PyUNLocBoX documentation

Release 0.2.1

Michaël Defferrard, EPFL LTS2

August 01, 2016

Contents

PyUNLocBoX is a convex optimization toolbox using proximal splitting methods implemented in Python. It is a free software distributed under the BSD license and is a port of the Matlab UNLocBoX toolbox.

- Development : https://github.com/epfl-lts2/pyunlocbox
- Documentation : https://pyunlocbox.readthedocs.io
- PyPI package : https://pypi.python.org/pypi/pyunlocbox
- Travis continuous integration : https://travis-ci.org/epfl-lts2/pyunlocbox
- UNLocBoX matlab toolbox : https://lts2.epfl.ch/unlocbox

CHAPTER 1

Features

- Solvers
  - Forward-backward splitting algorithm
  - Douglas-Rachford splitting algorithm
  - Monotone+Lipschitz Forward-Backward-Forward primal-dual algorithm
  - Projection-based primal-dual algorithm
- Proximal operators
  - L1-norm
  - L2-norm
  - TV-norm
  - Projection on the L2-ball

Following is a typical usage example who solves an optimization problem composed by the sum of two convex functions. The functions and solver objects are first instantiated with the desired parameters. The problem is then solved by a call to the solving function.

```
>>> import pyunlocbox
>>> f1 = pyunlocbox.functions.norm_l2(y=[4, 5, 6, 7])
>>> f2 = pyunlocbox.functions.dummy()
>>> solver = pyunlocbox.solvers.forward_backward()
>>> ret = pyunlocbox.solvers.solve([f1, f2], [0., 0, 0, 0], solver, atol=1e-5)
Solution found after 10 iterations:
    objective function f(sol) = 7.460428e-09
    stopping criterion: ATOL
>>> ret['sol']
array([ 3.99996922,  4.99996153,  5.99995383,  6.99994614])
```

CHAPTER 2

Installation

PyUnLocBox is continuously tested on Python 2.7, 3.3, 3.4 and 3.5.

System-wide installation:

```
$ pip install pyunlocbox
```

Installation in an isolated virtual environment:

```
$ mkvirtualenv --system-site-packages pyunlocbox
$ pip install pyunlocbox
```

You need virtualenvwrapper to run this command. The `--system-site-packages` option could be useful if you want to use a shared system installation of numpy and matplotlib. Their building and installation require quite some dependencies.

Another way is to manually download from PyPI, unpack the package and install with:

```
$ python setup.py install
```

Execute the project test suite once to make sure you have a working install:

```
$ python setup.py test
```

CHAPTER 3

Authors

PyUNLocBoX was started in 2014 as an academic project for research purpose at the LTS2 laboratory from EPFL (https://lts2.epfl.ch).

Development lead :

- Michaël Defferrard from EPFL LTS2 <michael.defferrard@epfl.ch>
- Nathanaël Perraudin from EPFL LTS2 <nathanael.perraudin@epfl.ch>

Contributors :

- Alexandre Lafaye from EPFL LTS2 <alexandre.lafaye@epfl.ch>
- Basile Châtillon from EPFL LTS2 <basile.chatillon@epfl.ch>
- Nicolas Rod from EPFL LTS2 <nicolas.rod@epfl.ch>
- Rodrigo Pena from EPFL LTS2 <rodrigo.pena@epfl.ch>

3.1 Tutorials

The following are some tutorials which show and explain how to use the toolbox to solve some real problems. They goes in increasing degree of difficulty. If you have never used the toolbox before, you are encouraged to follow them in order as they build one upon the other.

3.1.1 Simple least square problem

This simplistic example is only meant to demonstrate the basic workflow of the toolbox. Here we want to solve a least square problem, i.e. we want the solution to converge to the original signal without any constraint. Lets define this signal by :

```
>>> y = [4, 5, 6, 7]
```

The first function to minimize is the sum of squared distances between the current signal x and the original y. For this purpose, we instantiate an L2-norm object :

```
>>> from pyunlocbox import functions
>>> f1 = functions.norm_l2(y=y)
```

This standard function object provides the `eval()`, `grad()` and `prox()` methods that will be useful to the solver. We can evaluate them at any given point :

```
>>> f1.eval([0, 0, 0, 0])
126
>>> f1.grad([0, 0, 0, 0])
array([ -8, -10, -12, -14])
```

```
>>> f1.prox([0, 0, 0, 0], 1)
array([ 2.66666667,  3.33333333,  4.        ,  4.66666667])
```

We need a second function to minimize, which usually describes a constraint. As we have no constraint, we just define a dummy function object by hand. We have to define the `_eval()` and `_grad()` methods as the solver we will use requires it :

```
>>> f2 = functions.func()
>>> f2._eval = lambda x: 0
>>> f2._grad = lambda x: 0
```

Note: We could also have used the *pyunlocbox.functions.dummy* function object.

We can now instantiate the solver object :

```
>>> from pyunlocbox import solvers
>>> solver = solvers.forward_backward()
```

And finally solve the problem :

```
>>> x0 = [0., 0., 0., 0.]
>>> ret = solvers.solve([f2, f1], x0, solver, atol=1e-5, verbosity='HIGH')
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 1.260000e+02
INFO: Forward-backward method: FISTA
Iteration 1 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 1.400000e+01
    objective = 1.40e+01
Iteration 2 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 1.555556e+00
    objective = 1.56e+00
Iteration 3 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 3.293044e-02
    objective = 3.29e-02
Iteration 4 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 8.780588e-03
    objective = 8.78e-03
Iteration 5 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 6.391406e-03
    objective = 6.39e-03
Iteration 6 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 5.713369e-04
    objective = 5.71e-04
Iteration 7 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 1.726501e-05
    objective = 1.73e-05
Iteration 8 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 6.109470e-05
    objective = 6.11e-05
Iteration 9 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 1.212636e-05
    objective = 1.21e-05
Iteration 10 of forward_backward:
```

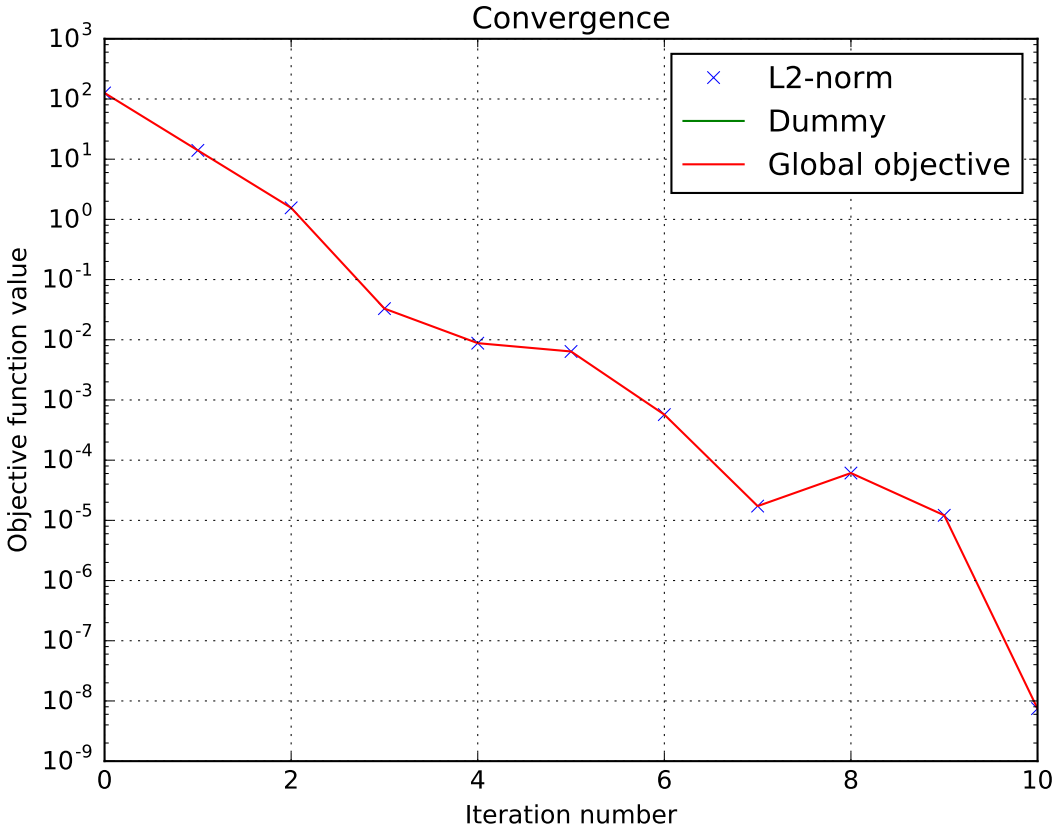
```
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 7.460428e-09
    objective = 7.46e-09
Solution found after 10 iterations:
    objective function f(sol) = 7.460428e-09
    stopping criterion: ATOL
```

The solving function returns several values, one is the found solution :

```
>>> ret['sol']
array([ 3.99996922,  4.99996153,  5.99995383,  6.99994614])
```

Another one is the value returned by each function objects at each iteration. As we passed two function objects (L2-norm and dummy), the *objective* is a 2 by 11 (10 iterations plus the evaluation at *x0*) `ndarray`. Lets plot a convergence graph out of it :

```
>>> import numpy as np
>>> objective = np.array(ret['objective'])
>>> import matplotlib.pyplot as plt
>>> _ = plt.figure()
>>> _ = plt.semilogy(objective[:, 1], 'x', label='L2-norm')
>>> _ = plt.semilogy(objective[:, 0], label='Dummy')
>>> _ = plt.semilogy(np.sum(objective, axis=1), label='Global objective')
>>> _ = plt.grid(True)
>>> _ = plt.title('Convergence')
>>> _ = plt.legend(numpoints=1)
>>> _ = plt.xlabel('Iteration number')
>>> _ = plt.ylabel('Objective function value')
```



The above graph shows an exponential convergence of the objective function. The global objective is obviously only composed of the L2-norm as the dummy function object was defined to always evaluate to 0 (`f2._eval = lambda x: 0`).

3.1.2 Compressed sensing using forward-backward

This tutorial presents a compressed sensing problem solved by the forward-backward splitting algorithm. The convex optimization problem is the sum of a data fidelity term and a regularization term which expresses a prior on the sparsity of the solution, given by

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

where y are the measurements, A is the measurement matrix and τ expresses the trade-off between the two terms.

The number of necessary measurements m is computed with respect to the signal size n and the sparsity level S in order to very often perform a perfect reconstruction. See *[CR07]* for details.

```
>>> n = 5000
>>> S = 100
>>> import numpy as np
>>> m = int(np.ceil(S * np.log(n)))
>>> print('Number of measurements: {}'.format(m))
Number of measurements: 852
>>> print('Compression ratio: {:3.2f}'.format(float(n) / m))
Compression ratio: 5.87
```

We generate a random measurement matrix A:

```
>>> np.random.seed(1)  # Reproducible results.
>>> A = np.random.normal(size=(m, n))
```

Create the S sparse signal x:

```
>>> x = np.zeros(n)
>>> I = np.random.permutation(n)
>>> x[I[0:S]] = np.random.normal(size=S)
>>> x = x / np.linalg.norm(x)
```

Generate the measured signal y:

```
>>> y = np.dot(A, x)
```

The prior objective to minimize is defined by

$$f_1(x) = \tau\|x\|_1$$

which can be expressed by the toolbox L1-norm function object. It can be instantiated as follows, while setting the regularization parameter *tau*:

```
>>> from pyunlocbox import functions
>>> tau = 1.0
>>> f1 = functions.norm_l1(lambda_=tau)
```

The fidelity objective to minimize is defined by

$$f_2(x) = \|Ax - y\|_2^2$$

which can be expressed by the toolbox L2-norm function object. It can be instantiated as follows:

```
>>> f2 = functions.norm_l2(y=y, A=A)
```

or alternatively as follows:

```
>>> f3 = functions.norm_l2(y=y)
>>> f3.A = lambda x: np.dot(A, x)
>>> f3.At = lambda x: np.dot(A.T, x)
```

Note: In this case the forward and adjoint operators were passed as functions not as matrices.

A third alternative would be to define the function object by hand:

```
>>> f4 = functions.func()
>>> f4._grad = lambda x: 2.0 * np.dot(A.T, np.dot(A, x) - y)
>>> f4._eval = lambda x: np.linalg.norm(np.dot(A, x) - y)**2
```

Note: The three alternatives to instantiate the function objects (*f2*, *f3* and *f4*) are strictly equivalent and give the exact same results.

Now that the two function objects to minimize (the L1-norm and the L2-norm) are instantiated, we can instantiate the solver object. The step size for optimal convergence is $\frac{1}{\beta}$ where β is the Lipschitz constant of the gradient of *f2*, *f3*, *f4* given by:

$$\beta = 2 \cdot \|A\|_{\text{op}}^2 = 2 \cdot \lambda_{max}(A^* A).$$

To solve this problem, we use the Forward-Backward splitting algorithm which is instantiated as follows:

```
>>> step = 0.5 / np.linalg.norm(A, ord=2)**2
>>> from pyunlocbox import solvers
>>> solver = solvers.forward_backward(method='FISTA', step=step)
```

Note: A complete description of the constructor parameters and default values is given by the solver object *pyunlocbox.solvers.forward_backward* reference documentation.

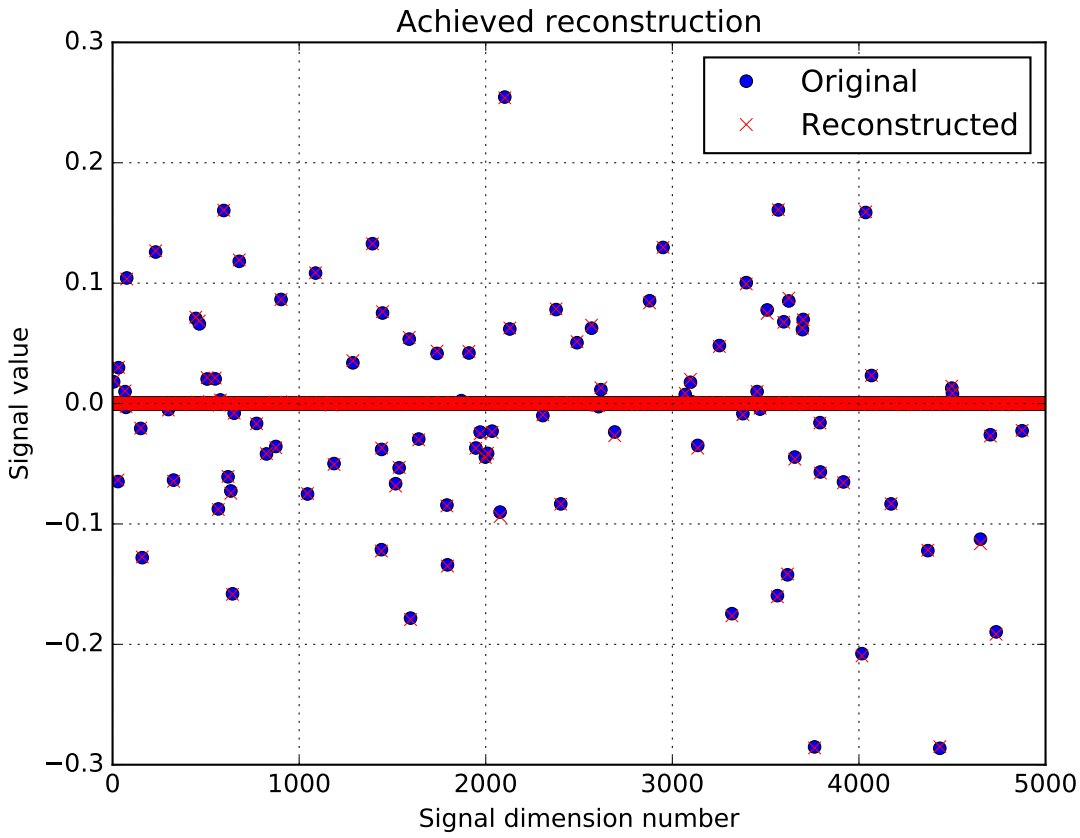
After the instantiations of the functions and solver objects, the setting of a starting point *x0*, the problem is solved by the toolbox solving function as follows:

```
>>> x0 = np.zeros(n)
>>> ret = solvers.solve([f1, f2], x0, solver, rtol=1e-4, maxit=300)
Solution found after 152 iterations:
    objective function f(sol) = 7.668195e+00
    stopping criterion: RTOL
```

Note: A complete description of the parameters, their default values and the returned values is given by the solving function *pyunlocbox.solvers.solve()* reference documentation.

Let's display the results:

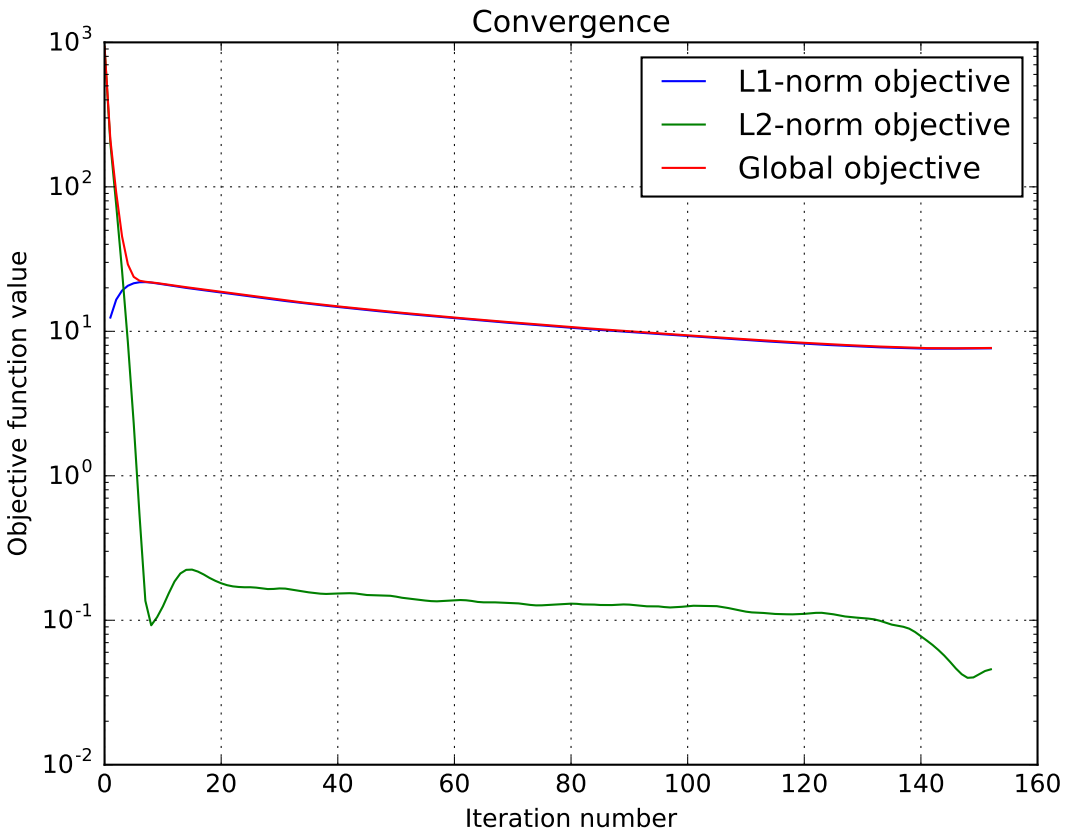
```
>>> import matplotlib.pyplot as plt
>>> _ = plt.figure()
>>> _ = plt.plot(x, 'o', label='Original')
>>> _ = plt.plot(ret['sol'], 'xr', label='Reconstructed')
>>> _ = plt.grid(True)
>>> _ = plt.title('Achieved reconstruction')
>>> _ = plt.legend(numpoints=1)
>>> _ = plt.xlabel('Signal dimension number')
>>> _ = plt.ylabel('Signal value')
```



The above figure shows a good reconstruction which is both sparse (thanks to the L1-norm objective) and close to the measurements (thanks to the L2-norm objective).

Let's display the convergence of the two objective functions:

```
>>> objective = np.array(ret['objective'])
>>> _ = plt.figure()
>>> _ = plt.semilogy(objective[:, 0], label='L1-norm objective')
>>> _ = plt.semilogy(objective[:, 1], label='L2-norm objective')
>>> _ = plt.semilogy(np.sum(objective, axis=1), label='Global objective')
>>> _ = plt.grid(True)
>>> _ = plt.title('Convergence')
>>> _ = plt.legend()
>>> _ = plt.xlabel('Iteration number')
>>> _ = plt.ylabel('Objective function value')
```



3.1.3 Compressed sensing using Douglas-Rachford

This tutorial presents a compressed sensing problem solved by the Douglas-Rachford splitting algorithm. The convex optimization problem, a term which expresses a prior on the sparsity of the solution constrained by some data fidelity, is given by

$$\min_x \|x\|_1 \text{ s.t. } \|Ax - y\|_2 \leq \epsilon$$

where y are the measurements and A is the measurement matrix.

The number of necessary measurements m is computed with respect to the signal size n and the sparsity level S in order to very often perform a perfect reconstruction. See *[CR07]* for details.

```
>>> n = 5000
>>> S = 100
>>> import numpy as np
>>> m = int(np.ceil(S * np.log(n)))
>>> print('Number of measurements: {}'.format(m))
Number of measurements: 852
>>> print('Compression ratio: {:3.2f}'.format(float(n) / m))
Compression ratio: 5.87
```

We generate a random measurement matrix A:

```
>>> np.random.seed(1)  # Reproducible results.
>>> A = np.random.normal(size=(m, n))
```

Create the S sparse signal x:

```
>>> x = np.zeros(n)
>>> I = np.random.permutation(n)
```

```
>>> x[I[0:S]] = np.random.normal(size=S)
>>> x = x / np.linalg.norm(x)
```

Generate the measured signal *y*:

```
>>> y = np.dot(A, x)
```

The first objective function to minimize is defined by

$$f_1(x) = \|x\|_1$$

which can be expressed by the toolbox L1-norm function object. It can be instantiated as follows:

```
>>> from pyunlocbox import functions
>>> f1 = functions.norm_l1()
```

The second objective function to minimize is defined by

$$f_2(x) = \iota_C(x)$$

where $\iota_C()$ is the indicator function of the set $C = \{z \in \mathbb{R}^n \mid \|Az - y\|_2 \leq \epsilon\}$ which is zero if z is in the set and infinite otherwise. This function can be expressed by the toolbox L2-ball function object which can be instantiated as follows:

```
>>> f2 = functions.proj_b2(epsilon=1e-7, y=y, A=A, tight=False,
... nu=np.linalg.norm(A, ord=2)**2)
```

Now that the two function objects to minimize (the L1-norm and the L2-ball) are instantiated, we can instantiate the solver object. To solve this problem, we use the Douglas-Rachford splitting algorithm which is instantiated as follows:

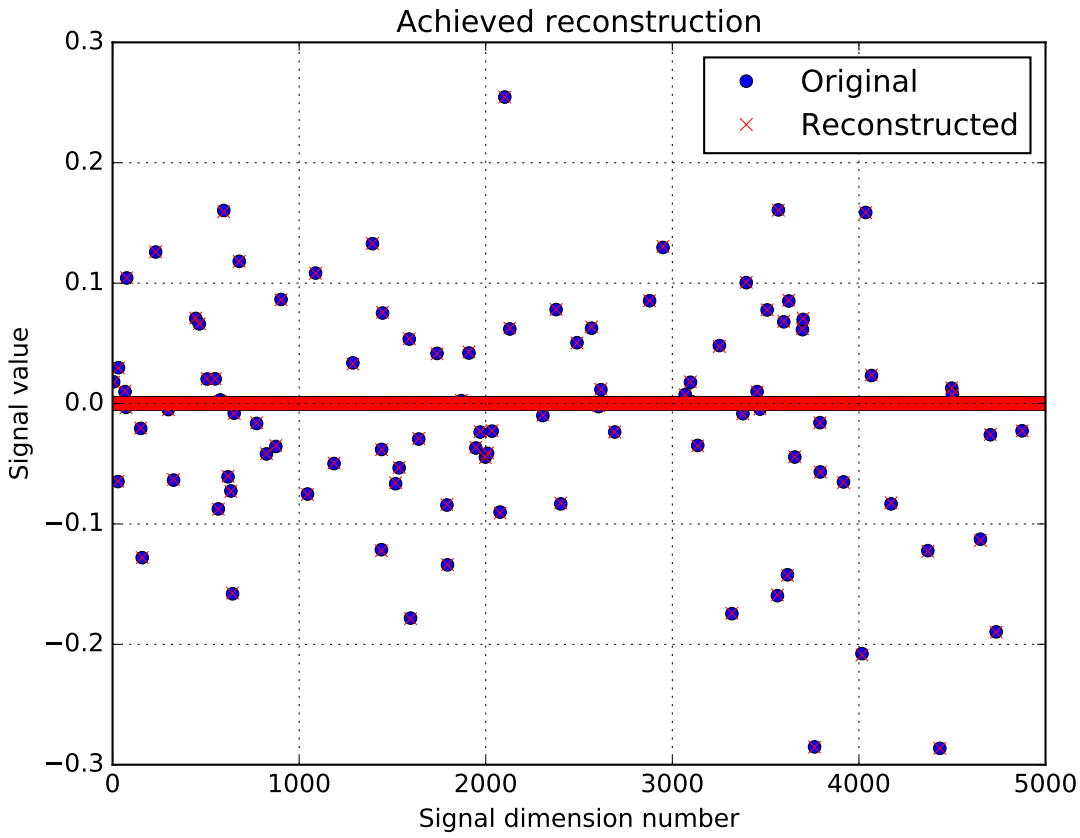
```
>>> from pyunlocbox import solvers
>>> solver = solvers.douglas_rachford(step=1e-2)
```

After the instantiations of the functions and solver objects, the setting of a starting point *x0*, the problem is solved by the toolbox solving function as follows:

```
>>> x0 = np.zeros(n)
>>> ret = solvers.solve([f1, f2], x0, solver, rtol=1e-4, maxit=300)
Solution found after 56 iterations:
    objective function f(sol) = 7.590460e+00
    stopping criterion: RTOL
```

Let's display the results:

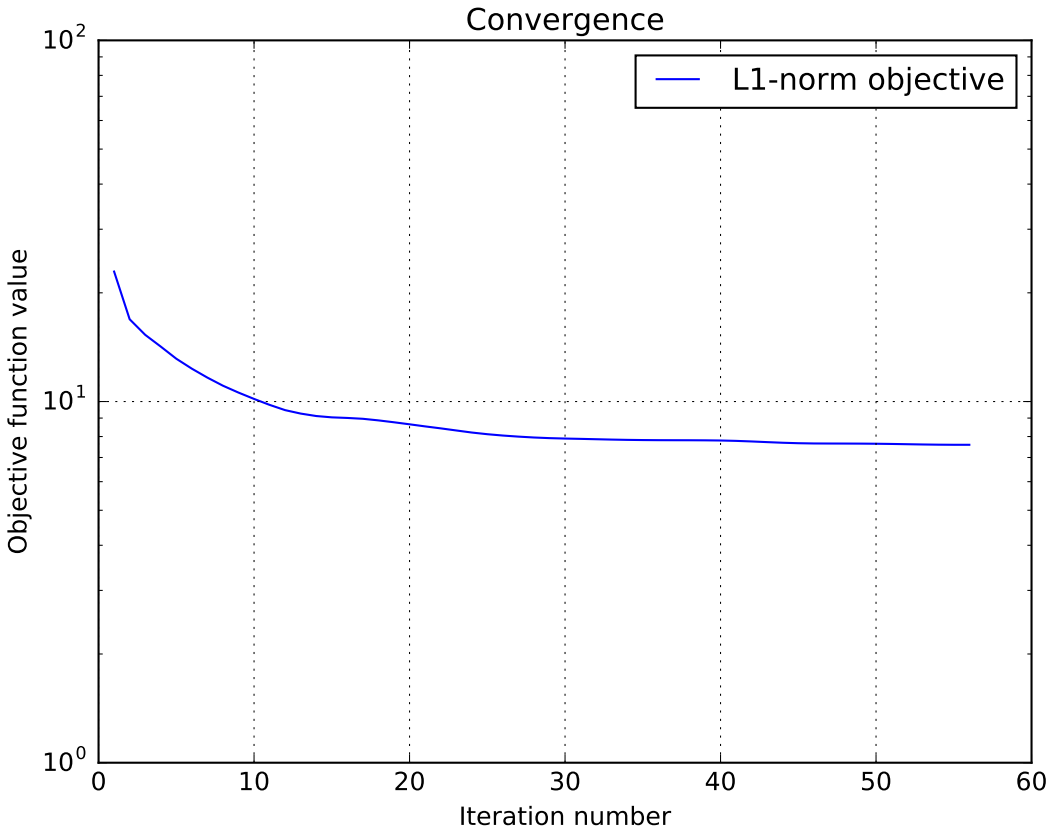
```
>>> import matplotlib.pyplot as plt
>>> _ = plt.figure()
>>> _ = plt.plot(x, 'o', label='Original')
>>> _ = plt.plot(ret['sol'], 'xr', label='Reconstructed')
>>> _ = plt.grid(True)
>>> _ = plt.title('Achieved reconstruction')
>>> _ = plt.legend(numpoints=1)
>>> _ = plt.xlabel('Signal dimension number')
>>> _ = plt.ylabel('Signal value')
```



The above figure shows a good reconstruction which is both sparse (thanks to the L1-norm objective) and close to the measurements (thanks to the L2-ball constraint).

Let's display the convergence of the objective function:

```
>>> objective = np.array(ret['objective'])
>>> _ = plt.figure()
>>> _ = plt.semilogy(objective[:, 0], label='L1-norm objective')
>>> _ = plt.grid(True)
>>> _ = plt.title('Convergence')
>>> _ = plt.legend()
>>> _ = plt.xlabel('Iteration number')
>>> _ = plt.ylabel('Objective function value')
```



3.1.4 Image reconstruction (Forward-Backward, Total Variation, L2-norm)

This tutorial presents an image reconstruction problem solved by the Forward-Backward splitting algorithm. The convex optimization problem is the sum of a data fidelity term and a regularization term which expresses a prior on the smoothness of the solution, given by

$$\min_x \tau \|g(x) - y\|_2^2 + \|x\|_{\mathrm{TV}}$$

where $\|\cdot\|_{\mathrm{TV}}$ denotes the total variation, y are the measurements, g is a masking operator and τ expresses the trade-off between the two terms.

Load an image and convert it to grayscale

```
>>> import matplotlib.image as mpimg
>>> import numpy as np
>>> try:
...     im_original = mpimg.imread('tutorials/lena.png')
... except:
...     im_original = mpimg.imread('doc/tutorials/lena.png')
>>> im_original = np.dot(im_original[..., :3], [0.299, 0.587, 0.144])
```

and generate a random masking matrix

```
>>> np.random.seed(14)  # Reproducible results.
>>> mask = np.random.uniform(size=im_original.shape)
>>> mask = mask > 0.85
```

which masks 85% of the pixels. The masked image is given by

```
>>> g = lambda x: mask * x
>>> im_masked = g(im_original)
```

The prior objective to minimize is defined by

$$f_1(x) = \|x\|_{\mathrm{TV}}$$

which can be expressed by the toolbox TV-norm function object, instantiated with

```
>>> from pyunlocbox import functions
>>> f1 = functions.norm_tv(maxit=50, dim=2)
```

The fidelity objective to minimize is defined by

$$f_2(x) = \tau\|g(x) - y\|_2^2$$

which can be expressed by the toolbox L2-norm function object, instantiated with

```
>>> tau = 100
>>> f2 = functions.norm_l2(y=im_masked, A=g, lambda_=tau)
```

Note: We set τ to a large value as we trust our measurements and want the solution to be close to them. For noisy measurements a lower value should be considered.

The step size for optimal convergence is $\frac{1}{\beta}$ where $\beta = 2\tau$ is the Lipschitz constant of the gradient of f_2 *[BT09a]*. The Forward-Backward splitting algorithm is instantiated with

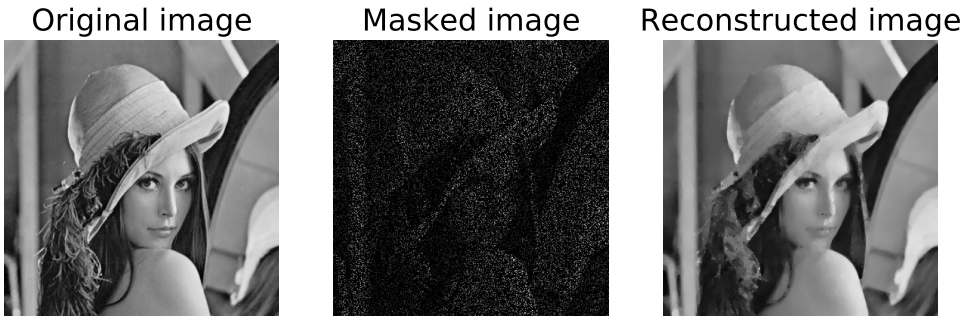
```
>>> from pyunlocbox import solvers
>>> solver = solvers.forward_backward(method='FISTA', step=0.5/tau)
```

and the problem solved with

```
>>> x0 = np.array(im_masked)  # Make a copy to preserve im_masked.
>>> ret = solvers.solve([f1, f2], x0, solver, maxit=100)
Solution found after 94 iterations:
    objective function f(sol) = 4.268147e+03
    stopping criterion: RTOL
```

Let's display the results:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure(figsize=(8, 2.5))
>>> ax1 = fig.add_subplot(1, 3, 1)
>>> _ = ax1.imshow(im_original, cmap='gray')
>>> _ = ax1.axis('off')
>>> _ = ax1.set_title('Original image')
>>> ax2 = fig.add_subplot(1, 3, 2)
>>> _ = ax2.imshow(im_masked, cmap='gray')
>>> _ = ax2.axis('off')
>>> _ = ax2.set_title('Masked image')
>>> ax3 = fig.add_subplot(1, 3, 3)
>>> _ = ax3.imshow(ret['sol'], cmap='gray')
>>> _ = ax3.axis('off')
>>> _ = ax3.set_title('Reconstructed image')
```



The above figure shows a good reconstruction which is both smooth (the TV prior) and close to the measurements (the L2 fidelity).

3.1.5 Image denoising (Douglas-Rachford, Total Variation, L2-norm)

This tutorial presents an image denoising problem solved by the Douglas-Rachford splitting algorithm. The convex optimization problem, a term which expresses a prior on the smoothness of the solution constrained by some data fidelity, is given by

$$\min_x \|x\|_{\mathrm{TV}} \text{ s.t. } \|x-y\|_2 \leq \epsilon$$

where $\|\cdot\|_{\mathrm{TV}}$ denotes the total variation, y are the measurements and ϵ expresses the noise level.

Create a white circle on a black background

```
>>> import numpy as np
>>> N = 650
>>> im_original = np.resize(np.linspace(-1, 1, N), (N, N))
>>> im_original = np.sqrt(im_original**2 + im_original.T**2)
>>> im_original = im_original < 0.7
```

and add some random Gaussian noise

```
>>> sigma = 0.5  # Variance of 0.25.
>>> np.random.seed(7)  # Reproducible results.
>>> im_noisy = im_original + sigma * np.random.normal(size=im_original.shape)
```

The prior objective function to minimize is defined by

$$f_1(x) = \|x\|_{\mathrm{TV}}$$

which can be expressed by the toolbox TV-norm function object, instantiated with

```
>>> from pyunlocbox import functions
>>> f1 = functions.norm_tv(maxit=50, dim=2)
```

The fidelity constraint expressed as an objective function to minimize is defined by

$$f_2(x) = \iota_S(x)$$

where $\iota_S()$ is the indicator function of the set $S = \{z \in \mathbb{R}^n \mid \|z-y\|_2 \leq \epsilon\}$ which is zero if z is in the set and infinite otherwise. This function can be expressed by the toolbox L2-ball function, instantiated with

```
>>> y = np.reshape(im_noisy, -1)  # Reshape the 2D image as a 1D vector.
>>> epsilon = N * sigma           # Variance multiplied by N^2.
>>> f = functions.proj_b2(y=y, epsilon=epsilon)
>>> f2 = functions.func()
>>> f2._eval = lambda x: 0        # Indicator functions evaluate to zero.
>>> def prox(x, step):
...     return np.reshape(f.prox(np.reshape(x, -1), 0), im_noisy.shape)
>>> f2._prox = prox
```

Note: We defined a custom proximal operator which transforms the 2D image as a 1D vector because *pyunlocbox.functions.proj_b2* operates on the columns of x while *pyunlocbox.functions.norm_tv* needs a two-dimensional array to compute the 2D TV norm.

The Douglas-Rachford splitting algorithm is instantiated with

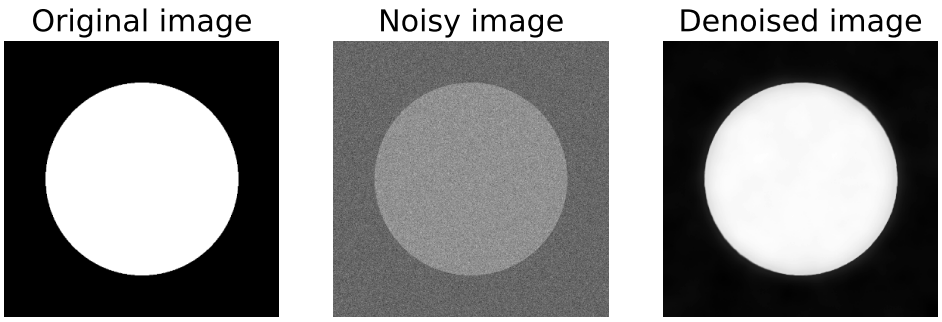
```
>>> from pyunlocbox import solvers
>>> solver = solvers.douglas_rachford(step=0.1)
```

and the problem solved with

```
>>> x0 = np.array(im_noisy)  # Make a copy to preserve y aka im_noisy.
>>> ret = solvers.solve([f1, f2], x0, solver)
Solution found after 25 iterations:
    objective function f(sol) = 2.080376e+03
    stopping criterion: RTOL
```

Let's display the results:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure(figsize=(8, 2.5))
>>> ax1 = fig.add_subplot(1, 3, 1)
>>> _ = ax1.imshow(im_original, cmap='gray')
>>> _ = ax1.axis('off')
>>> _ = ax1.set_title('Original image')
>>> ax2 = fig.add_subplot(1, 3, 2)
>>> _ = ax2.imshow(im_noisy, cmap='gray')
>>> _ = ax2.axis('off')
>>> _ = ax2.set_title('Noisy image')
>>> ax3 = fig.add_subplot(1, 3, 3)
>>> _ = ax3.imshow(ret['sol'], cmap='gray')
>>> _ = ax3.axis('off')
>>> _ = ax3.set_title('Denoised image')
```



The above figure shows a good reconstruction which is both smooth (the TV prior) and close to the measurements (the L2 fidelity constraint).

3.2 Reference guide

3.2.1 Toolbox overview

The toolbox is organized around two classes hierarchies: the functions and the solvers. Instantiated functions represent convex functions to optimize. Instantiated solvers represent solving algorithms. The *pyunlocbox.solvers.solve()* solving function takes as parameters a solver object and some function objects to actually solve the optimization problem. See this function's documentation for a typical usage example.

The *pyunlocbox* package is divided into the following modules :

- *pyunlocbox.solvers*: problem solvers, implement the solvers class hierarchy and the solving function
- *pyunlocbox.functions*: functions to be passed to the solvers, implement the functions class hierarchy

3.2.2 Functions module

Function objects

Interface

class pyunlocbox.functions.**func**(*y=0*, *A=None*, *At=None*, *tight=True*, *nu=1*, *tol=0.001*, *maxit=200*, ***kwargs*)

Bases: object

This class defines the function object interface.

It is intended to be a base class for standard functions which will implement the required methods. It can also be instantiated by user code and dynamically modified for rapid testing. The instanced objects are meant to be passed to the *pyunlocbox.solvers.solve()* solving function.

Parameters **y** : array_like, optional

Measurements. Default is 0.

A : function or ndarray, optional

The forward operator. Default is the identity, $A(x) = x$. If *A* is an ndarray, it will be converted to the operator form.

At : function or ndarray, optional

The adjoint operator. If *At* is an ndarray, it will be converted to the operator form. If *A* is an ndarray, default is the transpose of *A*. If *A* is a function, default is *A*, $At(x) = A(x)$.

tight : bool, optional

True if *A* is a tight frame, False otherwise. Default is True.

nu : float, optional

Bound on the norm of the operator *A*, i.e. $\|A(x)\|^2 \leq \nu\|x\|^2$. Default is 1.

tol : float, optional

The tolerance stopping criterion. The exact definition depends on the function object, please see the documentation of the considered function. Default is 1e-3.

maxit : int, optional

The maximum number of iterations. Default is 200.

Examples

Let's define a parabola as an example of the manual implementation of a function object :

```
>>> import pyunlocbox
>>> f = pyunlocbox.functions.func()
>>> f._eval = lambda x: x**2
>>> f._grad = lambda x: 2*x
>>> x = [1, 2, 3, 4]
>>> f.eval(x)
array([ 1,  4,  9, 16])
>>> f.grad(x)
array([2, 4, 6, 8])
```

```
>>> f.cap(x)
['EVAL', 'GRAD']
```

cap (*x*)

Test the capabilities of the function object.

Parameters **x** : array_like

The evaluation point. Not really needed, but this function calls the methods of the object to test if they can properly execute without raising an exception. Therefore it needs some evaluation point with a consistent size.

Returns **cap** : list of string

A list of capabilities ('EVAL', 'GRAD', 'PROX').

eval (*x*)

Function evaluation.

Parameters **x** : array_like

The evaluation point. If *x* is a matrix, the function gets evaluated for each column, as if it was a set of independent problems. Some functions, like the nuclear norm, are only defined on matrices.

Returns **z** : float

The objective function evaluated at *x*. If *x* is a matrix, the sum of the objectives is returned.

Notes

This method is required by the *pyunlocbox.solvers.solve()* solving function to evaluate the objective function. Each function class should therefore define it.

grad (*x*)

Function gradient.

Parameters **x** : array_like

The evaluation point. If *x* is a matrix, the function gets evaluated for each column, as if it was a set of independent problems. Some functions, like the nuclear norm, are only defined on matrices.

Returns **z** : ndarray

The objective function gradient evaluated for each column of *x*.

Notes

This method is required by some solvers.

prox (*x*, *T*)

Function proximal operator.

Parameters **x** : array_like

The evaluation point. If *x* is a matrix, the function gets evaluated for each column, as if it was a set of independent problems. Some functions, like the nuclear norm, are only defined on matrices.

T : float

The regularization parameter.

Returns **z** : ndarray

The proximal operator evaluated for each column of x.

Notes

This method is required by some solvers.

The proximal operator is defined by $\text{prox}_{\gamma f}(x) = \arg\min_z \frac{1}{2}\|x - z\|_2^2 + \gamma f(z)$

Dummy function

class pyunlocbox.functions.**dummy**(***kwargs*)

Bases: *pyunlocbox.functions.func*

Dummy function object.

This can be used as a second function object when there is only one function to minimize. It always evaluates as 0.

Examples

```
>>> import pyunlocbox
>>> f = pyunlocbox.functions.dummy()
>>> x = [1, 2, 3, 4]
>>> f.eval(x)
0
>>> f.prox(x, 1)
array([1, 2, 3, 4])
>>> f.grad(x)
array([ 0.,  0.,  0.,  0.])
```

Norm operators class hierarchy

Base class

class pyunlocbox.functions.**norm**(*lambda_=1*, *w=1*, ***kwargs*)

Bases: *pyunlocbox.functions.func*

Base class which defines the attributes of the *norm* objects.

See generic attributes descriptions of the *pyunlocbox.functions.func* base class.

Parameters **lambda_** : float, optional

Regularization parameter λ. Default is 1.

w : array_like, optional

Weights for a weighted norm. Default is 1.

L1-norm

class pyunlocbox.functions.**norm_l1**(***kwargs*)

Bases: *pyunlocbox.functions.norm*

L1-norm function object.

See generic attributes descriptions of the *pyunlocbox.functions.norm* base class. Note that the constructor takes keyword-only parameters.

Notes

- The L1-norm of the vector x is given by $\lambda \|w \cdot (A(x) - y)\|_1$.
- The L1-norm proximal operator evaluated at x is given by $\arg\min_z \frac{1}{2}\|x - z\|_2^2 + \gamma \|w \cdot (A(z) - y)\|_1$ where $\gamma = \lambda \cdot T$. This is simply a soft thresholding.

Examples

```
>>> import pyunlocbox
>>> f = pyunlocbox.functions.norm_l1()
>>> f.eval([1, 2, 3, 4])
10
>>> f.prox([1, 2, 3, 4], 1)
array([0, 1, 2, 3])
```

L2-norm

class pyunlocbox.functions.**norm_l2**(***kwargs*)

Bases: *pyunlocbox.functions.norm*

L2-norm function object.

See generic attributes descriptions of the *pyunlocbox.functions.norm* base class. Note that the constructor takes keyword-only parameters.

Notes

- The squared L2-norm of the vector x is given by $\lambda \|w \cdot (A(x) - y)\|_2^2$.
- The squared L2-norm proximal operator evaluated at x is given by $\arg\min_z \frac{1}{2}\|x-z\|_2^2+\gamma\|w\cdot(A(z)-y)\|_2^2$ where $\gamma = \lambda \cdot T$.
- The squared L2-norm gradient evaluated at x is given by $2\lambda \cdot At(w \cdot (A(x) - y))$.

Examples

```
>>> import pyunlocbox
>>> f = pyunlocbox.functions.norm_l2()
>>> x = [1, 2, 3, 4]
>>> f.eval(x)
30
>>> f.prox(x, 1)
array([ 0.33333333,  0.66666667,  1.        ,  1.33333333])
>>> f.grad(x)
array([2, 4, 6, 8])
```

Nuclear-norm

class pyunlocbox.functions.**norm_nuclear**(***kwargs*)

Bases: *pyunlocbox.functions.norm*

Nuclear-norm function object.

See generic attributes descriptions of the *pyunlocbox.functions.norm* base class. Note that the constructor takes keyword-only parameters.

Notes

- The nuclear-norm of the matrix x is given by $\lambda\|x\|_* = \lambda \operatorname{trace}(\sqrt{x^* x}) = \lambda \sum_{i=1}^{N} |e_i|$ where *e_i* are the eigenvalues of x.
- The nuclear-norm proximal operator evaluated at x is given by $\arg\min_z \frac{1}{2}\|x - z\|_2^2 + \gamma\|x\|_*$ where $\gamma = \lambda \cdot T$, which is a soft-thresholding of the eigenvalues.

Examples

```
>>> import pyunlocbox
>>> f = pyunlocbox.functions.norm_nuclear()
>>> f.eval([[1, 2],[2, 3]])
4.47213595...
>>> f.prox([[1, 2],[2, 3]], 1)
array([[ 0.89442719,  1.4472136 ],
       [ 1.4472136 ,  2.34164079]])
```

TV-norm

class pyunlocbox.functions.**norm_tv**(*dim=2*, *verbosity='LOW'*, ***kwargs*)

Bases: *pyunlocbox.functions.norm*

TV Norm function object.

See generic attributes descriptions of the *pyunlocbox.functions.norm* base class. Note that the constructor takes keyword-only parameters.

Notes

TODO

See *[BT09b]* for details about the algorithm.

Examples

```
>>> import pyunlocbox
>>> import numpy as np
>>> f = pyunlocbox.functions.norm_tv()
>>> x = np.arange(0, 16)
>>> x = x.reshape(4, 4)
>>> f.eval(x)
    norm_tv evaluation: 5.210795e+01
52.10795063...
```

Projection operators class hierarchy

Base class

class pyunlocbox.functions.**proj**(*epsilon=1*, *method='FISTA'*, ***kwargs*)

Bases: *pyunlocbox.functions.func*

Base class which defines the attributes of the *proj* objects.

See generic attributes descriptions of the *pyunlocbox.functions.func* base class.

Parameters **epsilon** : float, optional

The radius of the ball. Default is 1.

method : {'FISTA', 'ISTA'}, optional

The method used to solve the problem. It can be 'FISTA' or 'ISTA'. Default is 'FISTA'.

Notes

•All indicator functions (projections) evaluate to zero by definition.

L2-ball

class pyunlocbox.functions.**proj_b2**(***kwargs*)

Bases: *pyunlocbox.functions.proj*

L2-ball function object.

This function is the indicator function $i_S(z)$ of the set S which is zero if z is in the set and infinite otherwise. The set S is defined by $\left\{z \in \mathbb{R}^N \mid \|A(z) - y\|_2 \leq \epsilon\right\}$.

See generic attributes descriptions of the *pyunlocbox.functions.proj* base class. Note that the constructor takes keyword-only parameters.

Notes

•The *tol* parameter is defined as the tolerance for the projection on the L2-ball. The algorithm stops if $\frac{\epsilon}{1-tol} \leq \|y - A(z)\|_2 \leq \frac{\epsilon}{1+tol}$.

•The evaluation of this function is zero.

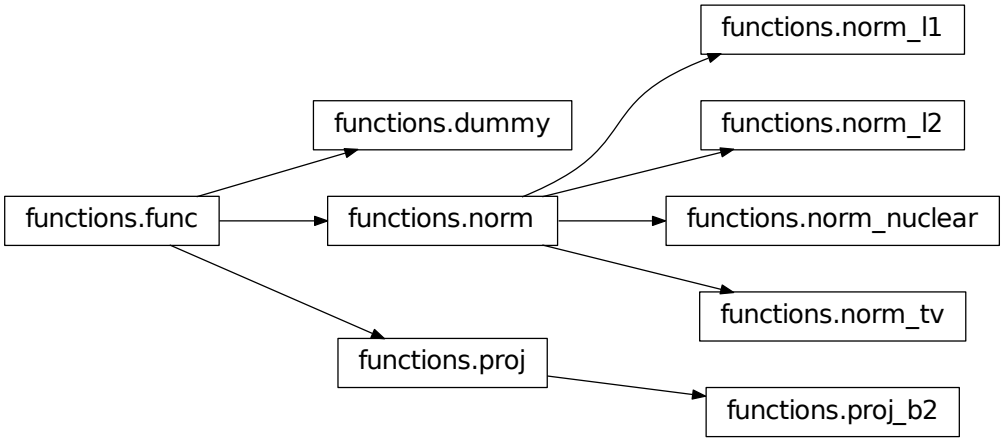
•The L2-ball proximal operator evaluated at x is given by $\arg\min_z \frac{1}{2}\|x - z\|_2^2 + i_S(z)$ which has an identical solution as $\arg\min_z \|x - z\|_2^2$ such that $\|A(z) - y\|_2 \leq \epsilon$. It is thus a projection of the vector x onto an L2-ball of diameter *epsilon*.

Examples

```
>>> import pyunlocbox
>>> f = pyunlocbox.functions.proj_b2(y=[1, 1])
>>> x = [3, 3]
>>> f.eval(x)
0
>>> f.prox(x, 0)
array([ 1.70710678,  1.70710678])
```

This module implements function objects which are then passed to solvers. The *func* base class defines the interface whereas specialised classes who inherit from it implement the methods. These classes include :

- *dummy*: A dummy function object which returns 0 for the `_eval()`, `_prox()` and `_grad()` methods.
- *norm*: Norm operators base class.
  - *norm_l1*: L1-norm who implements the `_eval()` and `_prox()` methods.
  - *norm_l2*: L2-norm who implements the `_eval()`, `_prox()` and `_grad()` methods.
  - *norm_nuclear*: nuclear-norm who implements the `_eval()` and `_prox()` methods.
  - *norm_tv*: TV-norm who implements the `_eval()` and `_prox()` methods.
- *proj*: Projection operators base class.
  - *proj_b2*: Projection on the L2-ball who implements the `_eval()` and `_prox()` methods.



3.2.3 Solvers module

Solving function

`pyunlocbox.solvers.`**`solve`**(*functions*, *x0*, *solver=None*, *atol=None*, *dtol=None*, *rtol=0.001*, *xtol=None*, *maxit=200*, *verbosity='LOW'*)

Solve an optimization problem whose objective function is the sum of some convex functions.

This function minimizes the objective function $f(x) = \sum_{k=0}^{k=K} f_k(x)$, i.e. solves $\arg\min_x f(x)$ for $x \in \mathbb{R}^{n \times N}$ where n is the dimensionality of the data and N the number of independent problems. It returns a dictionary with the found solution and some informations about the algorithm execution.

Parameters **functions** : list of objects

A list of convex functions to minimize. These are objects who must implement the *`pyunlocbox.functions.func.eval()`* method. The *`pyunlocbox.functions.func.grad()`* and / or *`pyunlocbox.functions.func.prox()`* methods are required by some solvers. Note also that some solvers can only handle two convex functions while others may handle more. Please refer to the documentation of the considered solver.

x0 : array_like

Starting point of the algorithm, $x_0 \in \mathbb{R}^{n \times N}$. Note that if you pass a numpy array it will be modified in place during execution to save memory. It will then contain the solution. Be careful to pass data of the type (int, float32, float64) you want your computations to use.

solver : solver class instance, optional

The solver algorithm. It is an object who must inherit from *`pyunlocbox.solvers.solver`* and implement the `_pre()`, `_algo()` and `_post()` methods. If no solver object are provided, a standard one will be chosen given the number of convex function objects and their implemented methods.

atol : float, optional

The absolute tolerance stopping criterion. The algorithm stops when $f(x^t) < atol$ where $f(x^t)$ is the objective function at iteration t. Default is None.

dtol : float, optional

Stop when the objective function is stable enough, i.e. when $\left|f(x^t) - f(x^{t-1})\right| < dtol$. Default is None.

rtol : float, optional

The relative tolerance stopping criterion. The algorithm stops when $\left|\frac{f(x^t)-f(x^{t-1})}{f(x^t)}\right| < rtol$. Default is 10^{-3}.

xtol : float, optional

Stop when the variable is stable enough, i.e. when $\frac{\|x^t - x^{t-1}\|_2}{\sqrt{nN}} < xtol$. Note that additional memory will be used to store x^{t-1}. Default is None.

maxit : int, optional

The maximum number of iterations. Default is 200.

verbosity : {'NONE', 'LOW', 'HIGH', 'ALL'}, optional

The log level : `'NONE'` for no log, `'LOW'` for resume at convergence, `'HIGH'` for info at all solving steps, `'ALL'` for all possible outputs, including at each steps of the proximal operators computation. Default is `'LOW'`.

Returns **sol** : ndarray

The problem solution.

solver : str

The used solver.

crit : {'ATOL', 'DTOL', 'RTOL', 'XTOL', 'MAXIT'}

The used stopping criterion. See above for definitions.

niter : int

The number of iterations.

time : float

The execution time in seconds.

objective : ndarray

The successive evaluations of the objective function at each iteration.

Examples

```
>>> import pyunlocbox
>>> import numpy as np
```

Define a problem:

```
>>> y = [4, 5, 6, 7]
>>> f = pyunlocbox.functions.norm_l2(y=y)
```

Solve it:

```
>>> x0 = np.zeros(len(y))
>>> ret = pyunlocbox.solvers.solve([f], x0, atol=1e-2, verbosity='ALL')
INFO: Dummy objective function added.
INFO: Selected solver: forward_backward
    norm_l2 evaluation: 1.260000e+02
    dummy evaluation: 0.000000e+00
INFO: Forward-backward method: FISTA
```

```
    Iteration 1 of forward_backward:
        norm_l2 evaluation: 1.400000e+01
        dummy evaluation: 0.000000e+00
        objective = 1.40e+01
    Iteration 2 of forward_backward:
        norm_l2 evaluation: 1.555556e+00
        dummy evaluation: 0.000000e+00
        objective = 1.56e+00
    Iteration 3 of forward_backward:
        norm_l2 evaluation: 3.293044e-02
        dummy evaluation: 0.000000e+00
        objective = 3.29e-02
    Iteration 4 of forward_backward:
        norm_l2 evaluation: 8.780588e-03
        dummy evaluation: 0.000000e+00
        objective = 8.78e-03
    Solution found after 4 iterations:
        objective function f(sol) = 8.780588e-03
        stopping criterion: ATOL
```

Verify the stopping criterion (should be smaller than atol=1e-2):

```
    >>> np.linalg.norm(ret['sol'] - y)**2
    0.00878058...
```

Show the solution (should be close to y w.r.t. the L2-norm measure):

```
    >>> ret['sol']
    array([ 4.03339154,  5.04173943,  6.05008732,  7.0584352 ])
```

Show the used solver:

```
    >>> ret['solver']
    'forward_backward'
```

Show some information about the convergence:

```
    >>> ret['crit']
    'ATOL'
    >>> ret['niter']
    4
    >>> ret['time']
    0.0012578964233398438
    >>> ret['objective']
    [[126.0, 0], [13.99999999..., 0], [1.55555555..., 0],
    [0.03293043..., 0], [0.00878058..., 0]]
```

Solver class hierarchy

Solver object interface

class pyunlocbox.solvers.**solver**(*step=1*, *post_step=None*, *post_sol=None*)

Bases: object

Defines the solver object interface.

This class defines the interface of a solver object intended to be passed to the *pyunlocbox.solvers.solve()* solving function. It is intended to be a base class for standard solvers which will implement the required methods. It can also be instantiated by user code and dynamically modified for rapid testing. This class also defines the generic attributes of all solver objects.

Parameters **step** : float

The gradient-descent step-size. This parameter is bounded by 0 and $\frac{2}{\beta}$ where β is the Lipschitz constant of the gradient of the smooth function (or a sum of smooth functions). Default is 1.

post_step : function

User defined function to post-process the step size. This function is called every iteration and permits the user to alter the solver algorithm. The user may start with a high step size and progressively lower it while the algorithm runs to accelerate the convergence. The function parameters are the following : *step* (current step size), *sol* (current problem solution), *objective* (list of successive evaluations of the objective function), *niter* (current iteration number). The function should return a new value for *step*. Default is to return an unchanged value.

post_sol : function

User defined function to post-process the problem solution. This function is called every iteration and permits the user to alter the solver algorithm. Same parameter as `post_step()`. Default is to return an unchanged value.

algo(*objective*, *niter*)
Call the solver iterative algorithm while allowing the user to alter it. This makes it possible to dynamically change the *step* step size while the algorithm is running. See parameters documentation in *pyunlocbox.solvers.solve()* documentation.

post()
Solver specific post-processing. Mainly used to delete references added during initialization so that the garbage collector can free the memory. See parameters documentation in *pyunlocbox.solvers.solve()* documentation.

pre(*functions*, *x0*)
Solver specific initialization. See parameters documentation in *pyunlocbox.solvers.solve()* documentation.

Forward-backward proximal splitting algorithm

class pyunlocbox.solvers.**forward_backward**(*method='FISTA'*, *lambda_=1*, **args*, ***kwargs*)

Bases: *pyunlocbox.solvers.solver*

Forward-backward proximal splitting algorithm.

This algorithm solves convex optimization problems composed of the sum of a smooth and a non-smooth function.

See generic attributes descriptions of the *pyunlocbox.solvers.solver* base class.

Parameters **method** : {'FISTA', 'ISTA'}, optional

The method used to solve the problem. It can be 'FISTA' or 'ISTA'. Note that while FISTA is much more time efficient, it is less memory efficient. Default is 'FISTA'.

lambda_ : float, optional

The update term weight for ISTA. It should be between 0 and 1. Default is 1.

Notes

This algorithm requires one function to implement the *pyunlocbox.functions.func.prox()* method and the other one to implement the *pyunlocbox.functions.func.grad()* method.

See *[BT09a]* for details about the algorithm.

Examples

```
>>> from pyunlocbox import functions, solvers
>>> import numpy as np
>>> y = [4, 5, 6, 7]
>>> x0 = np.zeros(len(y))
>>> f1 = functions.norm_l2(y=y)
>>> f2 = functions.dummy()
>>> solver = solvers.forward_backward(method='FISTA', lambda_=1, step=0.5)
>>> ret = solvers.solve([f1, f2], x0, solver, atol=1e-5)
Solution found after 12 iterations:
    objective function f(sol) = 4.135992e-06
    stopping criterion: ATOL
>>> ret['sol']
array([ 3.99927529,  4.99909411,  5.99891293,  6.99873176])
```

Douglas-Rachford proximal splitting algorithm

class pyunlocbox.solvers.**douglas_rachford**(*lambda_=1*, **args*, ***kwargs*)

Bases: *pyunlocbox.solvers.solver*

Douglas-Rachford proximal splitting algorithm.

This algorithm solves convex optimization problems composed of the sum of two non-smooth (or smooth) functions.

See generic attributes descriptions of the *pyunlocbox.solvers.solver* base class.

Parameters **lambda_** : float, optional

The update term weight. It should be between 0 and 1. Default is 1.

Notes

This algorithm requires the two functions to implement the *pyunlocbox.functions.func.prox()* method.

See *[CP07]* for details about the algorithm.

Examples

```
>>> from pyunlocbox import functions, solvers
>>> import numpy as np
>>> y = [4, 5, 6, 7]
>>> x0 = np.zeros(len(y))
>>> f1 = functions.norm_l2(y=y)
>>> f2 = functions.dummy()
>>> solver = solvers.douglas_rachford(lambda_=1, step=1)
>>> ret = solvers.solve([f1, f2], x0, solver, atol=1e-5)
Solution found after 8 iterations:
    objective function f(sol) = 2.927052e-06
    stopping criterion: ATOL
>>> ret['sol']
array([ 3.99939034,  4.99923792,  5.99908551,  6.99893309])
```

Generalized forward-backward proximal splitting algorithm

class pyunlocbox.solvers.**generalized_forward_backward**(*lambda_=1*, **args*, ***kwargs*)

Bases: *pyunlocbox.solvers.solver*

Generalized forward-backward proximal splitting algorithm.

This algorithm solves convex optimization problems composed of the sum of any number of non-smooth (or smooth) functions.

See generic attributes descriptions of the *pyunlocbox.solvers.solver* base class.

Parameters **lambda_** : float, optional

A relaxation parameter bounded by 0 and 1. Default is 1.

Notes

This algorithm requires each function to either implement the *pyunlocbox.functions.func.prox()* method or the *pyunlocbox.functions.func.grad()* method.

See *[RFP13]* for details about the algorithm.

Examples

```
>>> from pyunlocbox import functions, solvers
>>> import numpy as np
>>> y = [0.01, 0.2, 8, 0.3, 0 , 0.03, 7]
>>> x0 = np.zeros(len(y))
>>> f1 = functions.norm_l2(y=y)
>>> f2 = functions.norm_l1()
>>> solver = solvers.generalized_forward_backward(lambda_=1, step=0.5)
>>> ret = solvers.solve([f1, f2], x0, solver)
Solution found after 2 iterations:
    objective function f(sol) = 1.463100e+01
    stopping criterion: RTOL
>>> ret['sol']
array([ 0. ,  0. ,  7.5,  0. ,  0. ,  0. ,  6.5])
```

Primal-dual algorithms

class pyunlocbox.solvers.**primal_dual**(*L=None*, *Lt=None*, *d0=None*, **args*, ***kwargs*)

Bases: *pyunlocbox.solvers.solver*

Parent class of all primal-dual algorithms.

See generic attributes descriptions of the *pyunlocbox.solvers.solver* base class.

Parameters **L** : function or ndarray, optional

The transformation L that maps from the primal variable space to the dual variable space. Default is the identity, $L(x) = x$. If *L* is an `ndarray`, it will be converted to the operator form.

Lt : function or ndarray, optional

The adjoint operator. If *Lt* is an `ndarray`, it will be converted to the operator form. If *L* is an `ndarray`, default is the transpose of *L*. If *L* is a function, default is *L*, $Lt(x) = L(x)$.

d0: ndarray, optional

Initialization of the dual variable.

Monotone+Lipschitz forward-backward-forward algorithm

class pyunlocbox.solvers.**mlfbf**(*L=None*, *Lt=None*, *d0=None*, **args*, ***kwargs*)

Bases: *pyunlocbox.solvers.primal_dual*

Monotone + Lipschitz Forward-Backward-Forward primal-dual algorithm.

This algorithm solves convex optimization problems with objective of the form $f(x)+g(Lx)+h(x)$, where f and g are proper, convex, lower-semicontinuous functions with easy-to-compute proximity operators, and h has Lipschitz-continuous gradient with constant β.

See generic attributes descriptions of the *pyunlocbox.solvers.primal_dual* base class.

Notes

The order of the functions matters: set f first on the list, g second, and h third.

This algorithm requires the first two functions to implement the *pyunlocbox.functions.func.prox()* method, and the third function to implement the *pyunlocbox.functions.func.grad()* method.

The step-size should be in the interval $\left]0, \frac{1}{\beta+\|L\|_2}\right[$.

See *[KP15]*, Algorithm 6, for details.

Examples

```
>>> from pyunlocbox import functions, solvers
>>> import numpy as np
>>> y = np.array([294, 390, 361])
>>> L = np.array([[5, 9, 3], [7, 8, 5], [4, 4, 9], [0, 1, 7]])
>>> x0 = np.zeros(len(y))
>>> f = functions.dummy()
>>> f._prox = lambda x, T: np.maximum(np.zeros(len(x)), x)
>>> g = functions.norm_l2(lambda_=0.5)
>>> h = functions.norm_l2(y=y, lambda_=0.5)
>>> max_step = 1/(1 + np.linalg.norm(L, 2))
>>> solver = solvers.mlfbf(L=L, step=max_step/2.)
>>> ret = solvers.solve([f, g, h], x0, solver, maxit=1000, rtol=0)
Solution found after 1000 iterations:
    objective function f(sol) = 1.833865e+05
    stopping criterion: MAXIT
>>> ret['sol']
array([ 1.,  1.,  1.])
```

Projection-based primal-dual algorithm

class pyunlocbox.solvers.**projection_based**(*lambda_=1*, **args*, ***kwargs*)

Bases: *pyunlocbox.solvers.primal_dual*

Projection-based primal-dual algorithm.

This algorithm solves convex optimization problems with objective of the form $f(x)+g(Lx)$, where f and g are proper, convex, lower-semicontinuous functions with easy-to-compute proximity operators.

See generic attributes descriptions of the *pyunlocbox.solvers.primal_dual* base class.

Parameters **lambda_** : float, optional

The update term weight. It should be between 0 and 2. Default is 1.

Notes

The order of the functions matters: set f first on the list, and g second.

This algorithm requires the two functions to implement the *pyunlocbox.functions.func.prox()* method.

The step-size should be in the interval $]0, \infty[$.

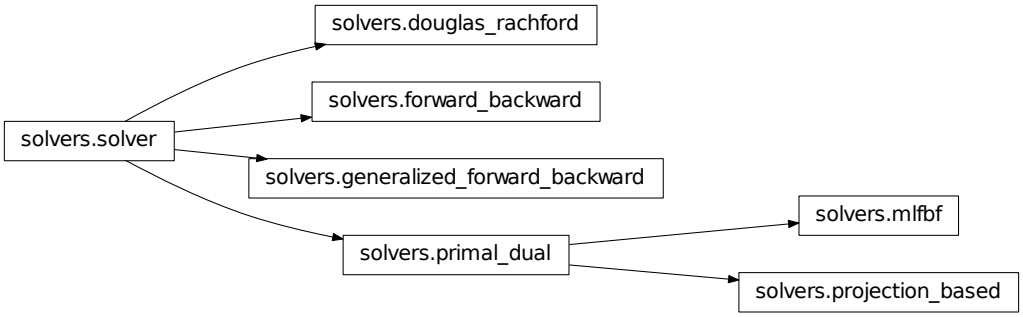
See *[KP15]*, Algorithm 7, for details.

Examples

```
>>> from pyunlocbox import functions, solvers
>>> import numpy as np
>>> y = np.array([294, 390, 361])
>>> L = np.array([[5, 9, 3], [7, 8, 5], [4, 4, 9], [0, 1, 7]])
>>> x0 = np.array([500, 1000, -400])
>>> f = functions.norm_l1(y=y)
>>> g = functions.norm_l1()
>>> solver = solvers.projection_based(L=L, step=1.)
>>> ret = solvers.solve([f, g], x0, solver, maxit=1000, rtol=None, xtol=.1)
Solution found after 996 iterations:
    objective function f(sol) = 1.045000e+03
    stopping criterion: XTOL
>>> ret['sol']
array([0, 0, 0])
```

This module implements solver objects who minimize an objective function. Call *solve()* to solve your convex optimization problem using your instantiated solver and functions objects. The *solver* base class defines the interface of all solver objects. The specialized solver objects inherit from it and implement the class methods. The following solvers are included :

- *forward_backward*: Forward-backward proximal splitting algorithm.
- *douglas_rachford*: Douglas-Rachford proximal splitting algorithm.
- *generalized_forward_backward*: Generalized Forward-Backward.
- *primal_dual*: Primal-dual algorithms.
  - *mlfbf*: Monotone+Lipschitz Forward-Backward-Forward primal-dual algorithm.
  - *projection_based*: Projection-based primal-dual algorithm.



3.2.4 Operators module

Gradient Operators

pyunlocbox.operators.**grad**(*x*, *dim=2*, ***kwargs*)

Returns the gradient of the array

Parameters **dim** : int

Dimension of the grad

wx : int

wy : int

wz : int

wt : int

Weights to apply on each axis

Returns **dx, dy, dz, dt** : ndarrays

Gradients following each axes, only the necessary ones are returned

Examples

```
>>> import pyunlocbox
>>> import numpy as np
>>> x = np.arange(16).reshape(4, 4)
>>> dx, dy = pyunlocbox.operators.grad(x)
```

Divergence Operators

pyunlocbox.operators.**div**(**args*, ***kwargs*)

Returns the divergence of the array

Parameters **dx** : array_like

dy : array_like

dz : array_like

dt : array_like

Arrays to operate on

Returns **x** : array_like

Divergence vector

Examples

```
>>> import pyunlocbox
>>> import numpy as np
>>> x = np.arange(16).reshape(4, 4)
>>> dx, dy = pyunlocbox.operators.grad(x)
>>> divx = pyunlocbox.operators.div(dx, dy)
```

This module implements operators functions :

- *grad()* Gradient function for up to 4 dimensions
- *div()* Divergence function for up to 4 dimensions

3.3 History

3.3.1 x.x.x (xxxx-xx-xx)

New Features:

- Monotone+Lipschitz Forward-Backward-Forward primal-dual algorithm
- Projection-based primal-dual algorithm
- L2-norm proximal operator supports non-tight frames

Bug fixes :

- prox_tv_2d has been fixed

Infrastructure :

- Continuous integration testing on Python 2.7, 3.3, 3.4 and 3.5
- Travis-ci: check style and build doc
- Removed tox config (too cumbersome to use on dev box)
- Monitor code coverage and report to coveralls.io

3.3.2 0.2.2 (2015-01-16)

New feature version. Still experimental.

New Features:

- norm_tv has been added with gradient, div, evaluation and prox.
- Module signals has been added.
- A demo for douglas rachford is also now present.

3.3.3 0.2.1 (2014-08-20)

Bug fix version. Still experimental.

Bug fixes :

- Avoid complex casting to real

- Do not stop iterating if the objective function stays at zero

3.3.4 0.2.0 (2014-08-04)

Second usable version, available on GitHub and released on PyPI. Still experimental.

New features :

- Douglas-Rachford splitting algorithm
- Projection on the L2-ball for tight and non tight frames
- Compressed sensing tutorial using L2-ball, L2-norm and Douglas-Rachford
- Automatic solver selection

Infrastructure :

- Unit tests for all functions and solvers
- Continuous integration testing on Python 2.6, 2.7, 3.2, 3.3 and 3.4

3.3.5 0.1.0 (2014-06-08)

First usable version, available on GitHub and released on PyPI. Still experimental.

Features :

- Forward-backward splitting algorithm
- L1-norm function (eval and prox)
- L2-norm function (eval, grad and prox)
- TV-norm function (eval, grad, div and prox)
- Least square problem tutorial using L2-norm and forward-backward
- Compressed sensing tutorial using L1-norm, L2-norm and forward-backward

Infrastructure :

- Sphinx generated documentation using Numpy style docstrings
- Documentation hosted on Read the Docs
- Code hosted on GitHub
- Package hosted on PyPI
- Code checked by flake8
- Docstring and tutorial examples checked by doctest (as a test suite)
- Unit tests for functions module (as a test suite)
- All test suites executed in Python 2.6, 2.7 and 3.2 virtualenvs by tox
- Distributed automatic testing on Travis CI continuous integration platform

3.4 References

[BT09a] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2(1):183–202, 2009.

[BT09b] Amir Beck and Marc Teboulle. Fast gradient-based algorithms for constrained total variation image denoising and deblurring problems. *Image Processing, IEEE Transactions on*, 18(11):2419–2434, 2009.

[CR07] Emmanuel Candes and Justin Romberg. Sparsity and incoherence in compressive sampling. *Inverse problems*, 23(3):969, 2007.

[CP07] Patrick L Combettes and Jean-Christophe Pesquet. A douglas–rachford splitting approach to nonsmooth convex variational signal recovery. *Selected Topics in Signal Processing, IEEE Journal of*, 1(4):564–574, 2007.

[KP15] Nikos Komodakis and Jean-Christophe Pesquet. Playing with duality. *IEEE Signal Processing Magazine*, pages 31–54, 2015.

[RFP13] Hugo Raguet, Jalal Fadili, and Gabriel Peyré. A generalized forward-backward splitting. *SIAM Journal on Imaging Sciences*, 6(3):1199–1226, 2013.

p