FOM – A MATLAB Toolbox of First Order Methods for Solving Convex Optimization Problems

Amir Beck * Nili Guttmann-Beck †

August 30, 2017

Abstract

This paper presents the FOM MATLAB toolbox for solving convex optimization problems using first order methods. The diverse features of the eight solvers included in the package are illustrated through a collection of examples of different nature.

1 Introduction

This paper describes the FOM MATLAB toolbox (FOM standing for "first order methods") comprising right first order methods for solving several convex programming models. The purpose of the package is to provide researchers and practitioners a set of methods that are able to solve a variety of convex optimization problems using only computations of, potentially, (sub)gradients of the involved functions as well as their conjugates, proximal mappings and employment of linear transformations and their adjoints.

There exist several excellent solvers such as SeDuMi [16] and SDPT3 [17] which can be applied to solve *conic* convex optimization problems. CVX [12] is a MATLAB toolbox which acts as an interface to these two solvers as well as other methods such as TFOCS [7] that solves conic convex problems using a class of first order algorithms.

FOM is not restricted to conic convex problems, and is able to tackle any convex problem for which the corresponding required oracles are available. Some of the solvers in FOM can also be employed on nonconvex problems, see more details in Section 3.5.
The software can be downloaded from the website

`https://sites.google.com/site/fomsolver/home`

The paper is organized as follows. Section 2 gives an overview of the different optimization models and methods that are tackled by FOM along with the corresponding assumptions and oracles. The solvers are partitioned into three groups that define a certain hierarchy between them. Section 3 offers a variety of examples that demonstrate the strengths and capabilities of the different FOM functions, and provides the user some insights regarding issues such as model, method and parameter choices.

*Faculty of Industrial Engineering and Management, Technion - Israel Institute of Technology, Haifa, Israel. Email: becka@ie.technion.ac.il

†School of Computer Science, Academic College of Tel-Aviv Yaffo, Yaffo, Israel. Email: becknili@mta.ac.il

2 Models, Methods and Underlying Assumptions

The eight solvers that comprise the package are listed in Table 1 along with the relevant models they tackle and the required assumptions. Sections 2.1 and 2.2 explain and elaborate on the notation used in the table.

MATLAB function	Method	Minimization Model	Assumptions	Oracles
prox_subgradient	proximal subgradient	$f(\mathbf{x}) + \lambda g(\mathbf{x})$	$f - \text{lip}$ $g - \text{pc}, \text{prx}$ $\lambda > 0$	f, f' $g, \text{prox}_{\alpha g}$
comd	co-mirror descent	$\min \quad f(\mathbf{x})$ $\text{s.t.} \quad g_i(\mathbf{x}) \leq 0,$ $\mathbf{x} \in X$	$f - \text{lip}$ $g_i - \text{lip}$ X−simple	f, f' g_i, g_i'
prox_gradient	proximal gradient	$f(\mathbf{x}) + \lambda g(\mathbf{x})$	$f - \text{sm}$ $g - \text{pc}, \text{prx}$ $\lambda > 0$	$f, \nabla f$ $g, \text{prox}_{\alpha g}$
sfista	smoothed FISTA	$f(\mathbf{x}) + \lambda_g g(\mathcal{A}\mathbf{x}) + \lambda_h h(\mathbf{x})$	$f - \text{sm}$ $g - \text{pc}, \text{prx}$ $h - \text{pc}, \text{prx}$ $\lambda_g, \lambda_h > 0$	$f, \nabla f$ $g, \text{prox}_{\alpha g}$ $h, \text{prox}_{\alpha h}$ $\mathcal{A}, \mathcal{A}^T$
adlpmm	alternating direction linearized proximal method of multipliers	$f(\mathbf{x}) + \lambda g(\mathcal{A}\mathbf{x})$	$f - \text{pc}, \text{prx}$ $g - \text{pc}, \text{prx}$ $\lambda > 0$	$f, \text{prox}_{\alpha f}$ $g, \text{prox}_{\alpha g}$ $\mathcal{A}, \mathcal{A}^T$
nested_fista	Nested FISTA	$\varphi(\mathbf{f}(\mathbf{x})) + \lambda g(\mathcal{A}\mathbf{x})$	$\varphi - \text{lip}, \text{nd}$ prx $\mathbf{f} - \text{sm}$ $g - \text{pc}, \text{prx}$ $\lambda > 0$	$\varphi, \text{prox}_{\alpha\varphi}$ $\mathbf{f}, \nabla\mathbf{f}$ $g, \text{prox}_{\alpha g}$ $\mathcal{A}, \mathcal{A}^T$
fista	FISTA	$f(\mathbf{x}) + \lambda g(\mathbf{x})$	$f - \text{sm}$ $g - \text{pc}, \text{prx}$ $\lambda > 0$	$f, \nabla f$ $g, \text{prox}_{\alpha g}$
fdpg	fast dual proximal gradient	$f(\mathbf{x}) + \lambda g(\mathcal{A}\mathbf{x})$	$f - \text{pc}, \text{sc}$ $g - \text{pc}, \text{prx}$ $\lambda > 0$	$f, \nabla f^*$ $g, \text{prox}_{\alpha g}$ $\mathcal{A}, \mathcal{A}^T$

Table 1: Models and assumptions of the eight solvers.

2.1 Assumptions

An underlying assumption that is not written in the table is that **all the involved functions are convex**. In some special cases, nonconvex problems can also be treated, see Section 3.5. In addition, the following abbreviations are used to denote properties of functions:

lip "Lipschitz". A function f is is Lipschitz if it Lipschitz continuous over the entire space. That is, there exists an $\ell > 0$ such that

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq \ell |\mathbf{x} - \mathbf{y}| \text{ for all } \mathbf{x}, \mathbf{y}.$$

pc "proper and closed".

prx "proximable". A function f is "proximable" if for any positive α, the prox operator of αf given by

$$\text{prox}_{\alpha f}(\mathbf{x}) = \underset{\mathbf{u}}{\text{argmin}} \left\{ \alpha f(\mathbf{u}) + \frac{1}{2}\|\mathbf{u} - \mathbf{x}\|^2 \right\},$$

can be computed efficiently.

sm "smooth". In our context, a function f is considered to be "smooth" if it is differentiable over the entire space and there exists $L > 0$ such that

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\| \text{ for all } \mathbf{x}, \mathbf{y}.$$

sc "strongly convex". A function f is strongly convex if there exists $\sigma > 0$ such that $f(\mathbf{x}) - \frac{\sigma}{2}\|\mathbf{x}\|^2$ is convex.[1]

nd "non-decreasing". A function $\varphi : \mathbb{R}^m \to \mathbb{R}$ is called *non-decreasing* if

$$\varphi(\mathbf{x}) \leq \varphi(\mathbf{y}) \text{ for any } \mathbf{x}, \mathbf{y} \text{ satisfying } \mathbf{x} \leq \mathbf{y}.$$

In addition, the comirror descent method requires the underlying set X to be "simple", which here means that it is one of the following four options:

'simplex'	$X = \{\mathbf{x} \in \mathbb{R}^n : \sum_{i=1}^n x_i = r, \ell_i \leq x_i \leq u_i, i = 1, \ldots, n\}$
'ball'	$X = \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x} - \mathbf{c}\|_2 \leq r\}$
'box'	$X = \{\mathbf{x} \in \mathbb{R}^n : \ell_i \leq x_i \leq u_i, i = 1, \ldots, n\}$
'spectahedron'	$X = \{\mathbf{X} \in \mathbb{S}^n : \mathbf{0} \preceq \mathbf{X} \preceq u\mathbf{I}, \text{Tr}(\mathbf{X}) = r\}$

2.2 Oracles

The solvers require various oracles as inputs, where each oracle is a MATLAB function handle. For a function f and a linear transformation $\mathcal{A}$, the following oracle notations are used:

f - function value of f ($\mathbf{x} \mapsto f(\mathbf{x})$).

f' - a (specific) subgradient of f ($\mathbf{x} \mapsto f'(\mathbf{x}) \in \partial f(\mathbf{x})$).

∇f - gradient of f ($\mathbf{x} \mapsto \nabla f(\mathbf{x})$).

∇f^* - gradient of the conjugate of f ($\mathbf{x} \mapsto \text{argmin}\{\langle \mathbf{u}, \mathbf{x} \rangle - f(\mathbf{u})\}$)

$\text{prox}_{\alpha f}$ - proximal operator of a positive constant times the function ($(\mathbf{x}, \alpha) \mapsto \text{prox}_{\alpha f}(\mathbf{x})$).

$\mathcal{A}$ - linear transformation $\mathcal{A}$ ($\mathbf{x} \mapsto \mathcal{A}\mathbf{x}$).

$\mathcal{A}$ - adjoint of $\mathcal{A}$ ($\mathbf{x} \mapsto \mathcal{A}^T\mathbf{x}$).

[1] This definition of strong convexity is valid since the underlying space is assumed to be Euclidean.

The proximal (or "prox") operator [14] can be provided by the user, but the package also contains a large amount of implementations of proximal operators; see Table 2. Also, if $f = \delta_C$ ($\delta_C(\mathbf{x}) = 0$ for $\mathbf{x} \in C$ and ∞ for $\mathbf{x} \notin C$) with a nonempty closed and convex C, then $\text{prox}_{\alpha f} = P_C$ which is the orthogonal projection operator on C. The orthogonal projections that are implemented in the package are described in Table 3.

2.3 Solvers Overview

The eight solvers can be divided into three group.

- **Group 1.** prox_subgradient, comd.
- **Group 2.** prox_gradient, sfista, adlpmm.
- **Group 3.** fista, nested_fista, fdpg.

The above partition is made according to the known iteration complexity results of the methods, where group 1 consists of the slowest methods and group 3 consists of the fastest methods. Thus, if several methods can solve a certain problem, it is better to choose a method with the highest possible group number.

The first group consists of two nonsmooth solvers: the proximal subgradient [10] and the co-mirror descent [2] methods. Both methods share a complexity of $O(1/\varepsilon^2)$, meaning that the number of iteration required to obtain an ε-optimal (or ε-optimal and feasible) solution is of an order of $1/\varepsilon^2$.

The second group consists of three solvers: proximal gradient [3, 4, 8], smoothed FISTA [5] and the alternating direction linearized proximal method of multipliers (ADLPMM) [11, 13, 15]. These three methods all share an $O(1/\varepsilon)$ complexity, and they are therefore considered to be faster than the methods from the first group, albeit slower than the third group's solvers.

The third group comprises three solvers: FISTA [3, 4], nested FISTA, which is a generalization of FISTA to a non-additive composite model and the fast dual proximal gradient method (FDPG) from [6]–all have an $O(1/\sqrt{\varepsilon})$ complexity in terms of function values of the primal or dual problems.

Most of the methods used in FOM are also described and analyzed in the book [1].

3 A Tour of FOM

We will not go over the syntax of each the eight solvers since this will be an extremely tedious and unnecessary task. Detailed explanations on the input and output arguments can be found in the FOM's website `https://sites.google.com/site/fomsolver/home`. In addition, all the functions are equipped with detailed help notes. For example,

```
>> help prox_subgradient
 prox_subgradient employs the proximal subgradient method
 for solving the problem min{f(x) + lambda* g(x)}
```

```
Underlying assumptions:
All functions are convex
f is Lipschitz
g is proper closed and proximable
lambda is a positive scalar
=====================================================
Usage:
out               = prox_subgradient(Ffun,Ffun_sgrad,Gfun,Gfun_prox,lambda,startx,[par])
[out,fmin]        = prox_subgradient(Ffun,Ffun_sgrad,Gfun,Gfun_prox,lambda,startx,[par])
[out,fmin,parout] = prox_subgradient(Ffun,Ffun_sgrad,Gfun,Gfun_prox,lambda,startx,[par])
=====================================================
Input:
Ffun        - function handle for the function f
Ffun_sgrad  - function handle for the subgradient of the function f
Gfun        - function handle for the function g
Gfun_prox   - function handle for the proximal mapping of g times a postive constant
lambda      - positive scalar penalty for the function g
startx      - starting vector
par         - struct which contains different values required
              for the operation of prox_subgradient
Fields of par:
      max_iter   - maximal number of iterations [default: 1000]
      eco_flag   - true if economic version (without calculating objective function
                           values) should run, otherwise false [default: false]
      print_flag - true if internal printing should take place, otherwise false [default: true]
      alpha      - positive constant determining the stepsize of the method
                   (which is alpha/sqrt(iternu+1) [default: 1]
      eps        - stopping criteria tolerance (the method stops when the
                   norm of the difference between consecutive iterates is < eps) [default: 1e-5]
=====================================================
Output:
out       - optimal solution (up to a tolerance)
fmin      - optimal value (up to a tolerance)
parout    - a struct containing additional information related to the convergence.
            The fields of parout are:
     iterNum   - number of performed iterations
     funValVec - vector of all function values generated by the method
```

We will however embark on a tour comprising several examples that will cover most of the features and capabilities of the package, demonstrating also important issues such as model and method choices, as well as questions regarding the input and output parameters.

3.1 Choosing the Model

Consider the problem

$$\min_{\mathbf{x}\in\mathbb{R}^4}\{\|\mathbf{A}\mathbf{x}-\mathbf{b}\|_1+2\|\mathbf{x}\|_1\}, \tag{3.1}$$

where $\mathbf{A}$ and $\mathbf{b}$ are generated by the commands

```
>> A = [0.6324    0.9575    0.9572    0.4218;
     0.0975    0.9649    0.4854    0.9157;
```

```
     0.2785    0.1576    0.8003    0.7922;
     0.5469    0.9706    0.1419    0.9595];
>> b = [0.6843;   0.6706;    0.4328;    0.8038];
```

The proximal subgradient method (implemented in the MATLAB function prox_subgradient) solves problems of the form (see Table 1)

$$\min f(\mathbf{x}) + \lambda g(\mathbf{x}). \tag{3.2}$$

Obviously, problem (3.1) fits model (3.2) with

$$f(\mathbf{x}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_1, g(\mathbf{x}) = \|\mathbf{x}\|_1, \lambda = 2. \tag{3.3}$$

Note that all the assumptions that f and g need to satisfy according to Table 1 (f - nonsmooth convex, g proper closed convex and proximable) are met. The syntax of prox_subgradient is

```
[out,fmin,parout] = prox_subgradient(Ffun,Ffun_sgrad,Gfun,Gfun_prox,lambda,startx,[par])
```

In the specific case of f and g chosen by (3.3), the input is as follows:

- `Ffun` is a function handle for f [`@(x)norm(A*x-b,1)`].
- `Ffun_sgrad` is a function handle for a subgradient of f [`@(x)A'*sign(A*x-b)`].
- `Gfun` is a function handle for g [`@(x)norm(x,1)`].
- `Gfun_prox` is a function handle for the prox of g times a constant [`@(x,a)prox_l1(x,a)`].

  In this case, the proximal mapping is one of the prox functions implemented in the package (see Table 2 for a complete list). In cases where the prox is not one of the implemented functions, the user can provide its own implementation of the prox.
- `lambda` is equal to 2.
- `startx` is an initial vector, and we will choose it in this example as the zeros vector.

Running the solver yields the following output

```
>> [out,fmin,parout] =prox_subgradient(@(x)norm(A*x-b,1),@(x)A'*sign(A*x-b),...
     @(x)norm(x,1),@(x,a)prox_l1(x,a),2,zeros(4,1));
*********************
prox_subgradient
*********************
#iter        fun. val.
     6          2.526541
     8          2.021784
    10          1.869343
    42          1.858085
     :              :
   828          1.821805
   901          1.820594
   974          1.820261
---------------------------------
Optimal value =        1.820261
```

The proximal subgradient method is not a descent method and only iterations in which an improved (that is, lower) value was obtained are printed. The array `parout.funValVec` contains all the function values obtained during the execution of the method.

```
>> parout.funValVec

    2.5915
    6.2732
    2.5915
       :
    1.8650
    1.8485
    1.8474
```

A plot of the function values can be generated by the command

```
plot(parout.funValVec)
```

The resulting graph (Figure 1) demonstrates that proximal subgradient is indeed not a descent method.

The best achieved function value is stored in `fmin`

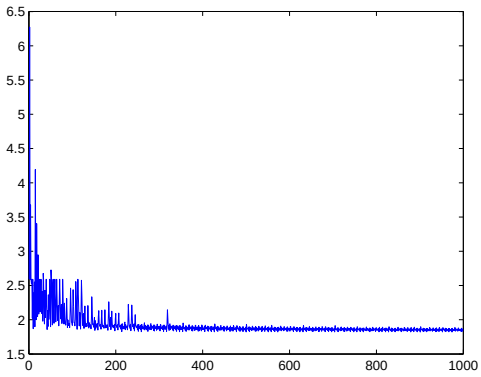


Figure 1: Function values generated by the proximal subgradient method.

```
>> fmin

fmin =

    1.8203
```

The choice of f and g in (3.3) is only one option. Another rather straightforward choice is to set

$$f(\mathbf{x}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_1 + 2\|\mathbf{x}\|_1, g(\mathbf{x}) \equiv 0.$$

The parameter λ can be chosen as any positive number; we will arbitrary set it to be one. We can run the proximal subgradient method for the above choice of f and g (recalling that $\text{prox}_{\alpha g}(\mathbf{x}) = \mathbf{x}$ for all $\alpha > 0$ whenever $g \equiv 0$).

```
>> [out,fmin,parout] =prox_subgradient(@(x)norm(A*x-b,1)+2*norm(x,1),...
@(x)A'*sign(A*x-b)+2*sign(x),@(x)0,@(x,a)x,1,zeros(4,1));
*********************
prox_subgradient
*********************
#iter        fun. val.
    73          2.587218
    74          2.280796
    75          1.976423
   200          1.965525
   201          1.937473
   212          1.893150
   214          1.865244
   297          1.856431
   510          1.844707
   643          1.842849
----------------------------------
Optimal value =        1.842849
```

Note that the obtained function value (1.8428) is higher than the one obtained in the previous run (1.8203). This is not surprising since the theoretical results of the proximal subgradient method shows that the rate of convergence in function values depends on the Lipschitz constant of the function f and not of g (see [10]). In the second run, the Lipschitz constant of f is larger, and thus the empirical results validate to some extent the known convergence results. Loosely speaking, it is better to put as much as possible from the objective function into g.

3.2 Choosing the Solver

It is actually possible to solve problem (3.1) using the solver `adlpmm` from group 2, which should exhibit better performance than `prox_subgradient` from group 1. The solver is a MATLAB implementation of the "alternating direction linearized proximal method of multipliers" algorithm. As can be seen in Table 1, the minimization model that adlpmm tackles is

$$\min_{\mathbf{x}} f(\mathbf{x}) + \lambda g(\mathcal{A}\mathbf{x}), \tag{3.4}$$

where both f and g are proper closed, convex, and in addition proximable. Problem (3.1) fits model (3.4) with

$$f(\mathbf{x}) = 2\|\mathbf{x}\|_1, g(\mathbf{y}) = \|\mathbf{y} - \mathbf{b}\|_1, \lambda = 1, \mathcal{A}(\mathbf{x}) = \mathbf{A}\mathbf{x}.$$

Note that both f and g are proximable. Indeed, denoting the prox of α times the l_1-norm function by $\mathcal{T}_\alpha$ (a.k.a. the "soft thresholding operator" [9]):

$$\mathcal{T}_\alpha(\mathbf{x}) \equiv [|\mathbf{x}| - \alpha\mathbf{e}]_+ \odot \operatorname{sgn}(\mathbf{x}),$$

the proximal mappings of αf and αg (for $\alpha > 0$) can be written explicitly as

$$\operatorname{prox}_{\alpha f}(\mathbf{x}) = \mathcal{T}_{2\alpha}(\mathbf{x}), \operatorname{prox}_{\alpha g}(\mathbf{x}) = \mathcal{T}_\alpha(\mathbf{x} - \mathbf{b}) + \mathbf{b}.$$

The syntax of `adlpmm` is

```
[out,fmin,parout] = adlpmm(Ffun,Ffun_prox,Gfun,Gfun_prox,Afun,Atfun,lambda,startx,[L],[par])
```

Note that an optional input parameter is the positive constant L. In fact, L should be chosen as an upper bound on $\|\mathcal{A}\|^2$. In case where $\mathcal{A}\mathbf{x} \equiv \mathbf{A}\mathbf{x}$ for some matrix $\mathbf{A}$, $\|\mathcal{A}\| = \|\mathbf{A}\|_2$ is the spectral norm of $\mathbf{A}$. It is highly recommended that user will insert a value of L, since otherwise the solver will find $\|\mathcal{A}\|$ by an inefficient method. It is also important to realize that in general, the function g in the model (3.4) can be extended real-valued and `adlpmm` is not guaranteed to generate a vector in $\mathrm{dom}(g)$. This is the reason why by default the method also computes the feasibility violation of the generated sequence of vectors. It is highly advisable, in cases where g is real-valued, as is the case in this example, to "notify" the solver that g is real-valued by setting `par.real_valued_flag` to `true`.

```
>> clear par;
>> par.real_valued_flag=true;
>> [out,fmin,parout] =adlpmm(@(x)2*norm(x,1),@(x,a)prox_l1(x,2*a),@(x)norm(x-b,1),...
   @(x,a)prox_l1(x-b,a)+b,@(x)A*x,@(x)A'*x,1,zeros(4,1),norm(A)^2,par);
*********************
adlpmm
*********************
#iter        fun. val.
     2          1.873908
    11          1.870255
    35          1.818719
     :               :
   575          1.814974
   674          1.814974
   793          1.814974
Stopping because the norm of the difference between consecutive iterates is too small
----------------------------------
Optimal value =        1.814974
```

Note that already in iteration 35 the method obtained a better function value than the one obtained by the proximal subgradient method after almost 1000 iterations. This is not a surprising outcome since, as was already noted, `adlpmm` belongs to the second group of solvers whereas `prox_subgradient` belongs to the first group.

3.3 Choice of Solver in the Same Group

Consider the problem

$$\min\left\{\max_{i=1,2,\ldots,80}\{\mathbf{a}_i^T\mathbf{x}\} : \mathbf{x} \in \Delta_{50}\right\}. \tag{3.5}$$

where $\mathbf{a}_1^T, \mathbf{a}_2^T, \ldots, \mathbf{a}_{80}^T$ are the rows of an 80×50 matrix generated by the commands

```
>> randn('seed',315);
>> A=randn(80,50);
```

One way to solve the problem is to use `prox_subgradient` by setting in its model $(f(\mathbf{x}) + \lambda g(\mathbf{x}))$:

$$f(\mathbf{x}) = \max_{i=1,2,\ldots,80}\{\mathbf{a}_i^T\mathbf{x}\}, g(\mathbf{x}) = \delta_{\Delta_{50}}(\mathbf{x}), \lambda = 1.$$

The proximal operator of g is the orthogonal projection onto the unit-simplex, which is implemented in the MATLAB function proj_simplex (see Table 3). To solve the problem using `prox_subgradient`, we require a function that computes a subgradient of f. A subgradient of f at $\mathbf{x}$ is given by $\mathbf{a}_{i(\mathbf{x})}$, where $i(\mathbf{x})$ is any member of $\underset{i=1,2,\ldots,80}{\operatorname{argmin}} \{\mathbf{a}_i^T \mathbf{x}\}$. Following is a MATLAB function implementing a computation of a subgradient of f that should be saved as an m-file called `f_sgrad.m`.

```
function out=f_sgrad(x,A)

[~,i]=max(A*x);
out=A(i,:)';
```

Running the solver with 10000 iterations (the default is 1000) and starting point $(\frac{1}{50}, \frac{1}{50}, \ldots, \frac{1}{50})^T$ yields the following output.

```
>> clear par
>> par.max_iter=10000;
>> [out,fmin,parout] =prox_subgradient(@(x)max(A*x),@(x)f_sgrad(x,A),...
@(x)0,@(x,a)proj_simplex(x),1,1/50*ones(50,1),par);
 **********************
prox_subgradient
**********************
#iter        fun. val.
   344          0.340105
   469          0.304347
   773          0.295849
    :              :
  5857          0.183907
  6592          0.169232
  7047          0.158440
---------------------------------
Optimal value =        0.158440
```

It is possible to change some basic parameters of the method that might accelerate (or slow down) the speed of convergence. For example, the stepsize of the proximal subgradient method is given by $\frac{\alpha}{\sqrt{k+1}}$ with k being the iteration index. The default value of α is 1. Changing this value to 0.2 (by setting the value of `par.alpha` to 0.2) yields faster convergence, and consequently a lower function value.

```
>> par.alpha=0.2;
>> [out,fmin,parout] =prox_subgradient(@(x)max(A*x),@(x)f_sgrad(x,A),@(x)0,...
@(x,a)proj_simplex(x),1,1/50*ones(50,1),par);
**********************
prox_subgradient
**********************
#iter        fun. val.
    17          0.322719
    45          0.298787
    91          0.285807
    :              :
  5675          0.081925
```

```
  7103          0.074788
  9926          0.074581
---------------------------------
Optimal value =        0.074581
```

In the context of optimization over the unit-simplex, a better method is the co-mirror descent method implemented in the MATLAB function `comd` whose syntax is

```
[out,fmin,parout] = comd(Ffun,Ffun_sgrad,Gfun,Gfun_sgrad,set,startx,[par])
```

`comd` can also handle additional functional constraints of the form $g_i(\mathbf{x}) \leq 0$ that should be inserted through the input `Gfun`. In our example, there are no additional functional constraints, and thus `Gfun` and `Gfun_sgrd` should both be empty function handles (`[]`). The input `set` in our case is the unit simplex, and thus should be fixed to be `'simplex'`. Running the method for 10000 iterations gives the following output.

```
>> clear parmd
>> parmd.max_iter=10000;
>> comd(@(x)max(A*x),@(x)f_sgrad(x,A),[],[],'simplex',1/50*ones(50,1),parmd);

*********************
Co-Mirror
*********************
#iter        fun. val.
     1          0.350156
     2          0.312403
     4          0.279940
     :              :
  6901          0.050879
  7727          0.050688
  9977          0.050557
---------------------------------
Optimal value =        0.050557
```

Clearly, `comd` was able to find a better solution than `prox_subgradient`. It is thus also important to choose the "correct" method among the solvers from the same group. As a rule of thumb, if the problem at hand fits the model relevant for `comd` (see Table 1), then it is better to use it rather than `prox_subgradient`.

3.4 l_1-Regularized Least Squares

Consider the problem

$$\min_{\mathbf{x}\in\mathbb{R}^{100}} \frac{1}{2}\|\mathbf{A}\mathbf{x}-\mathbf{b}\|_2^2 + 2\|\mathbf{x}\|_1, \tag{3.6}$$

where $\mathbf{A} \in \mathbb{R}^{80\times 100}$ and $\mathbf{b} \in \mathbb{R}^{100}$ are generated by

```
>> randn('seed',315);
>> A=randn(80,100);
>> b=randn(80,1);
```

We can solve the problem using the proximal gradient method implemented in the MATLAB function `prox_gradient`. The model tackled by the proximal gradient method has the form

$$\min f(\mathbf{x}) + \lambda g(\mathbf{x}),$$

where (in addition to being convex) f is smooth and g is proper and closed. Problem (3.6) fits the above model with $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2, g(\mathbf{x}) = \|\mathbf{x}\|_1, \lambda = 2$. The syntax for `prox_gradient` is

```
[out,fmin,parout] = prox_gradient(Ffun,Ffun_grad,Gfun,Gfun_prox,lambda,startx,[par])
```

To solve the problem using `prox_gradient`, we use the fact that $\nabla f(\mathbf{x}) = \mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{b})$. Invoking `prox_gradient` with 100 iterations starting from the zeros vector yields the following output.

```
>> clear par
>> par.max_iter=100;
>> [out,fmin,parout_pg] =prox_gradient(@(x)0.5*norm(A*x-b,2)^2,@(x)A'*(A*x-b),...
@(x)norm(x,1),@(x,a)prox_l1(x,a),2,zeros(100,1),par);
**********************
prox_gradient
**********************
#iter        fun. val.      L val.
     1         44.647100    256.000000
     2         23.720870    256.000000
     3         20.469023    256.000000
     :             :             :
    98         14.989947    256.000000
    99         14.989876    256.000000
   100         14.989808    256.000000
---------------------------------
Optimal value =       14.989744
```

The proximal gradient method uses a backtracking procedure to find the stepsize at each iteration. The stepsize at iteration k is given by $\frac{1}{L_k}$ where L_k is a certain "Lipschitz estimate". Note that in the above run, all the Lipschitz estimates were chosen as 256, meaning that the backtracking procedure had an effect only at the first iteration (in which the default initial Lipschitz estimate 1 was increased to 256).

FISTA (implemented in the MATLAB function `fista`), which belongs to the third group of solvers, is a better method than proximal gradient. FISTA possesses an improved $O(1/k^2)$ rate of convergence. The syntax of `fista` is the same as the one of `prox_gradient`. Running 100 iterations of FISTA results with a better function value:

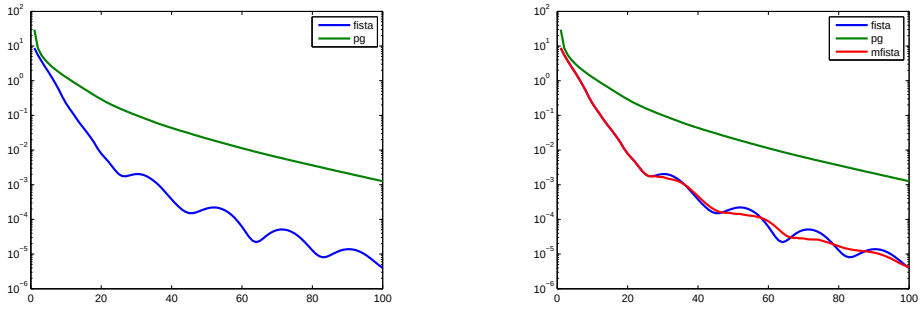
```
>> [out,fmin,parout_fista] =fista(@(x)0.5*norm(A*x-b,2)^2,@(x)A'*(A*x-b),@(x)norm(x,1),...
@(x,a)prox_l1(x,a),2,zeros(100,1),par);
**********************
FISTA
**********************
#iter        fun. val.      L val.
     1         23.720870    256.000000
     2         20.469023    256.000000
     3         18.708294    256.000000
     :             :             :
```

```
    99         14.988551    256.000000
   100         14.988550    256.000000
---------------------------------
Optimal value =       14.988550
```

To make a more detailed comparison between the two methods we plot the distance to optimality in terms of function values of the sequences generated by the two methods. The optimal value is approximated by 10000 iterations of FISTA.

```
>> clear par;
>> par.max_iter=10000;
>> [out,fmin_accurate]=fista(@(x)0.5*norm(A*x-b,2)^2,@(x)A'*(A*x-b),...
@(x)norm(x,1),@(x,a)prox_l1(x,a),2,zeros(100,1),par);
>> semilogy(1:100,parout_fista.funValVec-fmin_accurate,...
1:100,parout_pg.funValVec-fmin_accurate,'LineWidth',2);
>> legend('fista','pg');
```

The plot containing the comparison between the two methods is given in Figure 2(a). As can be clearly seen in Figure 2(a), FISTA is not a monotone method. If one wishes the



(a) without monotone FISTA (b) with monotone FISTA

Figure 2: Comparison between proximal gradient, FISTA and monotone FISTA.

method to produce a nonincreasing sequence of function values, then it is possible to invoke the monotone version of FISTA by setting `par.monotone_flag` to `true`. We can also prevent any screen output by setting `par.print_flag` to be `false`. The resulting plot is shown in Figure 2(b).

```
par.max_iter=100;
par.monotone_flag=true;
par.print_flag=false;
[out,fmin,parout_mfista] =fista(@(x)0.5*norm(A*x-b,2)^2,@(x)A'*(A*x-b),@(x)norm(x,1),...
@(x,a)prox_l1(x,a),2,zeros(100,1),par);
figure(2)
semilogy(1:100,parout_fista.funValVec-fmin_accurate,1:100,parout_pg.funValVec-...
fmin_accurate,1:100,parout_mfista.funValVec-fmin_accurate,'LineWidth',2);
legend('fista','pg','mfista');
```

3.5 Nonconvexity

Although the solvers in the FOM package require the input function to be convex, they will not prevent the user from inserting nonconvex functions and no error message will be

returned (like in CVX [12] for example). In cases where the input function is supposed to be smooth and convex, it is possible to plugin a smooth and *nonconvex* function, and the solver should work, but instead of guaranteeing convergence to an optimal solution, the corresponding method is guaranteed at best to converge to a stationary point. As an example, consider the problem

$$\min_{\mathbf{x}\in\mathbb{R}^3}\{\mathbf{x}^T\mathbf{A}\mathbf{x} : \|\mathbf{x}\|_2 \leq 1\}, \tag{3.7}$$

where

```
>> A=[1,1,4;1,1,4;4,4,-2];
```

$\mathbf{A}$ is not positive semidefinite, and thus the problem is nonconvex. It is easy to see that the stationary points of the problem are the eigenvector corresponding to the minimum eigenvalue -6 and the zeros vector, the former being the actual optimal solution. Problem (3.7) fits the model

$$\min f(\mathbf{x}) + \lambda g(\mathbf{x})$$

with $f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x}$, $\lambda = 1$ and $g(\mathbf{x}) = \delta_C(\mathbf{x})$, where $C = \{\ \mathbf{x} \in \mathbb{R}^3 : \|\mathbf{x}\|_2 \leq 1\}$. Invoking the proximal gradient method with starting point $(0, -1, 0)^T$ actually results with the optimal solution

```
>>  out =prox_gradient(@(x)x'*A*x,@(x)2*A*x,@(x)0,@(x,a)proj_Euclidean_ball(x),1,[0;-1;0]);
**********************
prox_gradient
**********************
#iter        fun. val.      L val.
     1          1.000000      8.000000
     2         -3.538462      8.000000
     3         -5.537778      8.000000
     :              :             :
    12         -6.000000      8.000000
    13         -6.000000      8.000000
    14         -6.000000      8.000000
Stopping because the norm of the difference between consecutive iterates is too small
---------------------------------
Optimal value =       -6.000000
>> out
out =

   -0.4082
   -0.4083
    0.8165
```

Note that the function handle for g is the zeros function, which has the correct value of g on its domain. In general, when inputting a function handle representing an extended real-valued function to one of the FOM solvers, it is only important that it will be consistent with the function over its domain.

Starting from $(1, 1, 1)^T$ produces the zeros vector which is just a stationary point.

```
>> out =prox_gradient(@(x)x'*A*x,@(x)2*A*x,@(x)0,@(x,a)proj_Euclidean_ball(x),1,[1;1;1]);
**********************
prox_gradient
```

```
********************
#iter        fun. val.      L val.
     1          6.000000     16.000000
     2          0.375000     16.000000
     3          0.023437     16.000000

     8          0.000000     16.000000
     9          0.000000     16.000000
    10          0.000000     16.000000
Stopping because the norm of the difference between consecutive iterates is too small
---------------------------------
Optimal value =        0.000000
>> out
out =

   1.0e-06 *

    0.5506
    0.5506
    0.5506
```

3.6 One-Dimensional Signal Denoising

Consider the following denoising problem:

$$\min_{\mathbf{x}\in\mathbb{R}^{1000}} \left\{ \frac{1}{2}(x_i - y_i)^2 + 4\sum_{i=1}^{999} |x_i - x_{i+1}| \right\}, \tag{3.8}$$

where $\mathbf{y}$ is a noisy step function generated as follows ($\mathbf{x}$ being the original step function):

```
>> randn('seed',314);
>> x=zeros(1000,1);
>> x(1:250)=1;
>> x(251:500)=3;
>> x(751:1000)=2;
>> y=x+0.05*randn(size(x));
```

We can plot "true" and noisy signals (see Figure 3).

```
>> figure(3)
>> subplot(1,2,1)
>> plot(1:1000,x,'.')
>> subplot(1,2,2)
>> plot(1:1000,y,'.')
```

The problem can be solved using the fast dual proximal gradient (FDPG) method implemented in the MATLAB function `fdpg`. The model tackled by the FDPG method is (see Table 1)

$$\min_{\mathbf{x}} f(\mathbf{x}) + \lambda g(\mathcal{A}\mathbf{x}),$$

where f is strongly-convex, $\lambda > 0$, $\mathcal{A}$ is a linear transformation and g is proper closed convex and proximable. The denoising problem (3.8) fits the above model with $\lambda = 4$, $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x} - \mathbf{y}\|^2$, $g(\mathbf{y}) = \|\mathbf{y}\|_1$ and $\mathcal{A} : \mathbb{R}^{1000} \to \mathbb{R}^{999}$ being the linear transformation for which $(\mathcal{A}\mathbf{x})_i = x_i - x_{i+1}, i = 1, 2, \ldots, 999$.

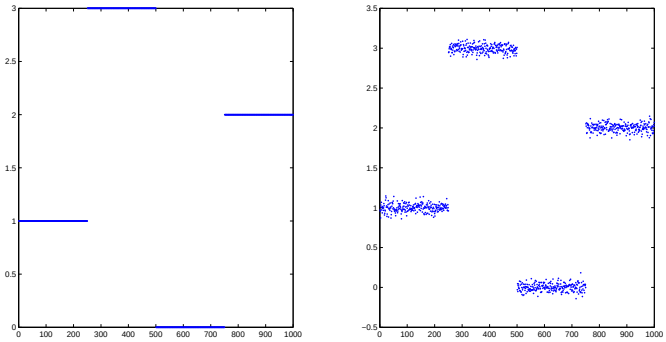


Figure 3: True and noisy step functions.

```
A=sparse(999,1000);
for i=1:999
    A(i,i)=1;
    A(i,i+1)=-1;
end
```

The syntax of `fdpg` is

```
[out,fmin,parout] = fdpg(Ffun,F_grad_conj,Gfun,Gfun_prox,Afun,Atfun,lambda,starty,[par])
```

The second input argument of `fdpg` is the gradient of f^*, which in this case is given by

$$\nabla f^*(\mathbf{x}) = \underset{\mathbf{z}}{\operatorname{argmax}} \{\langle \mathbf{x}, \mathbf{z} \rangle - f(\mathbf{z})\} = \underset{\mathbf{z}}{\operatorname{argmax}} \left\{ \langle \mathbf{x}, \mathbf{z} \rangle - \frac{1}{2}\|\mathbf{z} - \mathbf{y}\|^2 \right\} = \mathbf{x} + \mathbf{y}.$$

The following command computes an optimal solution of (3.8) using `fdpg`

```
>> [out,fmin,parout] = fdpg(@(x)0.5*norm(x-y)^2,@(x)x+y,@(x)norm(x,1),@(x,a)prox_l1(x,a),...
@(x)A*x,@(x)A'*x,4,zeros(999,1));
**********************
fdpg
**********************
#iter        fun. val.    feas. viol.  L val.
     1        248.511179      3.1607e-07      4.000000
     2        107.310785      3.1607e-07      4.000000
     3         74.301824      3.1607e-07      4.000000
     :             :               :               :
   998         28.904498     3.14599e-07      4.000000
   999         28.899496     3.13701e-07      4.000000
  1000         28.895505     3.13267e-07      4.000000
---------------------------------
Optimal value =       28.895505
```

Since the function g is real-valued, it is better to invoke the solver with `par.real_valued_flag` set to `true`, since in this case there is no need to check for feasibility violation and the algorithm outputs the iterate with the smallest function value.

```
>> clear par
>> par.real_valued_flag=true;
>> [out,fmin,parout] = fdpg(@(x)0.5*norm(x-y)^2,@(x)x+y,@(x)norm(x,1),@(x,a)prox_l1(x,a),...
```

```
@(x)A*x,@(x)A'*x,4,zeros(999,1),par);
*********************
fdpg
*********************
#iter        fun. val.    L val.
     2        107.310785      4.000000
     3         74.301824      4.000000
     4         60.396805      4.000000
     :             :              :
   498         28.910757      4.000000
   499         28.899488      4.000000
   500         28.892469      4.000000
-----------------------------------
Optimal value =       28.892469
```

The last recorded iteration is 500 since there was no improvement of function value following that iteration (although 1000 iterations were employed). Note also that a slightly smaller function value was obtained in this case. The obtained solution is an excellent reconstruction of the original signal (see Figure 4)

```
>> figure(4);
>> plot(1:1000,out,'.')
```

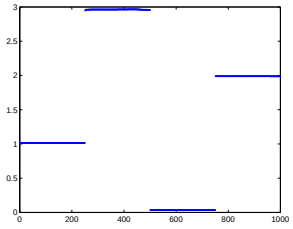


Figure 4: Reconstructed signal using fdpg.

4 Matrix Variables

With the exception of `nested_fista`, all the solvers in the FOM package are able to solve problems with matrix variables and are not restricted to solve problems over column vectors. For example, consider the problem

$$\min_{\mathbf{X}\in\mathbb{R}^{30\times 40}} \left\{ \frac{1}{2}\|\mathbf{C}\odot(\mathbf{X}-\mathbf{D})\|_F^2 + \|\mathbf{AXB}\|_F \right\},$$

where $\odot$ denotes the componentwise Hadamard product (that is, component-wise product) and $\mathbf{C},\mathbf{D}\in\mathbb{R}^{30\times 30}$,$\mathbf{A}\in\mathbb{R}^{20\times 30}$ and $\mathbf{B}\in\mathbb{R}^{40\times 50}$ are generated by the commands

```
>> randn('seed',314);
>> rand('seed',314);
>> A=randn(20,30);
>> B=randn(40,50);
>> C=1+rand(30,40);
>> D=randn(30,40);
```

To solve the problem, we will use the FDPG method with

$$f(\mathbf{X}) = \|\mathbf{C} \odot (\mathbf{X} - \mathbf{D})\|_F^2, g(\mathbf{Y}) = \|\mathbf{Y}\|_F, \lambda = 1, \mathcal{A}(\mathbf{X}) \equiv \mathbf{AXB}.$$

Note that since $\mathbf{C}$ has only nonzero components, f is strongly convex as required. In employing the FDPG method, we will use the following facts:

- the adjoint linear transformation is given by $\mathbf{X} \mapsto \mathbf{A}^T\mathbf{X}\mathbf{B}^T$;
- the gradient of the conjugate of f is given by $\nabla f^*(\mathbf{Y}) = \underset{\mathbf{X}}{\text{argmax}}\{\langle \mathbf{X}, \mathbf{Y}\rangle - f(\mathbf{X})\} = \mathbf{E} \odot \mathbf{Y} + \mathbf{D}$.

where $\mathbf{E}$ is the matrix defined by $E_{ij} = \frac{1}{C_{ij}^2}, i = 1, 2, \ldots, 30, j = 1, 2, \ldots, 40$.

The command invoking `fdpg` is

```
>> E = 1./(C.^2);
>> clear par
>> par.real_valued_flag=true;
>> [out,fmin,parout] = fdpg(@(X)0.5*norm(C.*(X-D),'fro')^2,@(X)E.*X+D,...
@(X)norm(X,'fro'),@(x,a)prox_Euclidean_norm(x,a),@(X)A*X*B,@(X)A'*X*B',1,zeros(20,50),par);

**********************
fdpg
**********************
#iter        fun. val.    L val.
     2        693.740808   4096.000000
     3        623.515605   4096.000000
     6        553.714050   8192.000000
        :                        :                    :
   281        485.921545   8192.000000
   282        485.921545   8192.000000
   283        485.921545   8192.000000
Stopping because the norm of the difference between consecutive iterates is too small
---------------------------------
Optimal value =      485.921545
```

4.1 Finding a point in the intersection of balls

Consider the problem of finding a point in the intersection of 5000 balls of dimension 200

$$\|\mathbf{x} - \mathbf{c}_i\| \leq r_i, \ i = 1, 2, \ldots, 5000,$$

where the centers and radii are generated by the commands

```
>> randn('seed',315);
>> rand('seed',315);
>> n=200;
>> m=5000;
>> x_true=randn(n,1);
>> r_all=[];
>> c_all=[];
>> for k=1:m
```

```
>>    r=rand;
>>    r_all=[r_all;r];
>>    d=randn(n,1);
>>    d=d/norm(d);
>>    c=x_true+0.9*r*d;
>>    c_all=[c_all,c];
>> end
```

The above process starts by choosing randomly a point `x_true` and then generates 5000 balls that are guaranteed to contain `x_true`. To solve the problem, we first formulate it as the following minimization problem:

$$\min_{\mathbf{x}\in\mathbb{R}^{200}} \sum_{i=1}^{5000} \left[\|\mathbf{x}-\mathbf{c}_i\|_2^2 - r_i^2\right]_+^2. \tag{4.1}$$

To solve the problem, we will use the nested FISTA method, which is a simple generalization of FISTA aimed at solving problems of the form (see Table 1)

$$\min_{\mathbf{x}} \varphi(\mathbf{f}(\mathbf{x})) + g(\mathcal{A}\mathbf{x}), \tag{4.2}$$

where (in addition to convexity of all the functions involved) φ is a Lipschitz continuous and (componentwise) nondecreasing function, $\mathbf{f}$ is a vector-valued functions whose components are smooth, $\mathcal{A}$ is a linear transformation and g is proper and closed. Problem (4.1) fits the model (4.1) with (the choice below of $\mathcal{A}$ is actually arbitrary)

$$\varphi(\mathbf{y}) = \sum_{i=1}^{5000}[y_i]_+, \mathbf{f}(\mathbf{x}) = \begin{pmatrix} \|\mathbf{x}-\mathbf{c}_1\|^2 - r_1^2 \\ \|\mathbf{x}-\mathbf{c}_2\|^2 - r_2^2 \\ \vdots \\ \|\mathbf{x}-\mathbf{c}_{5000}\|^2 - r_{5000}^2 \end{pmatrix}, g(\mathbf{x}) \equiv 0, \mathcal{A} = \mathcal{I}.$$

To solve the problem using `nested_fista`, we will exploit the following formulas:

$$\mathrm{prox}_{\alpha\varphi}(y) = \mathcal{T}_{\frac{\alpha}{2}}\left(\mathbf{y} - \frac{\alpha}{2}\mathbf{e}\right), \nabla f(\mathbf{x}) = 2\begin{pmatrix}\mathbf{x}-\mathbf{c}_1 & \mathbf{x}-\mathbf{c}_2 & \cdots & \mathbf{x}-\mathbf{c}_{5000}\end{pmatrix}.$$

where $\mathcal{T}_\beta$ is the soft-thresholding operator (also implemented in the MATLAB function `prox_l1`) and $\mathbf{e}$ is the vector of all ones. With the above formulas in mind, we can define

```
>> phi=@(y)sum(pos(y));
>> prox_phi=@(x,a)prox_l1(x-a/2,a/2);
>> f=@(x)(sum_square(x*ones(1,m)-c_all)-(r_all.^2)')';
>> grad_f=@(x)2*(x*ones(1,m)-c_all);
```

The syntax of `nested_fista` is

```
[out,fmin,parout] = nested_fista(Phifun,Phifun_prox,Ffun,Ffun_grad,Gfun,Gfun_prox,...
Afun,Atfun,lambda,startx,[par])
```

We can now solve the problem

```
>> [xf,fun_xf,parout ]  = nested_fista ( @(x) phi(x) , @(x,a) prox_phi(x,a), @(x) f(x),...
@(x) grad_f(x), @(x)0 ,@(x,a)x,@(x) x,@(x) x, 1,zeros(n,1)) ;
**********************
nested-fista
**********************
#iter        fun. val.     L val.   inner L val.   inner iternu.
     1      283431.366571   16384.000000      256        50
     2      71260.317218   16384.000000      128        50
     3       9347.554536   16384.000000       64        50
     :              :           :             :          :
   991          0.000000   16384.000000   2.842171e-14    2
   992          0.000000   16384.000000   2.842171e-14    2
   993          0.000000   16384.000000   5.684342e-14    2
Stopping because of 100 iterations with no improvement
---------------------------------
Optimal value =        0.000000
```

The obtained solution is a good reconstruction of the vector `x_true`

```
>> norm(x_true-xf)

ans =

   3.6669e-04
```

References

[1] A. Beck. *First Order Methods in Optimization*. To appear in MPS/SIAM Series on Optimization, 2017.

[2] A. Beck, A. Ben-Tal, N. Guttmann-Beck, and L. Tetruashvili. The CoMirror algorithm for solving nonsmooth constrained convex problems. *Oper. Res. Lett.*, 38(6):493–498, 2010.

[3] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J. Imaging Sci.*, 2(1):183–202, 2009.

[4] A. Beck and M. Teboulle. Gradient-based algorithms with applications to signal-recovery problems. In *Convex optimization in signal processing and communications*, pages 42–88. Cambridge Univ. Press, Cambridge, 2010.

[5] A. Beck and M. Teboulle. Smoothing and first order methods: a unified framework. *SIAM J. Optim.*, 22(2):557–580, 2012.

[6] A. Beck and M. Teboulle. A fast dual proximal gradient algorithm for convex minimization and applications. *Oper. Res. Lett.*, 42(1):1–6, 2014.

[7] S. R. Becker, E. J. Candès, and M. C. Grant. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical Programming Computation*, 3(3):165, Jul 2011.

[8] P. L. Combettes and V. R. Wajs. Signal recovery by proximal forward-backward splitting. *Multiscale Model. Simul.*, 4(4):1168–1200, 2005.

[9] D. L. Donoho. De-noising by soft-thresholding. *IEEE Transactions on Information Theory*, 41(3):613–627, May 1995.

[10] J. C. Duchi, S. Shalev-Shwartz, Y. Singer, and A. Tewari. Composite objective mirror descent. In *COLT 2010 - The 23rd Conference on Learning Theory*, pages 14–26, 2010.

[11] J. Eckstein. Some saddle-function splitting methods for convex programming. *Optim. Methods Softw.*, 4:75–83, 1994.

[12] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 2.0 beta. `http://cvxr.com/cvx`, September 2013.

[13] B. He and X. Yuan. On the $O(1/n)$ convergence rate of the Douglas-Rachford alternating direction method. *SIAM J. Numer. Anal.*, 50(2):700–709, 2012.

[14] J. J. Moreau. Proximité et dualité dans un espace hilbertien. *Bull. Soc. Math. France*, 93:273–299, 1965.

[15] R. Shefi and M. Teboulle. Rate of convergence analysis of decomposition methods based on the proximal method of multipliers for convex minimization. *SIAM J. Optim.*, 24(1):269–297, 2014.

[16] J. F. Sturm. Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11-12:625–653, 1999.

[17] K. C. Toh, M. J. Todd, and R. H. Tütüncü. SDPT3—a MATLAB software package for semidefinite programming, version 1.3. *Optim. Methods Softw.*, 11/12(1-4):545–581, 1999. Interior point methods.

MATLAB function	function	Assumptions
prox_quadratic	convex quadratic $\alpha\left(\frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x}+\mathbf{b}^T\mathbf{x}\right)$	$\mathbf{A}\in\mathbb{S}^n_+$
prox_Euclidean_norm*	Euclidean norm $\alpha\|\mathbf{x}\|_2$	
prox_l1*	l_1-norm $\alpha\|\mathbf{x}\|_1$	
prox_neg_sum_log*	negative sum of logs $-\alpha\sum_{i=1}^n\log x_i$	
prox_linf*	$l_\infty-$norm $\alpha\|\mathbf{x}\|_\infty$	
prox_max*	maximum $\alpha\max\{x_1,\ldots,x_n\}$	
prox_Huber*	Huber $\alpha H_\mu(\mathbf{x})=\alpha\begin{cases}\frac{1}{2\mu}\|\mathbf{x}\|_2^2 & \|\mathbf{x}\|\le\mu\\ \|\mathbf{x}\|-\frac{\mu}{2} & \|\mathbf{x}\|>\mu\end{cases}$	$\mu>0$
prox_sum_k_largest*	sum of k largest values $\alpha\sum_{i=1}^k x_{[i]}$	$k\in\{1,2,\ldots,n\}$
prox_sum_k_largest_abs*	sum of k largest absolute values $\alpha\sum_{i=1}^k\|x_{\langle i\rangle}\|$	$k\in\{1,2,\ldots,n\}$
prox_norm2_linear	l_2 norm of a linear transformation $\alpha\|\mathbf{A}\mathbf{x}\|_2$	$\mathbf{A}$ with full row rank
prox_l1_squared*	squared l_1-norm $\alpha\|\mathbf{x}\|_1^2$	
prox_max_eigenvalue	maximum eigenvalue $\alpha\lambda_{\max}(\mathbf{X})$	$\mathbf{X}\in\mathbb{S}^n$
prox_neg_log_det	negative log determinant $-\alpha\log(\det(\mathbf{X}))$	$\mathbf{X}\in\mathbb{S}^n$
prox_sum_k_largest_eigenvalues	sum of k largest eigenvalues $\alpha\sum_{i=1}^k\lambda_i(\mathbf{X})$	$\mathbf{X}\in\mathbb{S}^n$, $k\in\{1,2,\ldots,n\}$
prox_spectral	spectral norm $\alpha\|\mathbf{X}\|_{2,2}=\alpha\sigma_1(\mathbf{X})$	
prox_nuclear	nuclear norm $\alpha\|\mathbf{X}\|_{S_1}=\alpha\sum_{i=1}^{\min\{m,n\}}\sigma_i(\mathbf{X})$	$\mathbf{X}\in\mathbb{R}^{m\times n}$
prox_Ky_Fan	Ky Fan norm $\alpha\|\mathbf{X}\|_{\langle k\rangle}=\alpha\sum_{i=1}^k\sigma_i(\mathbf{X})$	$\mathbf{X}\in\mathbb{R}^{m\times n}$, $1\le k\le\min\{m,n\}$

Table 2: List of prox functions implemented in the FOM package. All functions assume that α is a positive scalar parameter. Functions marked by * operate on $m\times n$ matrices in the same way they operate on the corresponding mn-length column vector.

MATLAB function	set	Assumptions
proj_Euclidean_ball*	Euclidean ball $B[\mathbf{c}, r] = \{\mathbf{x} : \|\mathbf{x} - \mathbf{c}\| \leq r\}$	$\mathbf{c} \in \mathbb{R}^n$, $r > 0$
proj_box*	box $\text{Box}[\mathbf{l}, \mathbf{u}] = \{\mathbf{x} : \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\}$	$\mathbf{l} \leq \mathbf{u}$
proj_affine_set	affine set $\{\mathbf{x} : \mathbf{A}\mathbf{x} = \mathbf{b}\}$	$\mathbf{A}$ with full row rank
proj_halfspace*	half-space $H^-_{\mathbf{a},b} = \{\mathbf{x} : \langle \mathbf{a}, \mathbf{x} \rangle \leq b\}$	$\mathbf{a} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$,$b \in \mathbb{R}$
proj_two_halfspaces*	intersection of two half-spaces $H^-_{\mathbf{a}_1,b_1} \cap H^-_{\mathbf{a}_2,b_2}$ $= \{\mathbf{x} : \langle \mathbf{a}_1, \mathbf{x} \rangle \leq b_1, \langle \mathbf{a}_2, \mathbf{x} \rangle \leq b_2\}$	$\{\mathbf{a}_1, \mathbf{a}_2\}$ independent
proj_Lorentz	Lorentz cone $L^n = \{\mathbf{x} \in \mathbb{R}^{n+1} : \|\mathbf{x}_{\{1,\ldots,n\}}\| \leq x_{n+1}\}$	
proj_hyperplane_box*	intersection of a hyperplane and a box $H_{\mathbf{a},b} \cap \text{Box}[\mathbf{l}, \mathbf{u}]$ $= \{\mathbf{x} : \langle \mathbf{a}, \mathbf{x} \rangle = b, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\}$	$H_{\mathbf{a},b} \cap \text{Box}[\mathbf{l}, \mathbf{u}] \neq \emptyset$
proj_halfspace_box*	intersection of a half-space and a box $H^-_{\mathbf{a},b} \cap \text{Box}[\mathbf{l}, \mathbf{u}]$ $= \{\mathbf{x} : \langle \mathbf{a}, \mathbf{x} \rangle \leq b, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\}$	$H^-_{\mathbf{a},b} \cap \text{Box}[\mathbf{l}, \mathbf{u}] \neq \emptyset$
proj_simplex*	r-simplex $\Delta_n(r) = \{\mathbf{x} : \mathbf{e}^T\mathbf{x} = r, \mathbf{x} \geq \mathbf{0}\}$ r-full simplex $\Delta^+_n(r) = \{\mathbf{x} : \mathbf{e}^T\mathbf{x} \leq r, \mathbf{x} \geq \mathbf{0}\}$	$r > 0$
proj_product*	product superlevel set $\{\mathbf{x} > \mathbf{0} : \pi_{i=1}^n x_i \geq r\}$	$r > 0$
proj_l1_ball*	l_1 ball $\{\mathbf{x} : \|\mathbf{x}\|_1 \leq r\}$	$r > 0$
proj_l1ball_box*	intersection of weighted l_1 ball and a box $\{\mathbf{x} : \|\mathbf{w} \odot \mathbf{x}\|_1 \leq r, \|\mathbf{x}\|_\infty \leq u\}$	$r, u \geq 0$,$\mathbf{w} \geq \mathbf{0}$
proj_psd	cone of positive semidefinite matrices $\mathbb{S}^n_+ = \{\mathbf{X} : \mathbf{X} \succeq \mathbf{0}\}$	
proj_spectral_box_sym	spectral box (in $\mathbb{S}^n$) $\{\mathbf{X} \in \mathbb{S}^n : \ell\mathbf{I} \preceq \mathbf{X} \preceq u\mathbf{I}\}$	$\ell \leq u$, sym. input matrix
proj_spectral_ball	spectral-norm ball $B_{\|\cdot\|_{S_\infty}}[\mathbf{0}, r] = \{\mathbf{X} : \sigma_1(\mathbf{X}) \leq r\}$	$r > 0$
proj_nuclear_ball	nuclear-norm ball $B_{\|\cdot\|_{S_1}}[\mathbf{0}, r] = \{\mathbf{X} : \sum_i \sigma_i(\mathbf{X}) \leq r\}$	$r > 0$
proj_spectahedron	r-spectahedron $\Upsilon_n(r) = \{\mathbf{X} \in \mathbb{S}^n_+ : \text{Tr}(\mathbf{X}) = r\}$ r-full spectahedron $\Upsilon^+_n(r) = \{\mathbf{X} \in \mathbb{S}^n_+ : \text{Tr}(\mathbf{X}) \leq r\}$	$r > 0$, sym. input matrix

Table 3: List of orthogonal projection functions implemented in the FOM package. Functions marked by * operate on $m \times n$ matrices in the same way they operate on the corresponding mn-length column vector.