UNLOCBOX: A SIMPLE TUTORIAL

MATLAB CONVEX OPTIMIZATION TOOLBOX

Lausanne - November 2014

PERRAUDIN Nathanaël – LTS2 - EPFL

1 Introduction

Welcome to the tutorial of the UNLocBoX. In this document, we provide an example application that uses the basic concepts of the toolbox. Here you will also find some tricks that may be very useful. You can find an introduction and more detailed documentation in the userguide, available at http://unlocbox.sourceforge.net/notes/unlocbox-note-002.pdf

This toolbox is designed to solve convex optimization problems of the form:

$$arg \min_{x \in \mathbb{R}^N} \left(f_1(x) + f_2(x) \right),$$

or more generally

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

where the f_i are lower semi-continuous convex functions and x the optimization variables. For more details about the problems, please refer to the userguide (UNLocBoX-note-002) available on https://lts2.epfl.ch/unlocbox/notes/unlocbox-note-002.pdf .

This toolbox is based on proximal splitting methods. Those methods cut the problem into smaller (and easier) subproblems that can be solved in an iterative fashion. The UNLocBoX essentially consists of three families of functions:

- Proximity operators: they solve small minimization problems and allow a quick implementation of many composite problems.
- Solvers: generic minimization algorithms that can work with different combinations of proximity operators in order to minimize complex objective functions
- Demonstration files: examples to help you to use the toolbox

This toolbox is provided for free. We would be happy to receive comments, information about bugs or any other kind of help in order to improve the toolbox.

2 A simple example: Image in-painting

Let's suppose we have a noisy image with missing pixels. Our goal is simply to fill the unknown values in order to reconstruct an image close to the original one. We first begin by setting up some assumptions about the problem.

3 Assumptions

In this particular example, we firstly assume that we know the position of the missing pixels. This happens when we know that a specific part of a photo is destroyed, or when we have sampled some of the pixels in known positions and we wish to recover the rest of the image. Secondly, we assume that the image follows

Figure 1: The original image provided by the toolbox. Use cameraman() function to access.

some standard distribution. For example, many natural images are known to have sharp edges and almost flat regions (the extreme case would be the cartoon images with completely flat regions). Thirdly, we suppose that known pixels are subject to some Gaussian noise with a variance of ε.

Figure 2: Noisy image.

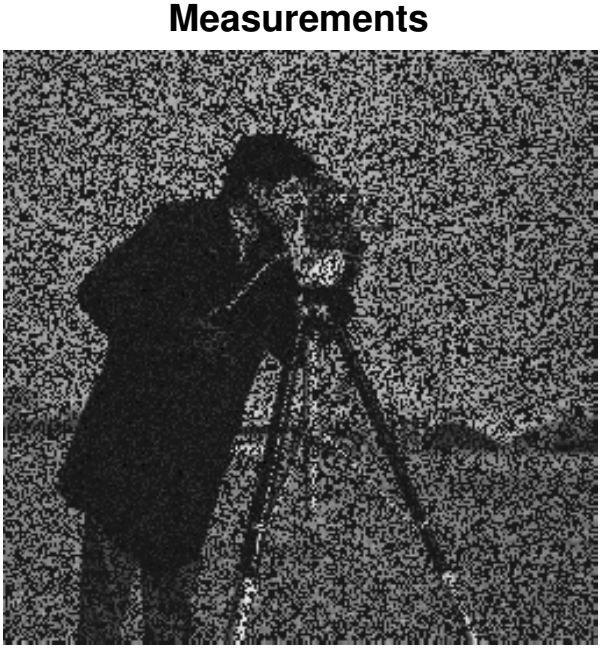

Figure 3: Measurements. 50 percent of the pixels have been removed.

4 Formulation of the problem

At this point, the problem can be expressed in a mathematical form. We will simulate the masking operation with an operator A. This first assumption leads to a constraint.

$$Ax = y$$

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

$$\|Ax - y\|_2 \leqslant \sqrt{N}\varepsilon$$

where N is the number of known pixels. Note that ε can be chosen to be equal to 0 so that the equality $y = Ax$ is satisfied. In our case, as the measurements are noisy, we set ε to be the expected value of the norm of the noise (standard deviation times square root of number of measurements).

We use as a prior assumption that the image has a small total variation norm (TV-norm). (The TV-norm is the l^1-norm of the gradient of x.) On images, this norm is low when the image is composed of patches of color and few "degradee" (gradients). This is the case for most of natural images. To summarize, we express the problem as

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

Note that if the amount of noise is not known, epsilon as a free parameter that tunes the confidence to the measurements. However, this is not the only way to define the problem. We could also write:

$$arg\min_x \|Ax - y\|_2^2 + \lambda \|x\|_{TV} \qquad \text{(Problem II)}$$

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 conversely. ε plays a similar role as λ.

We have presented two ways to formulate the problem. The reader should keep in mind that choosing between one or the other problem will affect the choice of the solver and the convergence rate. With experience, one should be able to know in advance which problem will lead to the best solver.

Note that there exists a bijection between the parameters λ and ε leading both problems to the same solution. Unfortunately, the bijection function is not trivial to determine.

Once your problem is well defined, we need to provide a list of functions to the UNLocBoX solver. (For example, in Problem 2, the functions are $\|Ax-y\|_2^2$ and :math:‘ lambda |x|_{TV}‘.) Every function is modeled by a MATLAB structure containing some special fields. We separate the functions in two different types: differentiable and non differentiable. For differentiable function, the user needs to fill the following fields: * *func.eval* : An anonymous function that evaluate the function * *func.grad* : An anonymous function that evaluate the gradient * *func.beta* : An upper bound on the Lipschitz constant of the gradient

For instance, the function $\|Ax-y\|_2^2$ is defined in MATLAB by:

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

The Lipschitz constant of a the gradient is defined as:

$$\min_{\beta} \text{ s.t } \forall x_1, x_2 \in \mathbb{R}^N \text{ we have } \|\nabla_f(x_1) - \nabla_f(x_2)\|_2 \leqslant \beta \|x_1 - x_2\|_2$$

When the function is not differentiable, the field *.beta* is dropped and *.grad* is replaced by the field *.prox* that contains an anonymous function for the proximity operator (They will be explained in more details the following section.

ftv.prox = @(x, T) prox_tv(x, T * lambda, paramtv); ftv.eval = @(x) lambda * tv_norm(x);

5 Proximity operators

The proximity operator of a lower semi-continuous convex function f is defined by:

$$prox_{\lambda f}(z) = arg\min_x \frac{1}{2}\|x-z\|_2^2 + \lambda f(x)$$

Proximity operators minimize a function without going too far from a initial point. They can be thought or assimilated as de-noising operators. Because of the l2-term in the minimization problem, proximity operators perform a regularized minimization of the function f. However, applied iteratively, they lead to the minimization of this function. For x^* the minimizer of the function f, it is obvious that:

$$x^* = prox_f(x^*) = arg\min_x \frac{1}{2}\|x-x^*\|_2^2 + f(x)$$

In a sense, proximity operators perform a regularized minimization of the function f. However, they also provide a framework to handle constraints. Those can be inserted into the problem thanks to indicative functions. These functions assert if x belong to a set C. They only have two output values: 0 if x is in the set and ∞ otherwise:

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

The solution of the proximity operator of this function has to be in the set C, otherwise the $i_C(x) = \infty$. Moreover, since it also minimizes $\|x-z\|_2^2$, it will select the closest point to z. As a result the proximity operators of indicator functions are projections.

It is important to keep in mind the equivalence between constraints and indicative functions. This is the trick that allows to use hard constraint with the UNLocBoX as it cannot directly handle them. The constraints will thus be inserted in the form of indicative functions.

6 Solving problem I

The UNLocBoX is based on proximal splitting techniques for solving convex optimization problems. These techniques divide the problem into smaller problems that are easier to solve. Topically, each function will compose a sub-problem that will be solved by its proximity operator (or gradient step). In the particular case of problem (I), the solver will iteratively, first minimize a little bit the TV norm and second perform the projection on the fidelity term B2-ball. (The B2-ball is the space of point x satisfying $\|Ax-y\| \leqslant \sqrt{N}\varepsilon$). To solve problem (I), we minimize two functions:

- The TV norm: $f_1(x) = \lambda ||x||_{TV}$ The proximity operator of f_1 is given by:

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

  In MATLAB, the function is defined by the following code:

```
paramtv.verbose = 1;
paramtv.maxit = 50;
f1.prox = @(x, T) prox_tv(x, T * lambda, paramtv);
f1.eval = @(x) lambda * 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 has be stay a free weight for the solver. it is going to be replaced by the timestep later). Second, *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).

  - *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 for this proximity operator.

  Not that for problem (I), lambda can be dropped or set to 1. This parameter will be used when solving problem (II).

- f_2 is the indicator function of the set S defined by $||Ax-y||_2 < \varepsilon$ The proximity operator of f_2 is:

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

  with $i_S(x)$ is zero if x is in the set S and infinite otherwise. Under some technical assumption, this previous problem has an identical solution as:

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

  It is simply a projection on the B2-ball (The B2-ball is the set of all points satisfying $\|Ax-y\|_2 < \varepsilon$). 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. Please notice here the two following lines:

```
param_proj.A = A;
param_proj.At = A;
```

In fact we consider here the masking operator *A* as a diagonal matrix containing 1's for observed pixels and 0's for hidden pixels. As a consequence: $A = At$. In MATLAB, one easy way to implement this operator is to use:

```
A = @(x) matA .* x;
```

with *matA* the mask. In a compressed sensing problem for instance, you would define:

```
param_proj.A = @(x) Phi * x;
param_proj.At = @(x) Phi' * x;
```

where *Phi* is the sensing matrix!

At this point, we are ready to solve the problem. The UNLocBoX contains many different solvers and also a universal one that will select a suitable method for the problem. To use it, just write:

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

You can also use a specific solver for your problem. In this tutorial, 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 = 2;
param.maxit = 50;
param.tol = 10e-5;
param.gamma = 0.1;
fig = figure(100);
param.do_sol=@(x) plot_image(x,fig);
sol = douglas_rachford(y,f1,f2,param);
```

Or in an equivalent manner (this second way is recommended):

```
param.method = "douglas_rachford"
sol = solvep(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 step-size. It is a compromise between convergence speed and precision. Note that if *gamma* is too big, the algorithm might not converge. By default, this parameter is computed automatically.
- Finally, the following line allows to display the current reconstruction of the image at each iteration:

```
param.do_sol=@(x) plot_image(x,fig);
```

Figure 4: This figure shows the reconstructed image by solving problem I using Douglas Rachford algorithm.

You can stop the simulation by typing "ctrl + d" in the consol. At the end of the next iteration, the algorithm will stop and return the current solution.

7 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:

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

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 that can be called by:

```
param.method = "forward_backward"
sol21 = solvep(y,{f1,f2},param);
```

In this case, we can also use the `douglas_rachford` solver. To do so, we need to define the field *f3.prox*. In general, this is not recommended because a gradient step is usually less computationally expensive than a proximal operator:

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

param.method = "douglas_rachford"
sol22 = solvep(y, {f1,f3}, param);
```

We remind the user that `forward_backward` will not use the field *f3.prox* and `douglas_rachford` will not use the field *f3.grad*.

These two solvers will converge (up to numerical error) 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.

Figure 5: This figure shows the reconstructed image by solving problem II using the Forward Backward algorithm.

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.

Figure 6: This figure shows the reconstructed image by solving problem II using the Douglas Rachford algorithm.

8 Conclusion

In this tutorial, the reader can observe that problem (II) is solved much more efficiently than problem (I).However, writing the problem with a constraint (like problem (I)) often allow a much easier tuning of the parameters at the cost of using a slower solver.

Only experience helps to know which formulation of a problem will lead to the best solver. Usually, forward backward (FISTA) and ADMM are considered to be the best solvers.

Speed consideration are relative when using the UNLocBoX. Due to general implementation of the toolbox, we estimate the overall speed between one and two times slower than an optimal algorithm cooked and optimized for a special problem (in MATLAB).

Thanks for reading this tutorial

References: [**?**], [**?**]

References