pyunlocbox Documentation

Release 0.1.0

EPFL LTS2

July 29, 2016

About

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.

- Code : https://github.com/epfl-lts2/pyunlocbox
- Documentation : http://pyunlocbox.readthedocs.org
- PyPI package : https://pypi.python.org/pypi/pyunlocbox
- Travis continuous integration : https://travis-ci.org/epfl-lts2/pyunlocbox
- UNLocBoX matlab toolbox : http://unlocbox.sourceforge.net

1.1 Features

- Solvers
  - Forward-backward splitting algorithm
- Proximal operators
  - L1-norm
  - L2-norm

1.2 Installation

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 requires quite some dependencies.

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

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

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

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

1.3 Authors

PyUNLocBoX was started in 2014 as an academic project for research purpose of the LTS2 laboratory from EPFL. See our website at http://lts2www.epfl.ch.

Development lead :

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

Contributors :

- None yet. Why not be the first ?

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.

2.1 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, absTol=1e-5, verbosity='high')
INFO: Forward-backward method : FISTA
Iteration   1 : objective = 1.40e+01, relative = 8.00e+00
Iteration   2 : objective = 1.56e+00, relative = 8.00e+00
Iteration   3 : objective = 3.29e-02, relative = 4.62e+01
Iteration   4 : objective = 8.78e-03, relative = 2.75e+00
Iteration   5 : objective = 6.39e-03, relative = 3.74e-01
Iteration   6 : objective = 5.71e-04, relative = 1.02e+01
Iteration   7 : objective = 1.73e-05, relative = 3.21e+01
Iteration   8 : objective = 6.11e-05, relative = 7.17e-01
Iteration   9 : objective = 1.21e-05, relative = 4.04e+00
Iteration  10 : objective = 7.46e-09, relative = 1.62e+03
Solution found after 10 iterations :
    objective function f(sol) = 7.460428e-09
    last relative objective improvement : 1.624424e+03
    stopping criterion : ABS_TOL
```

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
>>> import matplotlib, sys
>>> cmd_backend = 'matplotlib.use("AGG")'
>>> _ = eval(cmd_backend) if 'matplotlib.pyplot' not in sys.modules else 0
>>> import matplotlib.pyplot as plt
>>> objective = np.array(ret['objective'])
>>> _ = 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')
>>> _ = plt.savefig('doc/tutorials/simple_convergence.pdf')
>>> _ = plt.savefig('doc/tutorials/simple_convergence.png')
```

The below 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`).
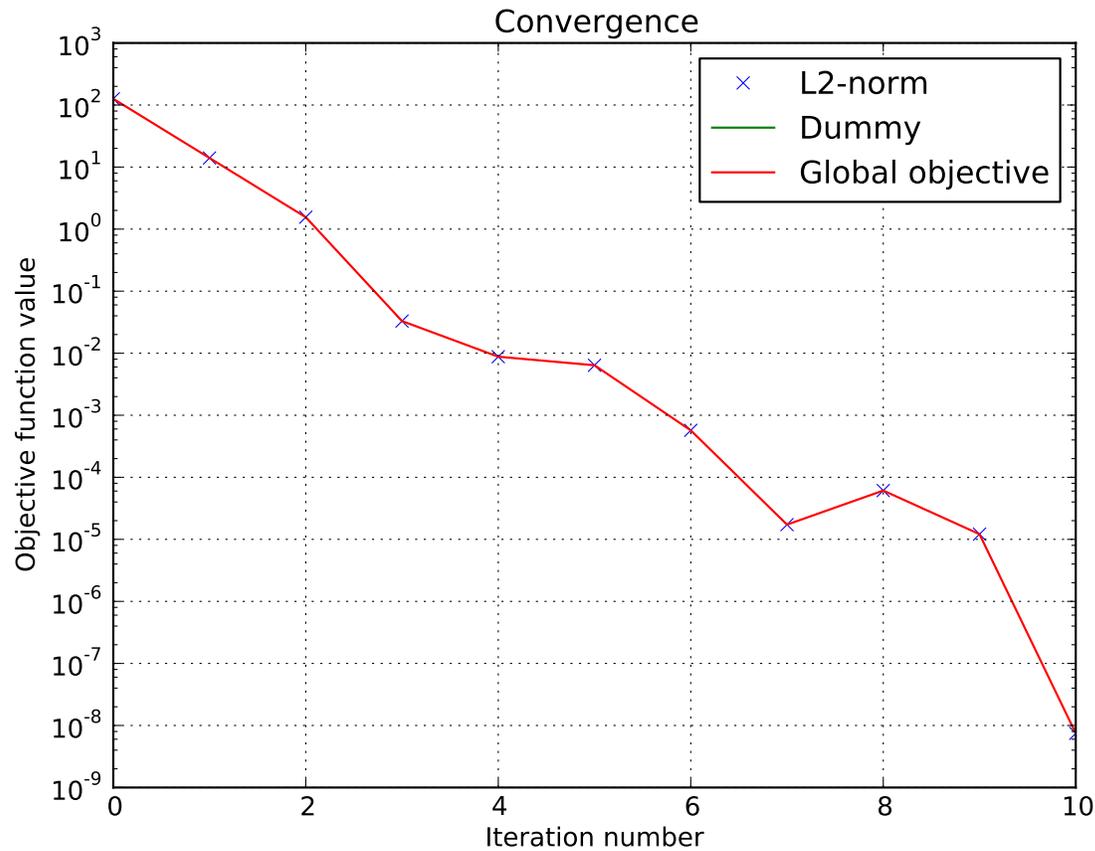
2.2 Compressed sensing using forward-backward

This tutorial presents a compressed sensing problem solved by the forward-backward splitting algorithm. The problem can be expressed as follow :

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

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

We first declare the signal size *N* and the sparsity level *K* :

```
>>> N = 5000
>>> K = 100
```

The number of measurements *M* is computed with respect to the size of the signal *N* and the sparsity level *K* :

```
>>> import numpy as np
>>> R = max(4, np.ceil(np.log(N)))
>>> M = K * R
>>> print('Number of measurements : %d' % (M,))
Number of measurements : 900
>>> print('Compression ratio : %3.2f' % (N/M,))
Compression ratio : 5.56
```

Note: With the above defined number of measurements, the algorithm is supposed to very often perform a perfect reconstruction.

We now generate a random measurement matrix :

```
>>> np.random.seed(1)  # Reproducible results.
>>> A = np.random.standard_normal((M, N))
```

And create the *K* sparse signal :

```
>>> x = np.zeros(N)
>>> I = np.random.permutation(N)
>>> x[I[0:K]] = np.random.standard_normal(K)
>>> x = x / np.linalg.norm(x)
```

We are now able to compute the measured signal :

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

The first objective function to minimize is defined by :

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

which is an L1-norm. The L1-norm function object is part of the toolbox standard function objects and can be instantiated as follow (the regularization parameter τ is implicitly set to 1.0):

```
>>> from pyunlocbox import functions
>>> f1 = functions.norm_l1(verbosity='none')
```

Note: You can also pass a verbosity of 'low' or 'high' if you want some informations about the norm evaluation. Please see the documentation of the norm function object for more information on how to instantiate norm objects (*pyunlocbox.functions.norm*).

The second objective function to minimize is defined by :

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

which is an L2-norm that is also part of the standard function objects. It can be instantiated as follow :

```
>>> f2 = functions.norm_l2(y=y, A=A, verbosity='none')
```

or alternatively as follow :

```
>>> A_ = lambda x: np.dot(A, x)
>>> At_ = lambda x: np.dot(np.transpose(A), x)
>>> f3 = functions.norm_l2(y=y, A=A_, At=At_, verbosity='none')
```

Note: In this case the forward and adjoint operators were passed as real operators 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(np.transpose(A), 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 will 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 given by

$$\beta = 2 \cdot \operatorname{norm}(A)^2$$

To solve this problem, we use the forward-backward splitting algorithm which is instantiated as follow :

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

Note: See the solver documentation for more information (*pyunlocbox.solvers.forward_backward*).

The problem is then solved by executing the solver on the objective functions, after the setting of a starting point *x0* :

```
>>> x0 = np.zeros(N)
>>> ret = solvers.solve([f1, f2], x0, solver, relTol=1e-4, maxIter=300)
Solution found after 176 iterations :
    objective function f(sol) = 8.221302e+00
    last relative objective improvement : 8.363264e-05
    stopping criterion : REL_TOL
```

Note: See the solving function documentation for more information on the parameters and the returned values (*pyunlocbox.solvers.forward_backward*).

Lets display the results :

```
>>> import matplotlib, sys
>>> cmd_backend = 'matplotlib.use("AGG")'
>>> _ = eval(cmd_backend) if 'matplotlib.pyplot' not in sys.modules else 0
>>> 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')
>>> _ = plt.savefig('doc/tutorials/compressed_sensing_1_results.pdf')
>>> _ = plt.savefig('doc/tutorials/compressed_sensing_1_results.png')
```
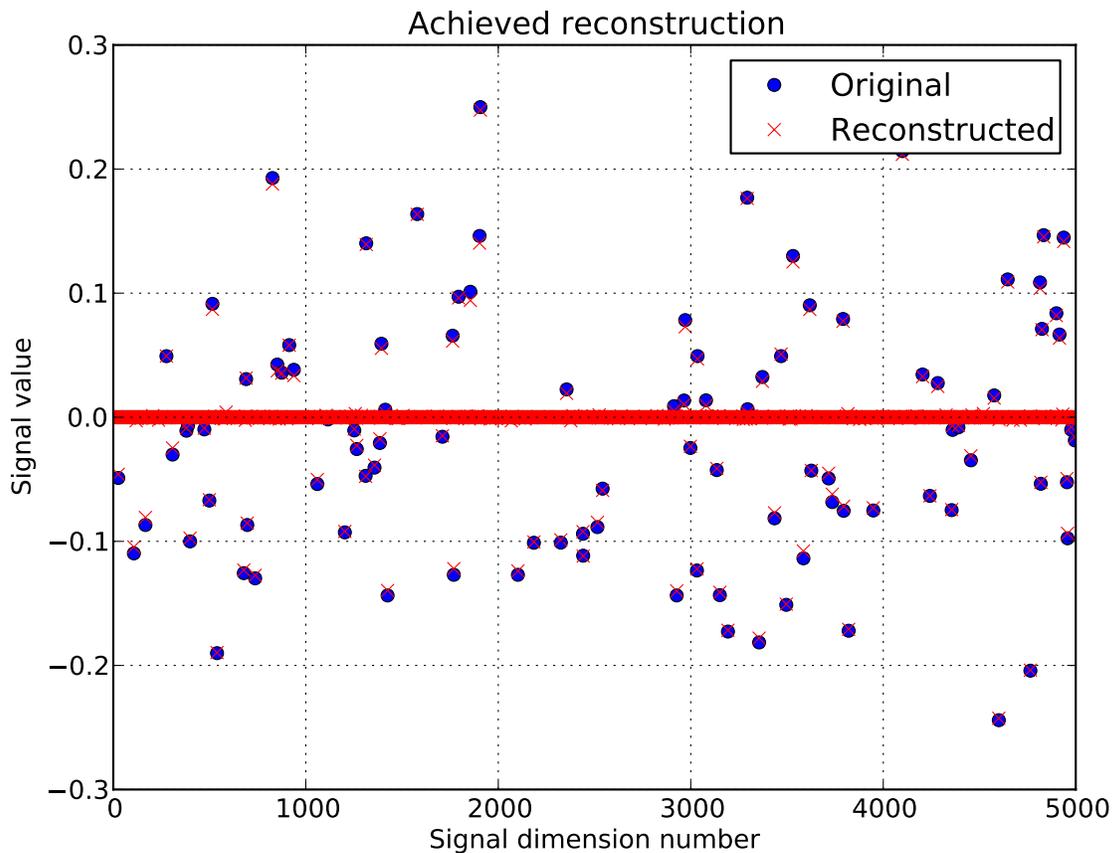
The above figure shows a good reconstruction which is both sparse (thanks to the L1-norm objective) and close to the measurements (thanks to the L2-norm objective).

We can also 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')
>>> _ = plt.savefig('doc/tutorials/compressed_sensing_1_convergence.pdf')
>>> _ = plt.savefig('doc/tutorials/compressed_sensing_1_convergence.png')
```
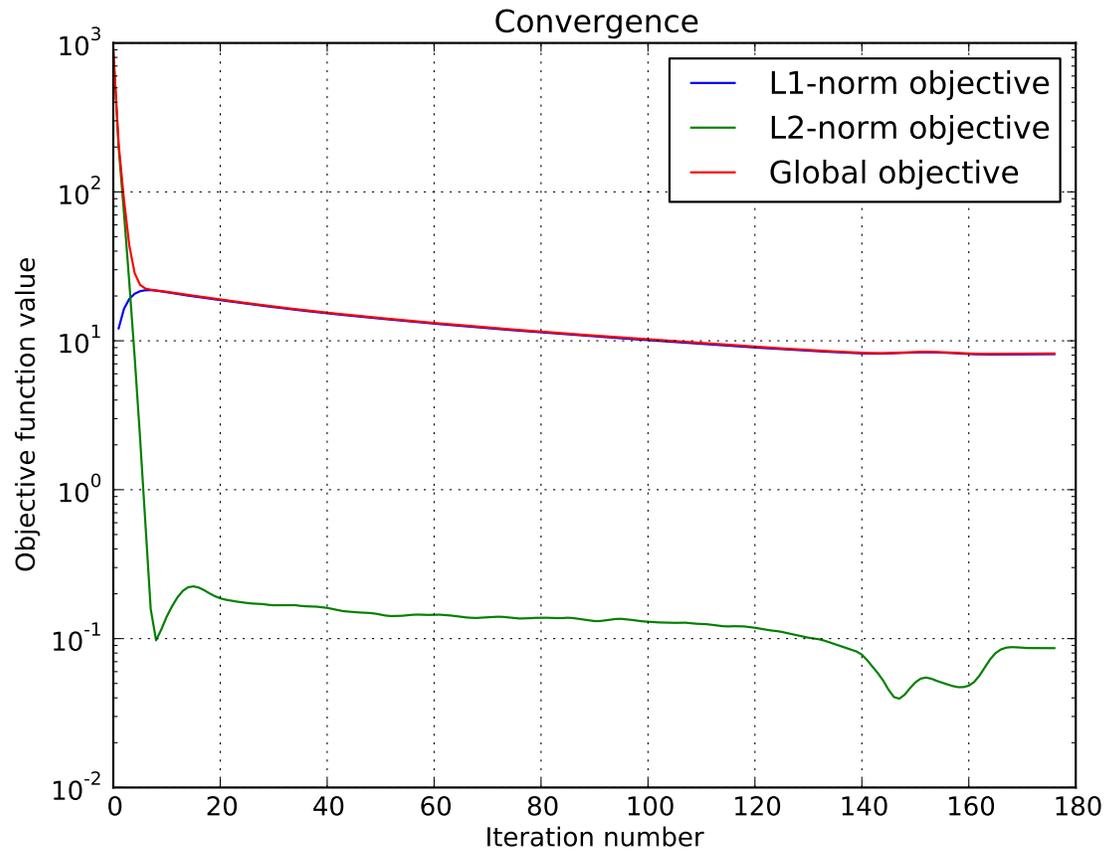
Reference guide

3.1 Toolbox overview

PyUNLocBoX is a convex optimization toolbox using proximal splitting methods. It is a port of the Matlab UNLocBoX toolbox.

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.

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

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, absTol=1e-5)
Solution found after 10 iterations :
    objective function f(sol) = 7.460428e-09
    last relative objective improvement : 1.624424e+03
    stopping criterion : ABS_TOL
>>> ret['sol']
array([ 3.99996922,  4.99996153,  5.99995383,  6.99994614])
```

3.2 Functions module

3.2.1 Function objects

Interface

class pyunlocbox.functions.**func**(*verbosity='none'*)
    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 verbosity** : {'none', 'low', 'high'}, optional
>
> > The log level : 'none' for no log, 'low' for resume at convergence, 'high' to for all steps. Default is 'low'.

Examples

Lets 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])
```

eval(*x*)

> Function evaluation.
>
> > **Parameters x** : array_like
> >
> > > The evaluation point.
> >
> > **Returns z** : float
> >
> > > The objective function evaluated at *x*.

> #### Notes
>
> This method is required by the *pyunlocbox.solvers.solve()* solving function to evaluate the objective function.

grad(*x*)

> Function gradient.
>
> > **Parameters x** : array_like
> >
> > > The evaluation point.
> >
> > **Returns z** : ndarray
> >
> > > The objective function gradient evaluated at *x*.

> #### Notes
>
> This method is required by some solvers.

prox(*x*, *T*)

> Function proximal operator.

Parameters **x** : array_like

The evaluation point.

T : float

The regularization parameter.

Returns **z** : ndarray

The proximal operator evaluated at *x*.

Notes

This method is required by some solvers.

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

Dummy function

class pyunlocbox.functions.**dummy**(*verbosity='none'*)

Bases: *pyunlocbox.functions.func*

Dummy function object.

This can be used as a second function object when there is only one function to minimize. The eval(), prox() and grad() methods then all return 0.

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

Examples

```
>>> import pyunlocbox
>>> f = pyunlocbox.functions.dummy(verbosity='low')
>>> x = [1, 2, 3, 4]
>>> f.eval(x)
dummy evaluation : 0.000000e+00
0
>>> f.prox(x, 1)
array([ 0.,  0.,  0.,  0.])
>>> f.grad(x)
array([ 0.,  0.,  0.,  0.])
```

3.2.2 Norm function class hierarchy

Base class

class pyunlocbox.functions.**norm**(*lambda_=1, y=0, w=1, A=None, At=None, tight=True, nu=1, *args, **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.

y : array_like, optional

Measurements. Default is 0.

w : array_like, optional

Weights for a weighted norm. Default is 1.

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.

L1-norm

class `pyunlocbox.functions.`**`norm_l1`** (*lambda_=1, y=0, w=1, A=None, At=None, tight=True, nu=1, *args, **kwargs*)

Bases: *pyunlocbox.functions.norm*

L1-norm function object.

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

Notes

- The L-1 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 $\min_z \frac{1}{2}||x - z||_2^2 + \gamma ||w \cdot (A(z) - y)||_1$ where $\gamma = \lambda \cdot T$ This is simply a soft thresholding.

Examples

```
>>> import pyunlocbox
>>> f = pyunlocbox.functions.norm_l1(verbosity='low')
>>> f.eval([1, 2, 3, 4])
norm_l1 evaluation : 1.000000e+01
10
>>> f.prox([1, 2, 3, 4], 1)
array([ 0.,  1.,  2.,  3.])
```

L2-norm

class pyunlocbox.functions.**norm_l2**(*lambda_=1, y=0, w=1, A=None, At=None, tight=True, nu=1, *args, **kwargs*)

Bases: *pyunlocbox.functions.norm*

L2-norm function object.

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

Notes

- The squared L-2 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 $\min_z \frac{1}{2}||x - z||_2^2 + \gamma||w \cdot (A(z) - y)||_2^2$ 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(verbosity='low')
>>> x = [1, 2, 3, 4]
>>> f.eval(x)
norm_l2 evaluation : 3.000000e+01
30
>>> f.prox(x, 1)
array([ 0.33333333,  0.66666667,  1.        ,  1.33333333])
>>> f.grad(x)
array([2, 4, 6, 8])
```

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 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.

3.3 Solvers module

3.3.1 Solving function

pyunlocbox.solvers.**solve**(*functions, x0, solver=None, relTol=0.001, absTol=-inf, convergence_speed=-inf, maxIter=200, verbosity='low'*)

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

This function minimizes the objective function $f(x) = \sum_{k=0}^{k=M} f_k(x)$, i.e. solves $\arg\min_x \sum_{k=0}^{k=M} f_k(x)$ for $x \in \mathbb{R}^N$ using whatever algorithm. 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, $x_0 \in \mathbb{R}^N$.

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.

relTol : float, optional

> The convergence (relative tolerance) stopping criterion. The algorithm stops if $\frac{n(k)-n(k-1)}{n(k)} < reltol$ where $n(k) = f(x) = f_1(x) + f_2(x)$ is the objective function at iteration k. Default is 10^{-3}.

absTol : float, optional

> The absolute tolerance stopping criterion. The algorithm stops if $n(k) < abstol$. Default is minus infinity.

convergence_speed : float, optional

> The minimum tolerable convergence speed of the objective function. The algorithm stops if n(k-1) - n(k) < *convergence_speed*. Default is minus infinity (i.e. the objective function may even increase).

maxIter : int, optional

> The maximum number of iterations. Default is 200.

verbosity : {'low', 'high', 'none'}, optional

> The log level : `'none'` for no log, `'low'` for resume at convergence, `'high'` to for all steps. Default is `'low'`.

Returns **sol** : ndarray

> problem solution

solver : str

> used solver

niter : int

> number of iterations

time : float

> execution time in seconds

eval : float

> final evaluation of the objective function $f(x)$

crit : {'MAX_IT', 'ABS_TOL', 'REL_TOL', 'CONV_SPEED'}

Used stopping criterion. 'MAX_IT' if the maximum number of iterations *maxIter* is reached, 'ABS_TOL' if the objective function value is smaller than *absTol*, 'REL_TOL' if the relative objective function improvement was smaller than *relTol* (i.e. the algorithm converged), 'CONV_SPEED' if the objective function improvement is smaller than *convergence_speed*.

rel : float

relative objective improvement at convergence

objective : ndarray

successive evaluations of the objective function at each iteration

Examples

Simple example showing the automatic selection of a solver (and a second function) :

```
>>> import pyunlocbox
>>> f1 = pyunlocbox.functions.norm_l2(y=[4, 5, 6, 7])
>>> ret = pyunlocbox.solvers.solve([f1], [0, 0, 0, 0], absTol=1e-5)
INFO: Added dummy objective function.
INFO: Selected solver : forward_backward
Solution found after 10 iterations :
    objective function f(sol) = 7.460428e-09
    last relative objective improvement : 1.624424e+03
    stopping criterion : ABS_TOL
>>> ret['sol']
array([ 3.99996922,  4.99996153,  5.99995383,  6.99994614])
```

3.3.2 Solver class hierarchy

Solver object interface

class pyunlocbox.solvers.**solver**(*gamma=1*, *post_gamma=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 gamma : float

The step size. This parameter is upper bounded by $\frac{1}{\beta}$ where the second convex function (gradient ?) is β Lipschitz continuous. Default is 1.

post_gamma : 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 : *gamma* (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 *gamma*. 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_gamma()`. 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 *gamma* step size while the algorithm is running. See parameters documentation in *pyunlocbox.solvers.solve()* documentation.

post(*verbosity*)
> Solver specific post-processing. See parameters documentation in *pyunlocbox.solvers.solve()* documentation.

pre(*functions*, *x0*, *verbosity*)
> 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 splitting algorithm.
>
> This algorithm solves convex optimization problems composed of the sum of two objective functions.
>
> 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'. 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.

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, gamma=1)
>>> ret = solvers.solve([f1, f2], x0, solver, absTol=1e-5)
Solution found after 10 iterations :
    objective function f(sol) = 7.460428e-09
    last relative objective improvement : 1.624424e+03
    stopping criterion : ABS_TOL
```

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

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.

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/epfl-lts2/pyunlocbox/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

4.1.4 Write Documentation

pyunlocbox could always use more documentation, whether as part of the official pyunlocbox docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/epfl-lts2/pyunlocbox/issues.

If you are proposing a feature:

- Explain in detail how it would work.

- Keep the scope as narrow as possible, to make it easier to implement.

- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *pyunlocbox* for local development.

1. Fork the *pyunlocbox* repo on GitHub.

2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pyunlocbox.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pyunlocbox
$ cd pyunlocbox/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 pyunlocbox tests
$ python setup.py test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.

2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.

3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/epfl-lts2/pyunlocbox/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_pyunlocbox
```

History

5.1 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

Indices and tables

- genindex
- modindex
- search

p

A

D

E

F

G

N

P

S