PyUNLocBoX documentation

Release 0.5.1

Michaël Defferrard, EPFL LTS2

Jul 04, 2017

Contents

The PyUNLocBoX is a convex optimization package based on proximal splitting methods and implemented in Python (a Matlab counterpart exists). It is a free software, distributed under the BSD license, and available on PyPI. The documentation is available online and development takes place on GitHub.

The package is designed to be easy to use while allowing any advanced tasks. It is not meant to be a black-box optimization tool. You'll have to carefully design your solver. In exchange you'll get full control of what the package does for you, without the pain of rewriting the proximity operators and the solvers and with the added benefit of tested algorithms. With this package, you can focus on your problem and the best way to solve it rather that the details of the algorithms. It comes with the following solvers:

- Gradient descent
- Forward-backward splitting algorithm (FISTA, ISTA)
- Douglas-Rachford splitting algorithm
- Generalized forward-backward
- Monotone+Lipschitz forward-backward-forward primal-dual algorithm
- Projection-based primal-dual algorithm

Moreover, the following acceleration schemes are included:

- FISTA acceleration scheme
- Backtracking based on a quadratic approximation of the objective
- Regularized nonlinear acceleration (RNA)

To compose your objective, you can either define your custom functions (which should implement an evaluation method and a gradient or proximity method) or use one of the followings:

- L1-norm
- L2-norm
- TV-norm
- Nuclear-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 9 iterations:
    objective function f(sol) = 6.714385e-08
    stopping criterion: ATOL
>>> ret['sol']
array([ 3.99990766,  4.99988458,  5.99986149,  6.99983841])
```

CHAPTER 1

Installation

The PyUnLocBox is available on PyPI:

```
$ pip install pyunlocbox
```

CHAPTER 2

Contributing

The development of this package takes place on GitHub. Issues and pull requests are welcome.

You can improve or add solvers, functions, and acceleration schemes in `pyunlocbox/solvers.py`, `pyunlocbox/functions.py`, and `pyunlocbox/acceleration.py`, along with their corresponding unit tests in `pyunlocbox/tests/test_*.py` (with reasonable coverage) and documentation in `doc/reference/*.rst`. If you have a nice example to demonstrate the use of the introduced functionality, please consider adding a tutorial in `doc/tutorials`.

Do not forget to update `README.rst` and `doc/history.rst` with e.g. new features or contributors. The version number needs to be updated in `setup.py` and `pyunlocbox/__init__.py`.

Please make sure that your changes pass the tests (enforced by CI) and check the generated coverage report at `htmlcov/index.html` to make sure the tests reasonably cover the changes you've introduced:

```
$ make lint
$ make test
$ make docall
```

CHAPTER 3

Authors

PyUNLocBoX was started in 2014 as an academic project for research purpose at the EPFL LTS2 laboratory.

Development lead :

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

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.

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
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: 2.963739e-01
    objective = 2.96e-01
Iteration 3 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 7.902529e-02
    objective = 7.90e-02
Iteration 4 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 5.752265e-02
    objective = 5.75e-02
Iteration 5 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 5.142032e-03
    objective = 5.14e-03
Iteration 6 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 1.553851e-04
    objective = 1.55e-04
Iteration 7 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 5.498523e-04
    objective = 5.50e-04
Iteration 8 of forward_backward:
```

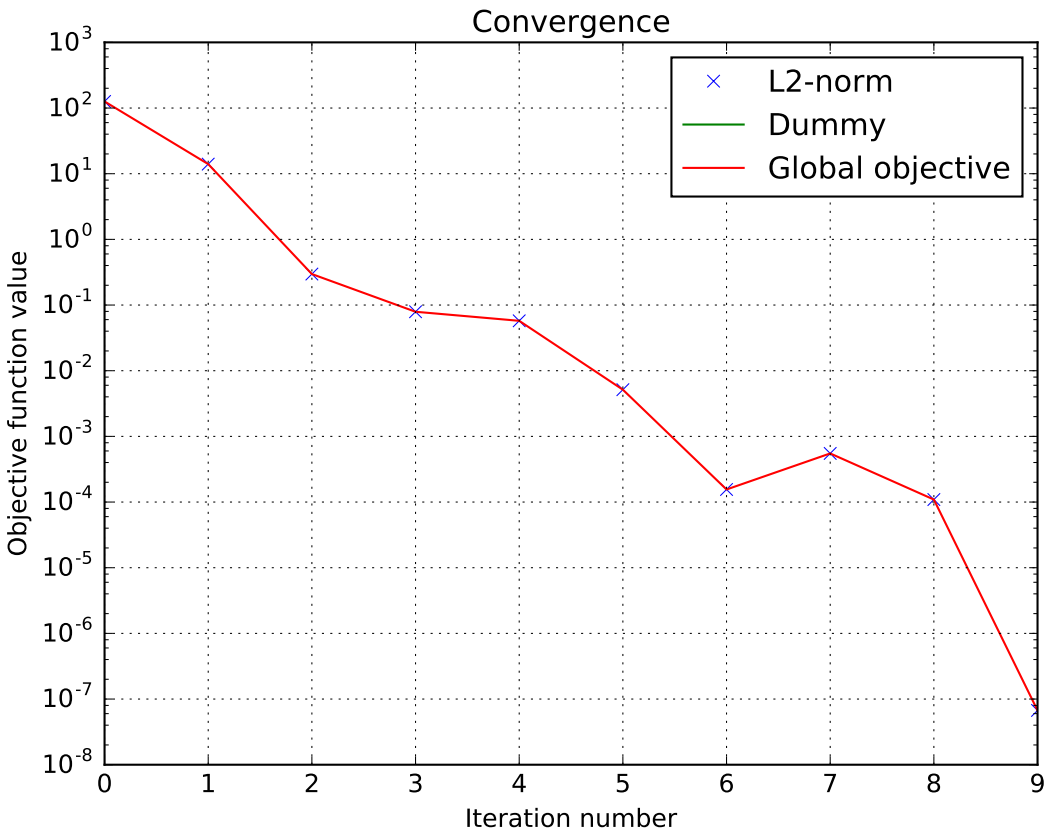
```
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 1.091372e-04
    objective = 1.09e-04
Iteration 9 of forward_backward:
    func evaluation: 0.000000e+00
    norm_l2 evaluation: 6.714385e-08
    objective = 6.71e-08
Solution found after 9 iterations:
    objective function f(sol) = 6.714385e-08
    stopping criterion: ATOL
```

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

```
>>> ret['sol']
array([ 3.99990766,  4.99988458,  5.99986149,  6.99983841])
```

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`).

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(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.

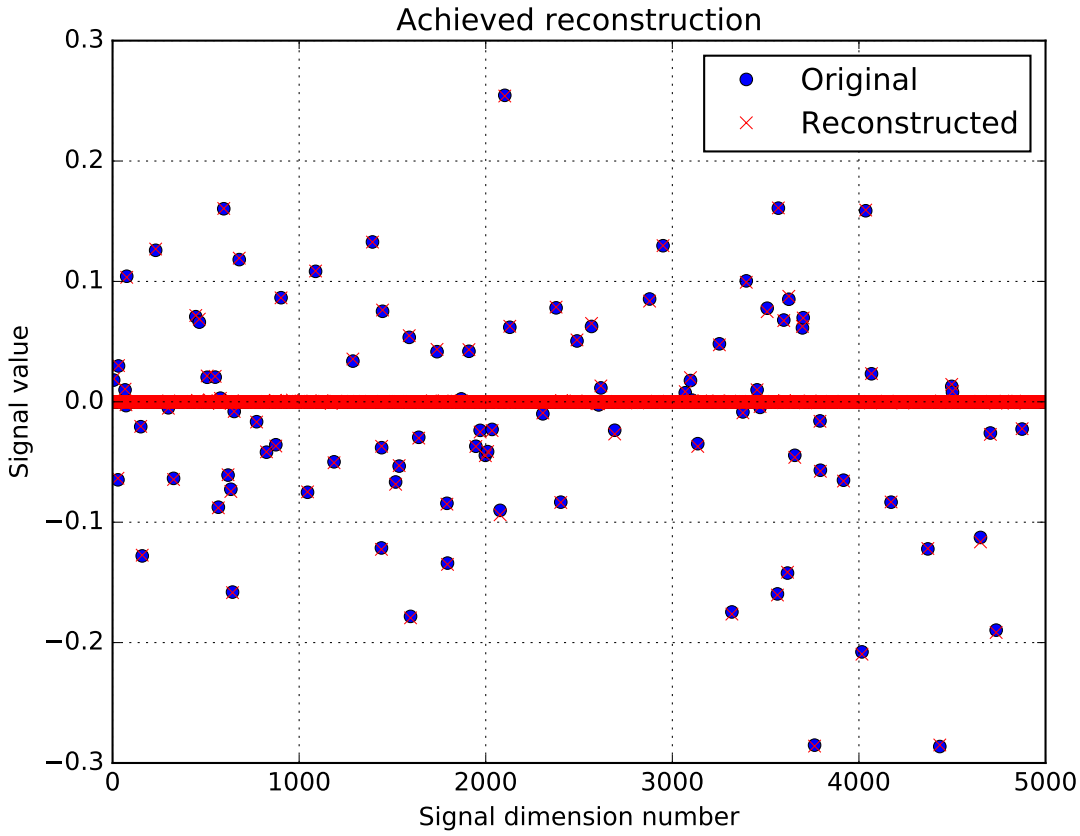
After the instantiations of the functions and solver objects, the setting of a starting point *x0*, the problem is solved by the toolbox solving function as follows:

```
>>> x0 = np.zeros(n)
>>> ret = solvers.solve([f1, f2], x0, solver, rtol=1e-4, maxit=300)
Solution found after 151 iterations:
    objective function f(sol) = 7.668167e+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')
```

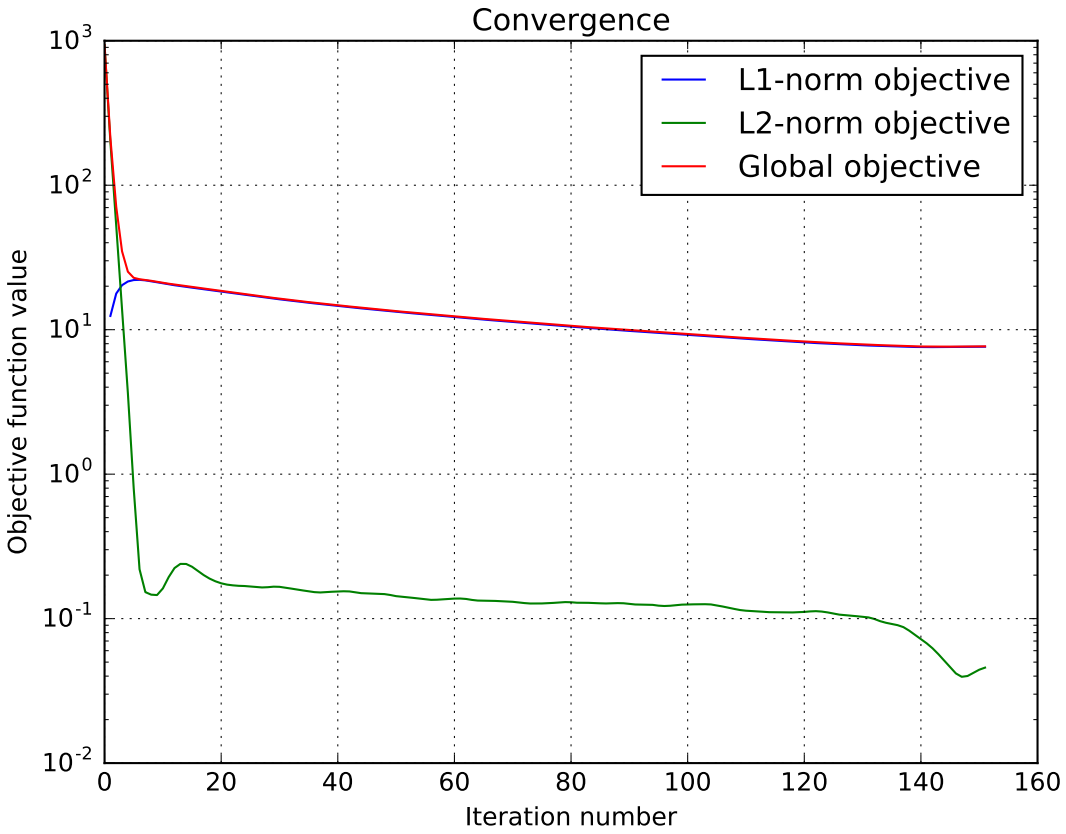


The above figure shows a good reconstruction which is both sparse (thanks to the L1-norm objective) and close to

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')
```



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 = 900
>>> S = 45
```

```
>>> import numpy as np
>>> m = int(np.ceil(S * np.log(n)))
>>> print('Number of measurements: {}'.format(m))
Number of measurements: 307
>>> print('Compression ratio: {:3.2f}'.format(float(n) / m))
Compression ratio: 2.93
```

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:

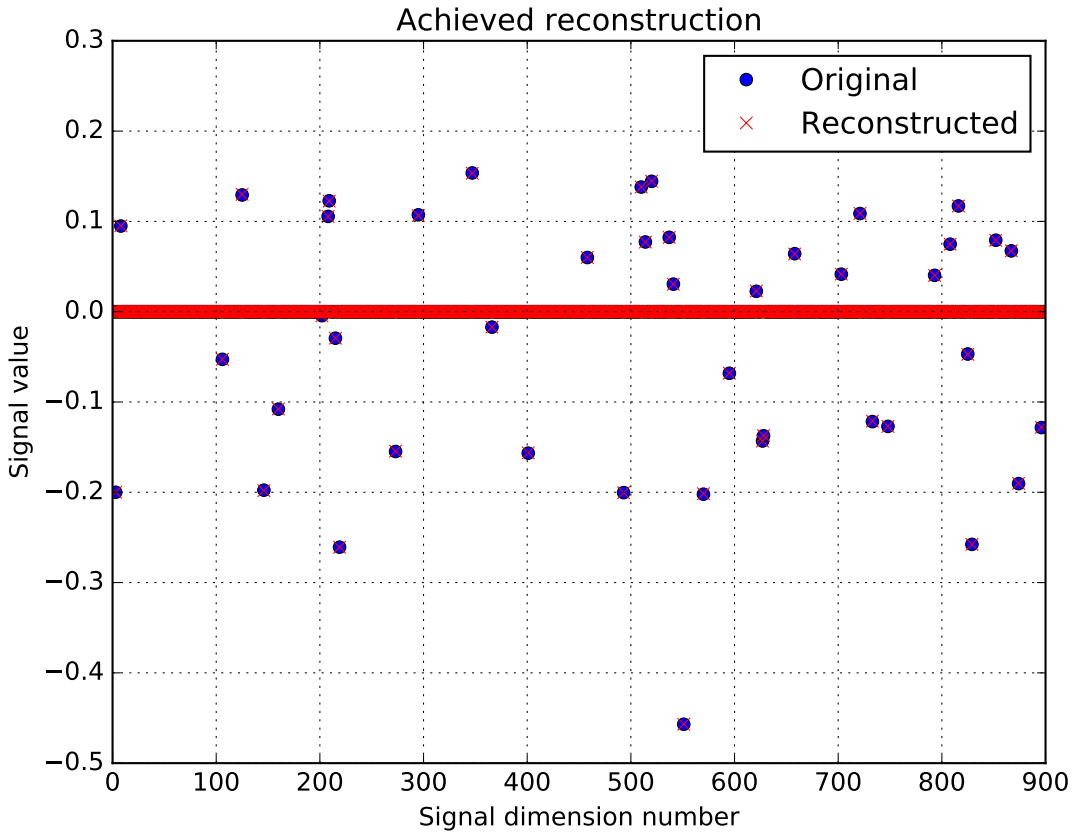
```
>>> from pyunlocbox import solvers
>>> solver = solvers.douglas_rachford(step=1e-2)
```

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 43 iterations:
    objective function f(sol) = 5.607407e+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')
```

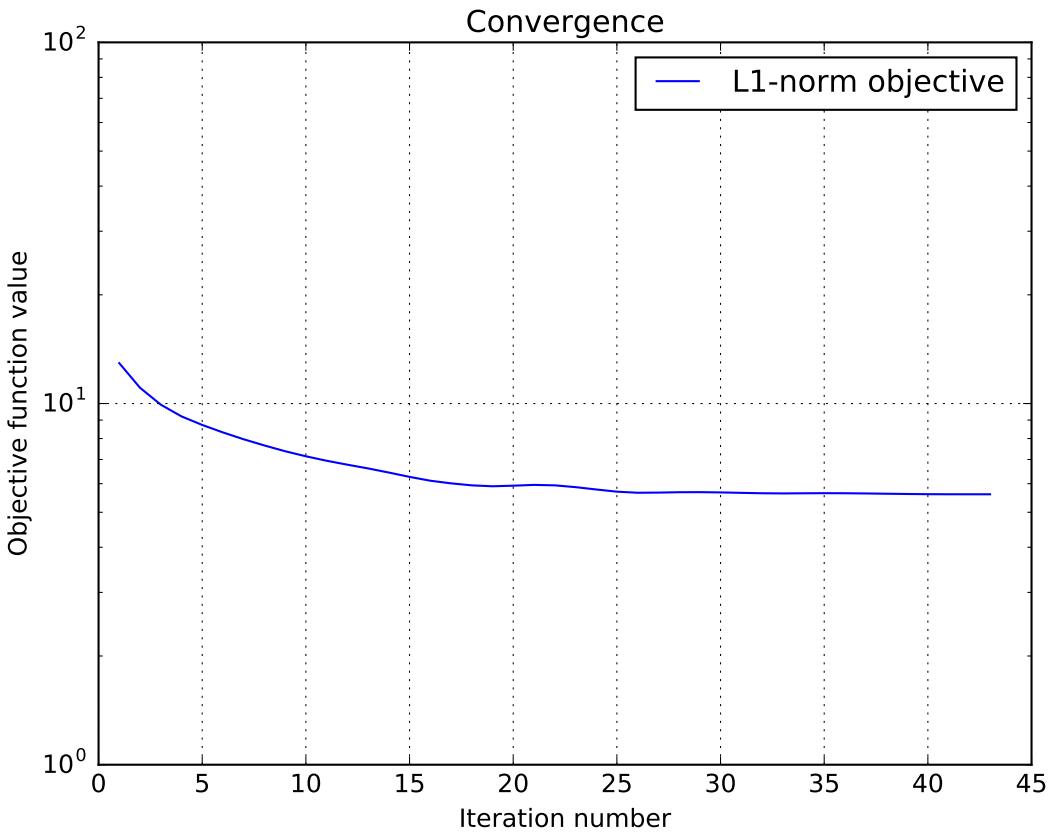


Image reconstruction (Forward-Backward, Total Variation, L2-norm)

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

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

where $\|\cdot\|_{\text{TV}}$ 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

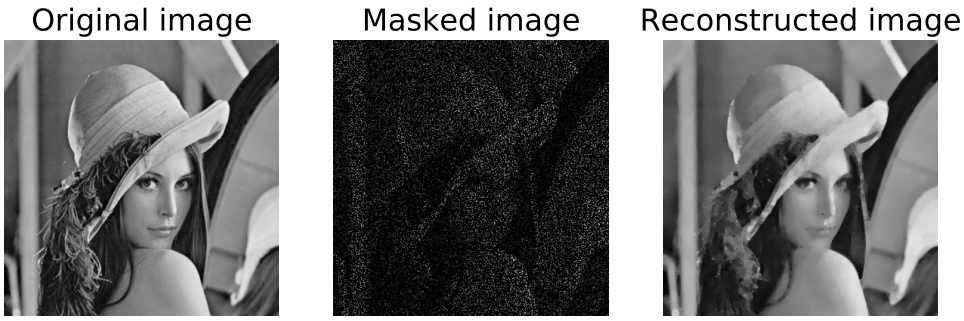
```
>>> from pyunlocbox import solvers
>>> solver = solvers.forward_backward(step=0.5/tau)
```

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 93 iterations:
    objective function f(sol) = 4.268861e+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).

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

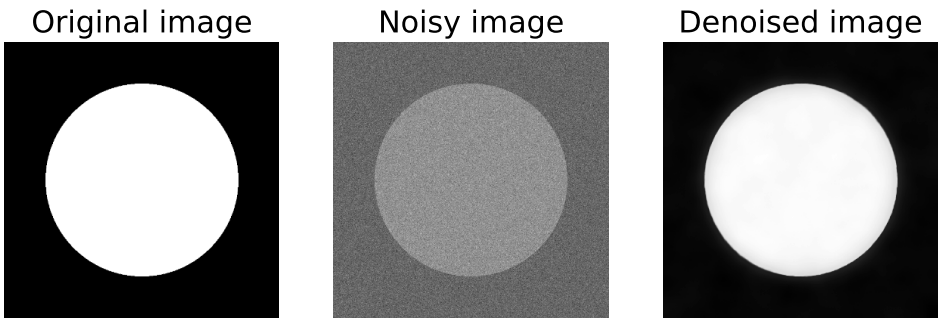
```
>>> from pyunlocbox import solvers
>>> solver = solvers.douglas_rachford(step=0.1)
```

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).

Reference guide

Package overview

The toolbox is organized around two class 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
- *pyunlocbox.operators*: useful operators to be passed to the functions
- *pyunlocbox.acceleration*: acceleration schemes to be passed to the solvers, implement the acceleration class hierarchy

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 (semi-orthogonal linear transform), 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

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

This method is required by some solvers.

When the map A in the function construction is a tight frame (semi-orthogonal linear transformation), we can use property (x) of Table 10.1 in *[CP11]* to compute the proximal operator of the composition of A with the base function. Whenever this is not the case, we have to resort to some iterative procedure, which may be very inefficient.

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

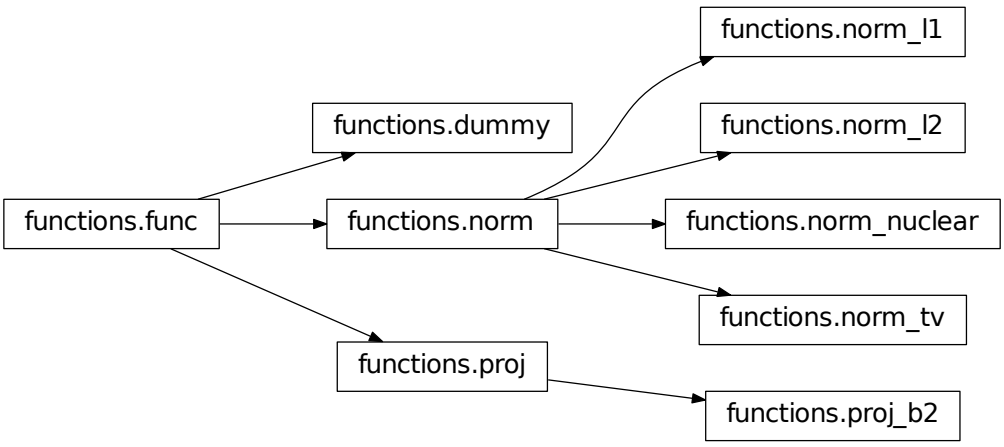
- The *tol* parameter is defined as the tolerance for the projection on the L2-ball. The algorithm stops if $\frac{\epsilon}{1-tol} \leq \|y - A(z)\|_2 \leq \frac{\epsilon}{1+tol}$.
- The evaluation of this function is zero.
- The L2-ball proximal operator evaluated at x is given by $\arg\min_z \frac{1}{2}\|x - z\|_2^2 + i_S(z)$ 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.



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
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: 2.963739e-01
    dummy evaluation: 0.000000e+00
    objective = 2.96e-01
Iteration 3 of forward_backward:
    norm_l2 evaluation: 7.902529e-02
    dummy evaluation: 0.000000e+00
    objective = 7.90e-02
Iteration 4 of forward_backward:
    norm_l2 evaluation: 5.752265e-02
    dummy evaluation: 0.000000e+00
    objective = 5.75e-02
Iteration 5 of forward_backward:
    norm_l2 evaluation: 5.142032e-03
    dummy evaluation: 0.000000e+00
    objective = 5.14e-03
Solution found after 5 iterations:
    objective function f(sol) = 5.142032e-03
    stopping criterion: ATOL
```

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

```
>>> np.linalg.norm(ret['sol'] - y)**2
0.00514203...
```

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

```
>>> ret['sol']
array([ 4.02555301,  5.03194126,  6.03832952,  7.04471777])
```

Show the used solver:

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

Show some information about the convergence:

```
>>> ret['crit']
'ATOL'
>>> ret['niter']
5
>>> ret['time']
0.0012578964233398438
>>> ret['objective']
[[126.0, 0], [13.99999999..., 0], [0.29637392..., 0], [0.07902528..., 0],
[0.05752265..., 0], [0.00514203..., 0]]
```

Solver class hierarchy

Solver object interface

class pyunlocbox.solvers.**solver**(*step=1.0*, *accel=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.

accel : pyunlocbox.acceleration.accel

User-defined object used to adaptively change the current step size and solution while the algorithm is running. Default is a dummy object that returns unchanged values.

algo(*objective*, *niter*)

Call the solver iterative algorithm and the provided acceleration scheme. See parameters documentation in *pyunlocbox.solvers.solve()*

Notes

The method `self.accel.update_sol()` is called before `self._algo()` because the acceleration schemes usually involves some sort of averaging of previous solutions, which can add some un-

wanted artifacts on the output solution. With this ordering, we guarantee that the output of solver.algo is not corrupted by the acceleration scheme.

Similarly, the method `self.accel.update_step()` is called after `self._algo()` to allow the step update procedure to act directly on the solution output by the underlying algorithm, and not on the intermediate solution output by the acceleration scheme in `self.accel.update_sol()`.

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()*.

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

Notes

When preprocessing the functions, the solver should split them into two lists: * *self.smooth_funs*, for functions involved in gradient steps. * *self.non_smooth_funs*, for functions involved proximal steps. This way, any method that takes in the solver as argument, such as the methods in *pyunlocbox.acceleration.accel*, can have some context as to how the solver is using the functions.

Gradient descent algorithm

class pyunlocbox.solvers.**gradient_descent**(***kwargs*)
Bases: *pyunlocbox.solvers.solver*

Gradient descent algorithm.

This algorithm solves optimization problems composed of the sum of any number of smooth functions.

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

Notes

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

See *pyunlocbox.acceleration.regularized_nonlinear* for a very efficient acceleration scheme for this method.

Examples

```
>>> from pyunlocbox import functions, solvers
>>> import numpy as np
>>> dim = 25;
>>> np.random.seed(0)
>>> xstar = np.random.rand(dim) # True solution
>>> x0 = np.random.rand(dim)
>>> x0 = xstar + 5.*(x0 - xstar) / np.linalg.norm(x0 - xstar)
>>> A = np.random.rand(dim, dim)
>>> step = 1/np.linalg.norm(np.dot(A.T, A))
>>> f = functions.norm_l2(lambda_=0.5, A=A, y=np.dot(A, xstar))
>>> fd = functions.dummy()
>>> solver = solvers.gradient_descent(step=step)
>>> params = {'rtol':0, 'maxit':14000, 'verbosity':'NONE'}
>>> ret = solvers.solve([f, fd], x0, solver, **params)
```

```
>>> pctdiff = 100*np.sum((xstar - ret['sol'])**2)/np.sum(xstar**2)
>>> print('Difference: {0:.1f}%'.format(pctdiff))
Difference: 1.3%
```

Forward-backward proximal splitting algorithm

class pyunlocbox.solvers.**forward_backward**(*accel=<pyunlocbox.acceleration.fista object>*, ***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 **accel** : *pyunlocbox.acceleration.accel*

Acceleration scheme to use. Default is *pyunlocbox.acceleration.fista()*, which corresponds to the 'FISTA' solver. Passing *pyunlocbox.acceleration.dummy()* instead results in the ISTA solver. Note that while FISTA is much more time-efficient, it is less memory-efficient.

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(step=0.5)
>>> ret = solvers.solve([f1, f2], x0, solver, atol=1e-5)
Solution found after 15 iterations:
    objective function f(sol) = 4.957288e-07
    stopping criterion: ATOL
>>> ret['sol']
array([ 4.0002509 ,  5.00031362,  6.00037635,  7.00043907])
```

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_=1.0*, **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.

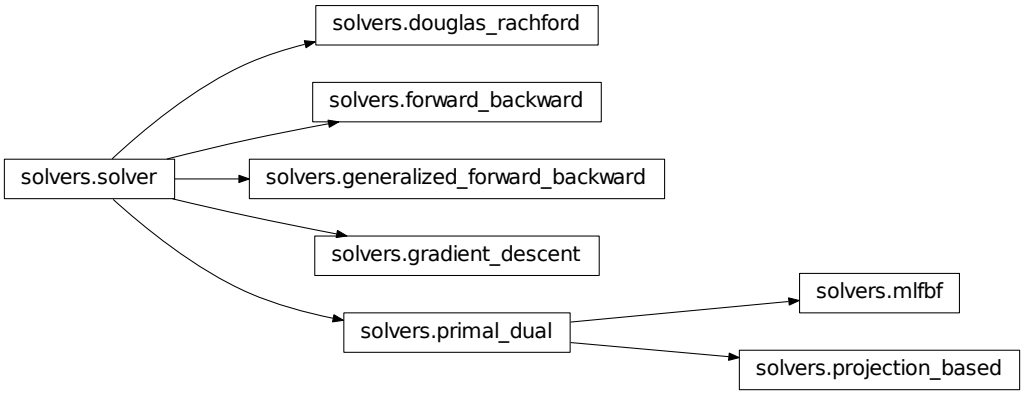
Examples

```
>>> 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 :

- *gradient_descent*: Gradient descent algorithm.
- *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.



Acceleration module

Acceleration class hierarchy

Acceleration scheme object interface

class pyunlocbox.acceleration.**accel**
Bases: object

Defines the acceleration scheme object interface.

This class defines the interface of an acceleration scheme object intended to be passed to a solver inheriting from *pyunlocbox.solvers.solver*. It is intended to be a base class for standard acceleration schemes 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 acceleration scheme objects.

post()
Post-processing specific to the acceleration scheme.

Mainly used to delete references added during initialization so that the garbage collector can free the memory. Gets called when *pyunlocbox.solvers.solve()* finishes running.

pre (*functions*, *x0*)
Pre-processing specific to the acceleration scheme.

Gets called when *pyunlocbox.solvers.solve()* starts running.

update_sol (*solver*, *objective*, *niter*)
Update the solution point for the next iteration.

Parameters **solver** : pyunlocbox.solvers.solver

Solver on which to act.

objective : list of floats

List of evaluations of the objective function since the beginning of the iterative process.

niter : int

Current iteration number.

Returns array_like

Updated solution point.

update_step (*solver*, *objective*, *niter*)
Update the step size for the next iteration.

Parameters **solver** : pyunlocbox.solvers.solver

Solver on which to act.

objective : list of floats

List of evaluations of the objective function since the beginning of the iterative process.

niter : int

Current iteration number.

Returns float

Updated step size.

Dummy acceleration scheme

class pyunlocbox.acceleration.**dummy**
Bases: *pyunlocbox.acceleration.accel*

Dummy acceleration scheme.

Used by default in most of the solvers. It simply returns unaltered the step size and solution point it receives.

Backtracking from quadratic approximation

class pyunlocbox.acceleration.**backtracking**(*eta=0.5*, ***kwargs*)
Bases: *pyunlocbox.acceleration.dummy*

Backtracking based on a local quadratic approximation of the the smooth part of the objective.

Parameters **eta** : float

A number between 0 and 1 representing the ratio of the geometric sequence formed by successive step sizes. In other words, it establishes the relation *step_new = eta * step_old*. Default is 0.5.

Notes

This is the backtracking strategy used in the original FISTA paper, *[BT09a]*.

Examples

```
>>> from pyunlocbox import functions, solvers, acceleration
>>> import numpy as np
>>> y = [4, 5, 6, 7]
>>> x0 = np.zeros(len(y))
>>> f1 = functions.norm_l1(y=y, lambda_=1.0)
>>> f2 = functions.norm_l2(y=y, lambda_=0.8)
>>> accel = acceleration.backtracking()
>>> solver = solvers.forward_backward(accel=accel, step=10)
>>> ret = solvers.solve([f1, f2], x0, solver, atol=1e-32, rtol=None)
Solution found after 4 iterations:
    objective function f(sol) = 0.000000e+00
    stopping criterion: ATOL
>>> ret['sol']
array([ 4.,  5.,  6.,  7.])
```

FISTA acceleration scheme

class pyunlocbox.acceleration.**fista**(***kwargs*)

Bases: *pyunlocbox.acceleration.dummy*

Acceleration scheme for forward-backward solvers.

Notes

This is the acceleration scheme proposed in the original FISTA paper, *[BT09a]*.

Examples

```
>>> from pyunlocbox import functions, solvers, acceleration
>>> import numpy as np
>>> y = [4, 5, 6, 7]
>>> x0 = np.zeros(len(y))
>>> f1 = functions.norm_l2(y=y)
>>> f2 = functions.dummy()
>>> accel=acceleration.fista()
>>> solver = solvers.forward_backward(accel=accel, step=0.5)
>>> ret = solvers.solve([f1, f2], x0, solver, atol=1e-5)
Solution found after 15 iterations:
    objective function f(sol) = 4.957288e-07
    stopping criterion: ATOL
>>> ret['sol']
array([ 4.0002509 ,  5.00031362,  6.00037635,  7.00043907])
```

FISTA acceleration scheme with backtracking

class pyunlocbox.acceleration.**fista_backtracking**(*eta=0.5*, ***kwargs*)

Bases: *pyunlocbox.acceleration.backtracking*, *pyunlocbox.acceleration.fista*

Acceleration scheme with backtracking for forward-backward solvers.

Notes

This is the acceleration scheme and backtracking strategy proposed in the original FISTA paper, *[BT09a]*.

Examples

```
>>> from pyunlocbox import functions, solvers, acceleration
>>> import numpy as np
>>> y = [4, 5, 6, 7]
>>> x0 = np.zeros(len(y))
>>> f1 = functions.norm_l2(y=y)
>>> f2 = functions.dummy()
>>> accel=acceleration.fista_backtracking()
>>> solver = solvers.forward_backward(accel=accel, step=0.5)
>>> ret = solvers.solve([f1, f2], x0, solver, atol=1e-5)
Solution found after 15 iterations:
    objective function f(sol) = 4.957288e-07
    stopping criterion: ATOL
>>> ret['sol']
array([ 4.0002509 ,  5.00031362,  6.00037635,  7.00043907])
```

Regularized Nonlinear Acceleration (RNA)

class pyunlocbox.acceleration.**regularized_nonlinear**(*k=10*, *lambda_=1e-06*, *adaptive=True*, *dolinesearch=True*, *forcedecrease=True*, ***kwargs*)

Bases: *pyunlocbox.acceleration.dummy*

Regularized nonlinear acceleration (RNA) for gradient descent.

Parameters **k** : int, optional

Number of points to keep in the buffer for computing the extrapolation. (Default is 10.)

lambda_ : float or list of floats, optional

Regularization parameter in the acceleration scheme. The user can pass a list of candidates, and the acceleration algorithm will pick the one that provides the best extrapolation. (Default is 1e-6.)

adaptive : boolean, optional

If adaptive = True and the user has not provided a list of regularization parameters, the acceleration algorithm will assemble a grid of possible regularization parameters based on the SVD of the Gram matrix of vectors of differences in the extrapolation buffer. If adaptive = False, the algorithm will simply try to use the value(s) given in *lambda_*. (Default is True.)

dolinesearch : boolean, optional

If dolinesearch = True, the acceleration scheme will try to return a point in the line segment between the current extrapolation and the previous one that provides a decrease in the value of the objective function. If dolinesearch = False, the algorithm simply returns the current extrapolation. (Default is True.)

forcedecrease : boolean, optional

If forcedecrese = True and we obtain a bad extrapolation, the algorithm returns the unchanged solution produced by the solver. If forcedecrease = False, the algorithm returns the new extrapolation no matter what. (Default is True.)

Notes

This is the acceleration scheme proposed in *[SdB16]*.

See also Damien Scieur's repository for the Matlab version that inspired this implementation.

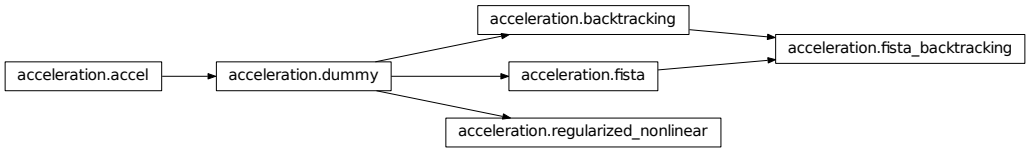
Examples

```
>>> from pyunlocbox import functions, solvers, acceleration
>>> import numpy as np
>>> dim = 25;
>>> np.random.seed(0)
>>> xstar = np.random.rand(dim) # True solution
>>> x0 = np.random.rand(dim)
>>> x0 = xstar + 5.*(x0 - xstar) / np.linalg.norm(x0 - xstar)
>>> A = np.random.rand(dim, dim)
>>> step = 1/np.linalg.norm(np.dot(A.T, A))
>>> f = functions.norm_l2(lambda_=0.5, A=A, y=np.dot(A, xstar))
>>> fd = functions.dummy()
>>> accel = acceleration.regularized_nonlinear(k=5)
>>> solver = solvers.gradient_descent(step=step, accel=accel)
>>> params = {'rtol':0, 'maxit':200, 'verbosity':'NONE'}
>>> ret = solvers.solve([f, fd], x0, solver, **params)
>>> pctdiff = 100*np.sum((xstar - ret['sol'])**2)/np.sum(xstar**2)
>>> print('Difference: {0:.1f}%'.format(pctdiff))
Difference: 1.3%
```

lambda_

This module implements acceleration schemes for use with the *pyunlocbox.solvers*. Pass a given acceleration object as an argument to your chosen solver during its initialization so that the solver can use it. The base class *accel* defines the interface of all acceleration objects. The specialized acceleration objects inherit from it and implement the class methods. The following acceleration schemes are included:

- *dummy*: Dummy acceleration scheme. It does nothing.
- *backtracking*: Backtracking line search.
- *fista*: FISTA acceleration scheme.
- *fista_backtracking*: FISTA with backtracking.
- *regularized_nonlinear*: Regularized nonlinear acceleration.



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

History

0.5.1 (2017-07-04)

Development status updated from Alpha to Beta.

New features:

- Acceleration module, decoupling acceleration strategies from the solvers
  - Backtracking scheme
  - FISTA acceleration
  - FISTA with backtracking
  - Regularized non-linear acceleration (RNA)
- Solvers: gradient descent algorithm

Bug fix:

- Decrease dimensionality of variables in Douglas Rachford tutorial to reduce test time and timeout on Travis CI.

Infrastructure:

- Continuous integration: dropped 3.3 (matplotlib dropped it), added 3.6
- We don't build PDF documentation anymore. Less burden, HTML can be downloaded from readthedocs.

0.4.0 (2016-08-01)

New feature:

- Monotone+Lipschitz forward-backward-forward primal-dual algorithm (MLFBF)

Bug fix:

- Plots generated when building documentation (not stored in the repository)

Infrastructure:

- Continuous integration: dropped 2.6 and 3.2, added 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

0.3.0 (2015-05-29)

New features:

- Generalized forward-backward splitting algorithm
- Projection-based primal-dual algorithm
- TV-norm function (eval, prox)
- Nuclear-norm function (eval, prox)
- L2-norm proximal operator supports non-tight frames
- Two new tutorials using the TV-norm with Forward-Backward and Douglas-Rachford for image reconstruction and denoising
- New stopping criterion XTOL allows to stop when the variable is stable

Bug fix:

- Much more memory efficient. Note that the array which contains the initial solution is now modified in place.

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

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

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)
- 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

References

Bibliography

[BT09a] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 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*, 2009.

[CR07] Emmanuel Candes and Justin Romberg. Sparsity and incoherence in compressive sampling. *Inverse problems*, 2007. arXiv:math/0611957.

[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*, 2007.

[CP11] Patrick L Combettes and Jean-Christophe Pesquet. Proximal splitting methods in signal processing. In *Fixed-Point Algorithms for Inverse Problems in Science and Engineering*. 2011. arXiv:0912.3522.

[KP15] Nikos Komodakis and Jean-Christophe Pesquet. Playing with duality. *IEEE Signal Processing Magazine*, 2015. arXiv:1406.5429.

[RFP13] Hugo Raguet, Jalal Fadili, and Gabriel Peyré. A generalized forward-backward splitting. *SIAM Journal on Imaging Sciences*, 2013. arXiv:1108.4404.

[SdB16] Damien Scieur, Alexandre dAspremont, and Francis Bach. Regularized nonlinear acceleration. *arXiv*, 2016. arXiv:1606.04133.

Python Module Index

p

Index