the indicator function of the set S defines by $||Ax - b||_2 < \varepsilon$. We define the proximity operator of f_2 as

$$prox_{f2,\gamma}(z) = arg\min_x \frac{1}{2}\|x - z\|_2^2 + i_S(x),$$

  with $i_S(x)$ is zero if x is in the set S and infinite otherwise. This previous problem has an identical solution as:

$$arg\min_z \|x - z\|_2^2 \qquad \text{subject to} \qquad \|Az - by\|_2 \leqslant \varepsilon$$

  It is simply a projection on the B2-ball. In matlab, we write:

```
    param_proj.epsilon=epsilon;
    param_proj.A=A;
    param_proj.At=A;
    param_proj.y=y;
    f2.prox=@(x,T) proj_b2(x,T,param_proj);
    f2.eval=@(x) eps;
```

  The *prox* field of *f2* is in that case the operator computing the projection. Since we suppose that the constraint is satisfied, the value of the indicator function is 0. For implementation reasons, it is better to set the value of the operator *f2.eval* to *eps* than to 0. Note that this hypothesis could lead to strange evolution of the objective function. Here the parameter *A* and *At* are mandatory. Note that *A* = *At*, since the masking operator can be performed by a diagonal matrix containing 1's for observed pixels and 0's for hidden pixels.

At this point, a solver needs to be selected. The UNLocBoX contains many different solvers. You can try them and observe the convergence speed. Just remember that some solvers are optimized for specific problems. In this example, we present two of them `forward_backward` and `douglas_rachford`. Both of them take as input two functions (they have generalization taking more functions), a starting point and some optional parameters.

In our problem, both functions are not smooth on all points of the domain leading to the impossibility to compute the gradient. In that case, solvers (such as forward backward) using gradient descent cannot be used. As a consequence, we will use Douglas Rachford instead. In matlab, we write:

```
param.verbose=1;
param.maxit=100;
param.tol=10e-5;
param.gamma=1;
sol = douglas_rachford(y,f1,f2,param);
```

- *param.verbose* selects the display level (0 no log, 1 summary at convergence and 2 display all steps).

- *param.maxit* defines the maximum number of iteration.

- *param.tol* is stopping criterion for the loop. The algorithm stops if

$$\frac{n(t) - n(t-1)}{n(t)} < tol,$$

  where $n(t)$ is the objective function at iteration t

- *param.gamma* defines the stepsize. It is a compromise between convergence speed and precision. Note that if *gamma* is too big, the algorithm might not converge.

The solution is displayed in figure 2

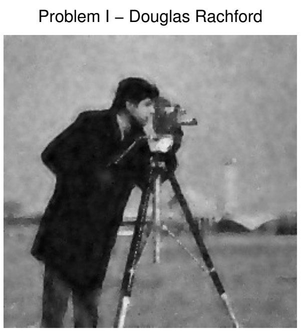

Figure 2: This figure shows the reconstructed image by solving problem I using Douglas Rachford algorithm.

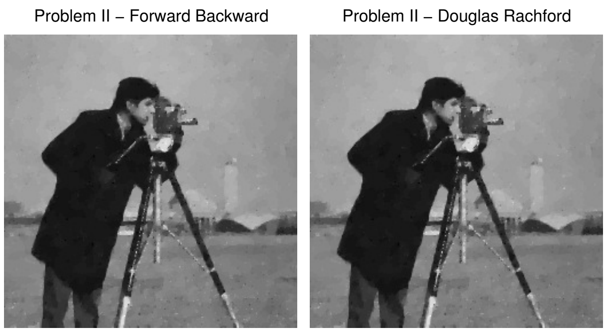

Figure 3: This figure shows the reconstructed image by solving problem II.

Solving problem II Solving problem II instead of problem I can be done with a small modification of the previous code. First we define another function as follow:

```
param_l2.A=A;
param_l2.At=A;
param_l2.y=y;
param_l2.verbose=1;
f3.prox=@(x,T) prox_l2(x,lambda*T,param_l2);
f3.grad=@(x) 2*lambda*A(A(x)-y);
f3.eval=@(x) lambda*norm(A(x)-y,'fro');
```

The structure of *f3* contains a field *f3.grad*. In fact, the l2 norm is a smooth function. As a consequence the gradient is well defined on the entire domain. This allows using the forward backward solver. However, we can in this case also use the Douglas Rachford solver. For this we have defined the field *f3.prox*.

We remind that forward backward will not use the field *f3.prox* and Douglas Rachford will not use the field *f3.grad*. The solvers can be called by:

```
sol21 = forward_backward(y,f1,f3,param);
```

Or:

```
sol22 = douglas_rachford(y,f3,f1,param);
```

These two solvers will converge (up to numerical errors) to the same solution. However, convergence speed might be different. As we perform only 100 iterations with both of them, we do not obtain exactly the same result. Results is shown in figure 3.

Remark: The parameter *lambda* (the regularization parameter) and *epsilon* (The radius of the l2 ball) can be chosen empirically. Some methods allow to compute those parameters. However, this is far beyond the scope of this tutorial.

Appendix: example's code

```
%% Initialisation
clear all;
close all;

% Loading toolbox
global GLOBAL_useGPU;
init_unlocbox();

verbose=2; % verbosity level

%% Load an image

% Original image
im_original=cameraman;

% Displaying original image
imagescgray(im_original,1,'Original image');

%% Creation of the problem

sigma_noise = 20/255;
im_noisy=im_original+sigma_noise*randn(size(im_original));

% Create a matrix with randomly 50 % of zeros entry
p=0.5;
matA=rand(size(im_original));
matA=(matA>(1-p));
% Define the operator
A=@(x) matA.*x;

% Depleted image
y=matA.*im_noisy;

% Displaying noisy image
imagescgray(im_noisy,2,'Noisy image');

% Displaying depleted image
imagescgray(y,3,'Depleted image');

%% Setting the proximity operator

% setting the function f1 (norm TV)
paramtv.useGPU = GLOBAL_useGPU;   % Use GPU
paramtv.verbose = verbose-1;
paramtv.maxit = 50;
f1.prox=@(x, T) prox_tv(x, T, paramtv);
f1.eval=@(x) tv_norm(x);

% setting the function f2
param_proj.epsilon = sqrt(sigma_noise^2*length(im_original(:))*p);
param_proj.A = A;
param_proj.At = A;
param_proj.y = y;
param_proj.verbose = verbose-1;
f2.prox=@(x,T) proj_b2(x,T,param_proj);
f2.eval=@(x) eps;

% setting the function f3
lambda = 10;
param_l2.A = A;
param_l2.At = A;
param_l2.y = y;
param_l2.verbose = verbose-1;
param_l2.tight = 0;
param_l2.nu = 1;
f3.prox=@(x,T) prox_l2(x,lambda*T,param_l2);
```

```
f3.grad=@(x) 2*lambda*A(A(x)-y);
f3.eval=@(x) lambda*norm(A(x)-y,'fro')^2;

%% Solving problem I

% setting different parameters for the simulation
param.verbose = verbose;    % display parameter
param.maxit = 100;    % maximum number of iterations
param.tol = 1e-5;    % tolerance to stop iterating
param.gamma = 1 ;     % Convergence parameter
% solving the problem with Douglas Rachord
sol = douglas_rachford(y,f1,f2,param);

%% Displaying the result
imagescgray(sol,4,'Problem I - Douglas Rachford');

%% Solving problem II (forward backward)
param.gamma=0.5/lambda;     % Convergence parameter
param.tol=1e-5;
% solving the problem with Douglas Rachord
sol21 = forward_backward(y,f1,f3,param);

%% Displaying the result
imagescgray(sol21,5,'Problem II - Forward Backward');

%% Solving problem II (Douglas Rachford)
param.gamma=0.5/lambda;     % Convergence parameter
sol22 = douglas_rachford(y,f3,f1,param);

 %% Displaying the result
imagescgray(sol22,6,'Problem II - Douglas Rachford');

%% Close the UNLcoBoX
close_unlocbox();
```

References