SPAMS: a SPArse Modeling Software, v2.3

Julien Mairal
julien.mairal@m4x.org

October 23, 2012

Contents

1 Introduction

SPAMS (SPArse Modeling Software) is an open-source optimization toolbox under licence GPLv3. It implements algorithms for solving various machine learning and signal processing problems involving sparse regularizations.

The library is coded in C++, is compatible with Linux, Mac, and Windows 32bits and 64bits Operating Systems. It is interfaced with Matlab, but can be called from any C++ application. A R and Python interface has been developed by Jean-Paul Chieze.

It requires an implementation of BLAS and LAPACK for performing efficient linear algebra operations such as the one provided by matlab/R, atlas, netlib, or the one provided by Intel (Math Kernel Library). It also exploits multi-core CPUs when this feature is supported by the compiler, through OpenMP.

The current licence is GPLv3 available at `http://www.gnu.org/licenses/gpl.html`, which limits its usage. For other usages (such as the use in proprietary softwares), please contact the author.

Version 2.3 of SPAMS is divided into three "toolboxes" and has a few additional miscellaneous functions:

- The **Dictionary learning and matrix factorization toolbox** contains the online learning technique of [19, 20] and its variants for solving various matrix factorization problems:
  - dictionary Learning for sparse coding;
  - sparse principal component analysis (seen as a sparse matrix factorization problem);
  - non-negative matrix factorization;
  - non-negative sparse coding.
- The **Sparse decomposition toolbox** contains efficient implementations of
  - Orthogonal Matching Pursuit, (or Forward Selection) [32, 24];
  - the LARS/homotopy algorithm [27, 8] (variants for solving Lasso and Elastic-Net problems);
  - a weighted version of LARS;
  - OMP and LARS when data come with a binary mask;
  - a coordinate-descent algorithm for ℓ_1-decomposition problems [11, 9, 33];
  - a greedy solver for simultaneous signal approximation as defined in [31, 30] (SOMP);
  - a solver for simulatneous signal approximation with ℓ_1/ℓ_2-regularization based on block-coordinate descent;

  - a homotopy method for the Fused-Lasso Signal Approximation as defined in [9] with the homotopy method presented in the appendix of [20];
  - a tool for projecting efficiently onto a few convex sets inducing sparsity such as the ℓ_1-ball using the method of [3, 17, 7], and Elastic-Net or Fused Lasso constraint sets as proposed in the appendix of [20].

- The **Proximal toolbox**: An implementation of proximal methods (ISTA and FISTA [1]) for solving a large class of sparse approximation problems with different combinations of loss and regularizations. One of the main features of this toolbox is to provide a robust stopping criterion based on *duality gaps* to control the quality of the optimization, whenever possible. It also handles sparse feature matrices for large-scale problems. The following regularizations are implemented:
  - Tikhonov regularization (squared ℓ_2-norm);
  - ℓ_1-norm, ℓ_2, ℓ_∞-norms;
  - Elastic-Net [35];
  - Fused Lasso [29];
  - tree-structured sum of ℓ_2-norms (see [14, 15]);
  - tree-structured sum of ℓ_∞-norms (see [14, 15]);
  - general sum of ℓ_∞-norms (see [21, 22]);
  - mixed ℓ_1/ℓ_2-norms on matrices [34, 26];
  - mixed ℓ_1/ℓ_∞-norms on matrices [34, 26];
  - mixed ℓ_1/ℓ_2-norms on matrices plus ℓ_1 [28, 10];
  - mixed ℓ_1/ℓ_∞-norms on matrices plus ℓ_1;
  - group-lasso with ℓ_2 or ℓ_∞-norms;
  - group-lasso+ℓ_1;
  - multi-task tree-structured sum of ℓ_∞-norms (see [21, 22]);
  - trace norm;
  - ℓ_0 pseudo-norm (only with ISTA);
  - tree-structured ℓ_0 (only with ISTA);
  - rank regularization for matrices (only with ISTA);
  - the path-coding penalties of [23].

  All of these regularization functions can be used with the following losses

  - square loss;
  - square loss with missing observations;
  - logistic loss, weighted logistic loss;
  - multi-class logistic.

  This toolbox can also enforce non-negativity constraints, handle intercepts and sparse matrices. There are also a few additional undocumented functionalities, which are available in the source code.

- A few tools for performing linear algebra operations such as a conjugate gradient algorithm, manipulating sparse matrices and graphs.

The toolbox was written by Julien Mairal when he was at INRIA, with the collaboration of Francis Bach (INRIA), Jean Ponce (Ecole Normale Supérieure), Guillermo Sapiro (University of Minnesota), Guillaume Obozinski (INRIA) and Rodolphe Jenatton (INRIA).

R and Python interfaces have been written by Jean-Paul Chieze (INRIA), and a few contributors have helped us making compilation scripts for various platforms.

2 Installation

The SPAMS toolbox for Python is ditributed in source mode only. It should compile on linux and Mac. The installation procedure is described in the file `INSTALL-package`

- Download the tar gzipped file, unpack it.
- Enter directory spams-python
- execute
  `python setup.py install --prefix=<your-installation-dir>`

You have the choice of the BLAS library, but the Intel MKL is recommended for the best performance.
If you want to add or change libraries, you must modify the file `setup.py`.
The documentation is available in pdf and html format in the doc subdirectory.

3 Dictionary Learning and Matrix Factorization Toolbox

This is the section for dictionary learning and matrix factorization, corresponding to [19, 20].

3.1 Function spams.trainDL

This is the main function of the toolbox, implementing the learning algorithms of [20]. Given a training set $\mathbf{x}^1, \ldots,$. It aims at solving

$$\min_{\mathbf{D}\in\mathcal{C}} \lim_{n\to+\infty} \frac{1}{n}\sum_{i=1}^{n} \min_{\boldsymbol{\alpha}^i} \Big(\frac{1}{2}\|\mathbf{x}^i - \mathbf{D}\boldsymbol{\alpha}^i\|_2^2 + \psi(\boldsymbol{\alpha}^i)\Big). \tag{1}$$

ψ is a sparsity-inducing regularizer and $\mathcal{C}$ is a constraint set for the dictionary. As shown in [20] and in the help file below, various combinations can be used for ψ and $\mathcal{C}$ for solving different matrix factorization problems. What is more, positivity constraints can be added to $\boldsymbol{\alpha}$ as well. The function admits several modes for choosing the optimization parameters, using the parameter-free strategy proposed in [19], or using the parameters t_0 and ρ presented in [20]. **Note that for problems of a reasonable size, and when ψ is the ℓ_1-norm, the function spams.trainDL_Memory can be faster but uses more memory.**

```
#
# Name: trainDL
#
# Usage: spams.trainDL(X,return_model= False,model= None,D = None,numThreads = -1,batchsize =
    -1,
#                K= -1,lambda1= None,lambda2= 10e-10,iter=-1,t0=1e-5,mode=spams_wrap.PENALTY,
#                posAlpha=False,posD=False,expand=False,modeD=spams_wrap.L2,whiten=False,
#                clean=True,verbose=True,gamma1=0.,gamma2=0.,rho=1.0,iter_updateD=None,
#                stochastic_deprecated=False,modeParam=0,batch=False,log_deprecated=False,
#                logName='')
#
# Description:
#     trainDL is an efficient implementation of the
#     dictionary learning technique presented in
#
#     "Online Learning for Matrix Factorization and Sparse Coding"
#     by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro
#     arXiv:0908.0050
#
#     "Online Dictionary Learning for Sparse Coding"
#     by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro
#     ICML 2009.
#
#     Note that if you use mode=1 or 2, if the training set has a
#     reasonable size and you have enough memory on your computer, you
#     should use trainDL_Memory instead.
#
#
#     It addresses the dictionary learning problems
```

```
#        1) if mode=0
#     min_{D in C} (1/n) sum_{i=1}^n (1/2)||x_i-Dalpha_i||_2^2  s.t. ...
#                                                  ||alpha_i||_1 <= lambda1
#        2) if mode=1
#     min_{D in C} (1/n) sum_{i=1}^n  ||alpha_i||_1  s.t.  ...
#                                          ||x_i-Dalpha_i||_2^2 <= lambda1
#        3) if mode=2
#     min_{D in C} (1/n) sum_{i=1}^n (1/2)||x_i-Dalpha_i||_2^2 + ...
#                                 lambda1||alpha_i||_1 + lambda1_2||alpha_i||_2^2
#        4) if mode=3, the sparse coding is done with OMP
#     min_{D in C} (1/n) sum_{i=1}^n (1/2)||x_i-Dalpha_i||_2^2  s.t. ...
#                                                  ||alpha_i||_0 <= lambda1
#        5) if mode=4, the sparse coding is done with OMP
#     min_{D in C} (1/n) sum_{i=1}^n  ||alpha_i||_0  s.t.  ...
#                                          ||x_i-Dalpha_i||_2^2 <= lambda1
#        6) if mode=5, the sparse coding is done with OMP
#     min_{D in C} (1/n) sum_{i=1}^n 0.5||x_i-Dalpha_i||_2^2 +lambda1||alpha_i||_0
#
#
#     C is a convex set verifying
#        1) if modeD=0
#           C={  D in Real^{m x p}  s.t.  forall j,  ||d_j||_2^2 <= 1 }
#        2) if modeD=1
#           C={  D in Real^{m x p}  s.t.  forall j,  ||d_j||_2^2 + ...
#                                                  gamma1||d_j||_1 <= 1 }
#        3) if modeD=2
#           C={  D in Real^{m x p}  s.t.  forall j,  ||d_j||_2^2 + ...
#                                 gamma1||d_j||_1 + gamma2 FL(d_j) <= 1 }
#        4) if modeD=3
#           C={  D in Real^{m x p}  s.t.  forall j,  (1-gamma1)||d_j||_2^2 + ...
#                                 gamma1||d_j||_1 <= 1 }
#
#     Potentially, n can be very large with this algorithm.
#
# Inputs:
#       X:  double m x n matrix   (input signals)
#               m is the signal size
#               n is the number of signals to decompose
#       return_model:
#               if true the function will return the model
#               as a named list ('A' = A, 'B' = B, 'iter' = n)
#       model:        None or model (as A,B,iter) to use as initialisation
#       D: (optional) double m x p matrix   (dictionary)
#         p is the number of elements in the dictionary
#         When D is not provided, the dictionary is initialized
#         with random elements from the training set.
#       K: (size of the dictionary, optional is D is provided)
#       lambda1:  (parameter)
#       lambda2:  (optional, by default 0)
#       iter: (number of iterations).  If a negative number is
#          provided it will perform the computation during the
#          corresponding number of seconds. For instance iter=-5
#          learns the dictionary during 5 seconds.
#       mode: (optional, see above, by default 2)
#       posAlpha: (optional, adds positivity constraints on the
#         coefficients, false by default, not compatible with
#         mode =3,4)
#       modeD: (optional, see above, by default 0)
#       posD: (optional, adds positivity constraints on the
#         dictionary, false by default, not compatible with
#         modeD=2)
#       gamma1: (optional parameter for modeD >= 1)
#       gamma2: (optional parameter for modeD = 2)
#       batchsize: (optional, size of the minibatch, by default
#          512)
#       iter_updateD: (optional, number of BCD iterations for the dictionary
#          update step, by default 1)
```

```
#       modeParam: (optimization mode).
#          1) if modeParam=0, the optimization uses the
#             parameter free strategy of the ICML paper
#          2) if modeParam=1, the optimization uses the
#             parameters rho as in arXiv:0908.0050
#          3) if modeParam=2, the optimization uses exponential
#             decay weights with updates of the form
#            A_{t} <- rho A_{t-1} + alpha_t alpha_t^T
#       rho: (optional) tuning parameter (see paper arXiv:0908.0050)
#       t0: (optional) tuning parameter (see paper arXiv:0908.0050)
#       clean: (optional, true by default. prunes
#          automatically the dictionary from unused elements).
#       verbose: (optional, true by default, increase verbosity)
#       numThreads: (optional, number of threads for exploiting
#          multi-core / multi-cpus. By default, it takes the value -1,
#          which automatically selects all the available CPUs/cores).
#       expand:    undocumented; modify at your own risks!
#       whiten:    undocumented; modify at your own risks!
#       stochastic_deprecated:    undocumented; modify at your own risks!
#       batch:    undocumented; modify at your own risks!
#       log_deprecated:    undocumented; modify at your own risks!
#       logName:    undocumented; modify at your own risks!
#
# Output:
#       D:     double m x p matrix   (dictionary)
#       model:  the model as A B iter
#        D = spams.trainDL(X,return_model = False,...)
#        (D,model) = spams.trainDL(X,return_model = True,...)
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#    this function admits a few experimental usages, which have not
#    been extensively tested:
#        - single precision setting
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
img_file = '../extdata/boat.png'
try:
    img = Image.open(img_file)
except:
    print "Cannot load image %s : skipping test" %img_file
I = np.array(img) / 255.
if I.ndim == 3:
    A = np.asfortranarray(I.reshape((I.shape[0],I.shape[1] * I.shape[2])))
    rgb = True
else:
    A = np.asfortranarray(I)
    rgb = False

m = 8;n = 8;
X = spams.im2col_sliding(A,m,n,rgb)

X = X - np.tile(np.mean(X,0),(X.shape[0],1))
X = np.asfortranarray(X / np.tile(np.sqrt((X * X).sum(axis=0)),(X.shape[0],1)))
param = { 'K' : 100, # learns a dictionary with 100 elements
          'lambda1' : 0.15, 'numThreads' : 4, 'batchsize' : 400,
          'iter' : 1000}

########## FIRST EXPERIMENT ###########
tic = time.time()
D = spams.trainDL(X,**param)
```

```
tac = time.time()
t = tac - tic
print 'time of computation for Dictionary Learning: %f' %t

##param['approx'] = 0
print 'Evaluating cost function...'
lparam = _extract_lasso_param(param)
alpha = spams.lasso(X,D = D,**lparam)
xd = X - D * alpha
R = np.mean(0.5 * (xd * xd).sum(axis=0) + param['lambda1'] * np.abs(alpha).sum(axis=0))

print "objective function: %f" %R

#### SECOND EXPERIMENT ####
print "********** SECOND EXPERIMENT **********"

X1 = X[:,0:X.shape[1]/2]
X2 = X[:,X.shape[1]/2 -1:]
param['iter'] = 500
tic = time.time()
(D,model) = spams.trainDL(X1,return_model = True,**param)
tac = time.time()
t = tac - tic
print 'time of computation for Dictionary Learning: %f\n' %t
print 'Evaluating cost function...'
alpha = spams.lasso(X,D = D,**lparam)
xd = X - D * alpha
R = np.mean(0.5 * (xd * xd).sum(axis=0) + param['lambda1'] * np.abs(alpha).sum(axis=0))
print "objective function: %f" %R

# Then reuse the learned model to retrain a few iterations more.
param2 = param.copy()
param2['D'] = D
tic = time.time()
(D,model) = spams.trainDL(X2,return_model = True,model = model,**param2)
tac = time.time()
t = tac - tic
print 'time of computation for Dictionary Learning: %f' %t
print 'Evaluating cost function...'
alpha = spams.lasso(X,D = D,**lparam)
xd = X - D * alpha
R = np.mean(0.5 * (xd * xd).sum(axis=0) + param['lambda1'] * np.abs(alpha).sum(axis=0))
print "objective function: %f" %R

###################### THIRD & FOURTH EXPERIMENT ######################
# let us add sparsity to the dictionary itself

print '********** THIRD EXPERIMENT **********'
param['modeParam'] = 0
param['iter'] = 1000
param['gamma1'] = 0.3
param['modeD'] = 1

tic = time.time()
D = spams.trainDL(X,**param)
tac = time.time()
t = tac - tic
print 'time of computation for Dictionary Learning: %f' %t
print 'Evaluating cost function...'
alpha = spams.lasso(X,D = D,**lparam)
xd = X - D * alpha
R = np.mean(0.5 * (xd * xd).sum(axis=0) + param['lambda1'] * np.abs(alpha).sum(axis=0))
print "objective function: %f" %R

print '********** FOURTH EXPERIMENT **********'
param['modeParam'] = 0
param['iter'] = 1000
```

```
param['gamma1'] = 0.3
param['modeD'] = 3

tic = time.time()
D = spams.trainDL(X,**param)
tac = time.time()
t = tac - tic
print 'time of computation for Dictionary Learning: %f' %t
print 'Evaluating cost function...'
alpha = spams.lasso(X,D = D,**lparam)
xd = X - D * alpha
R = np.mean(0.5 * (xd * xd).sum(axis=0) + param['lambda1'] * np.abs(alpha).sum(axis=0))
print "objective function: %f" %R
```

3.2 Function spams.trainDL_Memory

Memory-consuming version of spams.trainDL. This function is well adapted to small/medium-size problems: It requires storing all the coefficients $\boldsymbol{\alpha}$ and is therefore impractical for very large datasets. However, in many situations, one can afford this memory cost and it is better to use this method, which is faster than spams.trainDL. Note that unlike spams.trainDL this function does not allow warm-restart.

```
#
# Name: trainDL_Memory
#
# Usage: spams.trainDL_Memory(X,D = None,numThreads = -1,batchsize = -1,K= -1,lambda1= None,
    iter=-1,
#                         t0=1e-5,mode=spams_wrap.PENALTY,posD=False,expand=False,
#                         modeD=spams_wrap.L2,whiten=False,clean=True,gamma1=0.,gamma2=0.,
#                         rho=1.0,iter_updateD=1,stochastic_deprecated=False,modeParam=0,
#                         batch=False,log_deprecated=False,logName='')
#
# Description:
#     trainDL_Memory is an efficient but memory consuming
#     variant of the dictionary learning technique presented in
#
#     "Online Learning for Matrix Factorization and Sparse Coding"
#     by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro
#     arXiv:0908.0050
#
#     "Online Dictionary Learning for Sparse Coding"
#     by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro
#     ICML 2009.
#
#     Contrary to the approaches above, the algorithm here
#        does require to store all the coefficients from all the training
#        signals. For this reason this variant can not be used with large
#        training sets, but is more efficient than the regular online
#        approach for training sets of reasonable size.
#
#     It addresses the dictionary learning problems
#        1) if mode=1
#     min_{D in C} (1/n) sum_{i=1}^n  ||alpha_i||_1  s.t.  ...
#                                          ||x_i-Dalpha_i||_2^2 <= lambda1
#        2) if mode=2
#     min_{D in C} (1/n) sum_{i=1}^n (1/2)||x_i-Dalpha_i||_2^2 + ...
#                                                    lambda1||alpha_i||_1
#
#     C is a convex set verifying
#        1) if modeD=0
#           C={  D in Real^{m x p}  s.t.  forall j,  ||d_j||_2^2 <= 1 }
#        1) if modeD=1
#           C={  D in Real^{m x p}  s.t.  forall j,  ||d_j||_2^2 + ...
#                                                 gamma1||d_j||_1 <= 1 }
#        1) if modeD=2
#           C={  D in Real^{m x p}  s.t.  forall j,  ||d_j||_2^2 + ...
#                                  gamma1||d_j||_1 + gamma2 FL(d_j) <= 1 }
```

```
#
#     Potentially, n can be very large with this algorithm.
#
# Inputs:
#       X:  double m x n matrix   (input signals)
#               m is the signal size
#               n is the number of signals to decompose
#       D: (optional) double m x p matrix   (dictionary)
#         p is the number of elements in the dictionary
#         When D is not provided, the dictionary is initialized
#         with random elements from the training set.
#       K: (size of the dictionary, optional is D is provided)
#       lambda1:  (parameter)
#       iter: (number of iterations).  If a negative number is
#          provided it will perform the computation during the
#          corresponding number of seconds. For instance iter=-5
#          learns the dictionary during 5 seconds.
#       mode: (optional, see above, by default 2)
#       modeD: (optional, see above, by default 0)
#       posD: (optional, adds positivity constraints on the
#         dictionary, false by default, not compatible with
#         modeD=2)
#       gamma1: (optional parameter for modeD >= 1)
#       gamma2: (optional parameter for modeD = 2)
#       batchsize: (optional, size of the minibatch, by default
#         512)
#       iter_updateD: (optional, number of BCD iterations for the dictionary
#           update step, by default 1)
#       modeParam: (optimization mode).
#         1) if modeParam=0, the optimization uses the
#            parameter free strategy of the ICML paper
#         2) if modeParam=1, the optimization uses the
#            parameters rho as in arXiv:0908.0050
#         3) if modeParam=2, the optimization uses exponential
#            decay weights with updates of the form
#            A_{t} <- rho A_{t-1} + alpha_t alpha_t^T
#       rho: (optional) tuning parameter (see paper arXiv:0908.0050)
#       t0: (optional) tuning parameter (see paper arXiv:0908.0050)
#       clean: (optional, true by default. prunes
#         automatically the dictionary from unused elements).
#       numThreads: (optional, number of threads for exploiting
#         multi-core / multi-cpus. By default, it takes the value -1,
#         which automatically selects all the available CPUs/cores).
#       expand:    undocumented; modify at your own risks!
#       whiten:    undocumented; modify at your own risks!
#       stochastic_deprecated:    undocumented; modify at your own risks!
#       batch:    undocumented; modify at your own risks!
#       log_deprecated:    undocumented; modify at your own risks!
#       logName:    undocumented; modify at your own risks!
#
# Output:
#       D:     double m x p matrix   (dictionary)
#       model:  the model as A B iter
#        D = spams.trainDL_Memory(X,...)
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     this function admits a few experimental usages, which have not
#     been extensively tested:
#         - single precision setting (even though the output alpha is double
#           precision)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
img_file = '../extdata/lena.png'
try:
    img = Image.open(img_file)
except:
    print "Cannot load image %s : skipping test" %img_file
I = np.array(img) / 255.
if I.ndim == 3:
    A = np.asfortranarray(I.reshape((I.shape[0],I.shape[1] * I.shape[2])))
    rgb = True
else:
    A = np.asfortranarray(I)
    rgb = False

m = 8;n = 8;
X = spams.im2col_sliding(A,m,n,rgb)

X = X - np.tile(np.mean(X,0),(X.shape[0],1))
X = np.asfortranarray(X / np.tile(np.sqrt((X * X).sum(axis=0)),(X.shape[0],1)))
X = np.asfortranarray(X[:,np.arange(0,X.shape[1],10)])

param = { 'K' : 200, # learns a dictionary with 100 elements
      'lambda1' : 0.15, 'numThreads' : 4,
      'iter' : 100}

############## FIRST EXPERIMENT ###################
tic = time.time()
D = spams.trainDL_Memory(X,**param)
tac = time.time()
t = tac - tic
print 'time of computation for Dictionary Learning: %f' %t

print 'Evaluating cost function...'
lparam = _extract_lasso_param(param)
alpha = spams.lasso(X,D = D,**lparam)
xd = X - D * alpha
R = np.mean(0.5 * (xd * xd).sum(axis=0) + param['lambda1'] * np.abs(alpha).sum(axis=0))
print "objective function: %f" %R

############## SECOND EXPERIMENT ###################
tic = time.time()
D = spams.trainDL(X,**param)
tac = time.time()
t = tac - tic
print 'time of computation for Dictionary Learning: %f' %t
print 'Evaluating cost function...'
alpha = spams.lasso(X,D = D,**lparam)
xd = X - D * alpha
R = np.mean(0.5 * (xd * xd).sum(axis=0) + param['lambda1'] * np.abs(alpha).sum(axis=0))
print "objective function: %f" %R
```

3.3 Function nmf

This function is an example on how to use the function spams.trainDL for the problem of non-negative matrix factorization formulated in [16]. Note that spams.trainDL can be replaced by spams.trainDL_Memory in this function for small or medium datasets.

```
#
# Name: nmf
#
# Usage: spams.nmf(X,return_lasso= False,model= None,numThreads = -1,batchsize = -1,K= -1,iter
    =-1,
#            t0=1e-5,clean=True,rho=1.0,modeParam=0,batch=False)
#
```

```
# Description:
#     trainDL is an efficient implementation of the
#     non-negative matrix factorization technique presented in
#
#     "Online Learning for Matrix Factorization and Sparse Coding"
#     by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro
#     arXiv:0908.0050
#
#     "Online Dictionary Learning for Sparse Coding"
#     by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro
#     ICML 2009.
#
#     Potentially, n can be very large with this algorithm.
#
# Inputs:
#       X:  double m x n matrix   (input signals)
#                m is the signal size
#                n is the number of signals to decompose
#       return_lasso:
#                          if true the function will return a tuple of matrices.
#       K: (number of required factors)
#       iter: (number of iterations).  If a negative number
#         is provided it will perform the computation during the
#         corresponding number of seconds. For instance iter=-5
#         learns the dictionary during 5 seconds.
#       batchsize: (optional, size of the minibatch, by default
#          512)
#       modeParam: (optimization mode).
#          1) if modeParam=0, the optimization uses the
#             parameter free strategy of the ICML paper
#          2) if modeParam=1, the optimization uses the
#             parameters rho as in arXiv:0908.0050
#          3) if modeParam=2, the optimization uses exponential
#             decay weights with updates of the form
#             A_{t} <- rho A_{t-1} + alpha_t alpha_t^T
#       rho: (optional) tuning parameter (see paper
#       arXiv:0908.0050)
#       t0: (optional) tuning parameter (see paper
#       arXiv:0908.0050)
#       clean: (optional, true by default. prunes automatically
#         the dictionary from unused elements).
#       batch: (optional, false by default, use batch learning
#         instead of online learning)
#       numThreads: (optional, number of threads for exploiting
#            multi-core / multi-cpus. By default, it takes the value -1,
#            which automatically selects all the available CPUs/cores).
#       model: struct (optional) learned model for "retraining" the data.
#
# Output:
#       U: double m x p matrix
#       V: double p x n matrix   (optional)
#       model: struct (optional) learned model to be used for
#         "retraining" the data.
#         U = spams.nmf(X,return_lasso = False,...)
#     (U,V) = spams.nmf(X,return_lasso = True,...)
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
img_file = '../extdata/boat.png'
try:
    img = Image.open(img_file)
```

```
except:
    print "Cannot load image %s : skipping test" %img_file
I = np.array(img) / 255.
if I.ndim == 3:
    A = np.asfortranarray(I.reshape((I.shape[0],I.shape[1] * I.shape[2])))
    rgb = True
else:
    A = np.asfortranarray(I)
    rgb = False

m = 16;n = 16;
X = spams.im2col_sliding(A,m,n,rgb)
X = X[:,::10]
X = np.asfortranarray(X / np.tile(np.sqrt((X * X).sum(axis=0)),(X.shape[0],1)))
########## FIRST EXPERIMENT ############
tic = time.time()
(U,V) = spams.nmf(X,return_lasso= True,K = 49,numThreads=4,iter = -5)
tac = time.time()
t = tac - tic
print 'time of computation for Dictionary Learning: %f' %t

print 'Evaluating cost function...'
Y = X - U * V
R = np.mean(0.5 * (Y * Y).sum(axis=0))
print 'objective function: %f' %R
```

3.4 Function nnsc

This function is an example on how to use the function spams.trainDL for the problem of non-negative sparse coding as defined in [13]. Note that spams.trainDL can be replaced by spams.trainDL_Memory in this function for small or medium datasets.

```
#
# Name: nmf
#
# Usage: spams.nnsc(X,return_lasso= False,model= None,lambda1= None,numThreads = -1,batchsize =
    -1,
#            K= -1,iter=-1,t0=1e-5,clean=True,rho=1.0,modeParam=0,batch=False)
#
# Description:
#     trainDL is an efficient implementation of the
#     non-negative sparse coding technique presented in
#
#     "Online Learning for Matrix Factorization and Sparse Coding"
#     by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro
#     arXiv:0908.0050
#
#     "Online Dictionary Learning for Sparse Coding"
#     by Julien Mairal, Francis Bach, Jean Ponce and Guillermo Sapiro
#     ICML 2009.
#
#     Potentially, n can be very large with this algorithm.
#
# Inputs:
#       X:  double m x n matrix   (input signals)
#               m is the signal size
#               n is the number of signals to decompose
#       return_lasso:
#                         if true the function will return a tuple of matrices.
#       K: (number of required factors)
#       lambda1: (parameter)
#       iter: (number of iterations).  If a negative number
#          is provided it will perform the computation during the
#          corresponding number of seconds. For instance iter=-5
#          learns the dictionary during 5 seconds.
#       batchsize: (optional, size of the minibatch, by default
```

```
#          512)
#       modeParam: (optimization mode).
#          1) if modeParam=0, the optimization uses the
#             parameter free strategy of the ICML paper
#          2) if modeParam=1, the optimization uses the
#             parameters rho as in arXiv:0908.0050
#          3) if modeParam=2, the optimization uses exponential
#             decay weights with updates of the form
#             A_{t} <- rho A_{t-1} + alpha_t alpha_t^T
#       rho: (optional) tuning parameter (see paper
#       arXiv:0908.0050)
#       t0: (optional) tuning parameter (see paper
#       arXiv:0908.0050)
#       clean: (optional, true by default. prunes automatically
#          the dictionary from unused elements).
#       batch: (optional, false by default, use batch learning
#          instead of online learning)
#       numThreads: (optional, number of threads for exploiting
#          multi-core / multi-cpus. By default, it takes the value -1,
#          which automatically selects all the available CPUs/cores).
#       model: struct (optional) learned model for "retraining" the data.
#
# Output:
#       U: double m x p matrix
#       V: double p x n matrix    (optional)
#       model: struct (optional) learned model to be used for
#          "retraining" the data.
#          U = spams.nnsc(X,return_lasso = False,...)
#       (U,V) = spams.nnsc(X,return_lasso = True,...)
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

4 Sparse Decomposition Toolbox

This toolbox implements several algorithms for solving signal reconstruction problems. It is mostly adapted for solving a large number of small/medium scale problems, but can be also efficient sometimes with large scale ones.

4.1 Function spams.omp

This is a fast implementation of the Orthogonal Matching Pursuit algorithm (or forward selection) [24, 32]. Given a matrix of signals $\mathbf{X} = [\mathbf{x}^1, \ldots, \mathbf{x}^n]$ in $\mathbb{R}^{m \times n}$ and a dictionary $\mathbf{D} = [\mathbf{d}^1, \ldots, \mathbf{d}^p]$ in $\mathbb{R}^{m \times p}$, the algorithm computes a matrix $\mathbf{A} = [\boldsymbol{\alpha}^1, \ldots, \boldsymbol{\alpha}^n]$ in $\mathbb{R}^{p \times n}$, where for each column $\mathbf{x}$ of $\mathbf{X}$, it returns a coefficient vector $\boldsymbol{\alpha}$ which is an approximate solution of the following NP-hard problem

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 \quad \text{s.t.} \quad \|\boldsymbol{\alpha}\|_0 \leq L, \tag{2}$$

or

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \|\boldsymbol{\alpha}\|_0 \quad \text{s.t.} \quad \|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 \leq \varepsilon, \tag{3}$$

or

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \frac{1}{2}\|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 + \lambda \|\boldsymbol{\alpha}\|_0. \tag{4}$$

For efficienty reasons, the method first computes the covariance matrix $\mathbf{D}^T\mathbf{D}$, then for each signal, it computes $\mathbf{D}^T\mathbf{x}$ and performs the decomposition with a Cholesky-based algorithm (see [6] for instance).

Note that spams.omp can return the "greedy" regularization path if needed (see below):

```
#
# Name: omp
```

```
#
# Usage: spams.omp(X,D,L=None,eps= None,lambda1 = None,return_reg_path = False,numThreads = -1)
#
# Description:
#     omp is an efficient implementation of the
#     Orthogonal Matching Pursuit algorithm. It is optimized
#     for solving a large number of small or medium-sized
#     decomposition problem (and not for a single large one).
#     It first computes the Gram matrix D'D and then perform
#     a Cholesky-based OMP of the input signals in parallel.
#     X=[x^1,...,x^n] is a matrix of signals, and it returns
#     a matrix A=[alpha^1,...,alpha^n] of coefficients.
#
#     it addresses for all columns x of X,
#         min_{alpha} ||alpha||_0  s.t  ||x-Dalpha||_2^2 <= eps
#         or
#         min_{alpha} ||x-Dalpha||_2^2  s.t. ||alpha||_0 <= L
#         or
#         min_{alpha} 0.5||x-Dalpha||_2^2 + lambda1||alpha||_0
#
# Inputs:
#       X:  double m x n matrix   (input signals)
#           m is the signal size
#           n is the number of signals to decompose
#       D:  double m x p matrix   (dictionary)
#          p is the number of elements in the dictionary
#          All the columns of D should have unit-norm !
#       return_reg_path:
#                           if true the function will return a tuple of matrices.
#       L: (optional, maximum number of elements in each decomposition,
#          min(m,p) by default)
#       eps: (optional, threshold on the squared l2-norm of the residual,
#          0 by default
#       lambda1: (optional, penalty parameter, 0 by default
#       numThreads: (optional, number of threads for exploiting
#       multi-core / multi-cpus. By default, it takes the value -1,
#       which automatically selects all the available CPUs/cores).
#
# Output:
#       A: double sparse p x n matrix (output coefficients)
#         path (optional): double dense p x L matrix (regularization path of the first signal)
#         A = spams.omp(X,D,L,eps,return_reg_path = False,...)
#     (A,path) = spams.omp(X,D,L,eps,return_reg_path = True,...)
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     this function admits a few experimental usages, which have not
#     been extensively tested:
#      - single precision setting (even though the output alpha is double
#        precision)
#      - Passing an int32 vector of length n to L provides
#        a different parameter L for each input signal x_i
#      - Passing a double vector of length n to eps and or lambda1
#        provides a different parameter eps (or lambda1) for each input signal x_i
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
print 'test omp'
X = np.asfortranarray(np.random.normal(size=(64,100000)))
D = np.asfortranarray(np.random.normal(size=(64,200)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
```

```
L = 10
eps = 0.1
numThreads = -1
tic = time.time()
alpha = spams.omp(X,D,L=L,eps= eps,return_reg_path = False,numThreads = numThreads)
tac = time.time()
t = tac - tic
print "%f signals processed per second\n" %(float(X.shape[1]) / t)
##########################################
# Regularization path of a single signal
##########################################
X = np.asfortranarray(np.random.normal(size=(64,1)))
D = np.asfortranarray(np.random.normal(size=(64,10)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
L = 5
(alpha,path) = spams.omp(X,D,L=L,eps= eps,return_reg_path = True,numThreads = numThreads)
```

4.2 Function spams.ompMask

This is a variant of spams.omp with the possibility of handling a binary mask. Given a binary mask $\mathbf{B} = [\boldsymbol{\beta}^1, \ldots, \boldsymbol{\beta}^n]$ in $\{0,1\}^{m \times n}$, it returns a matrix $\mathbf{A} = [\boldsymbol{\alpha}^1, \ldots, \boldsymbol{\alpha}^n]$ such that for every column $\mathbf{x}$ of $\mathbf{X}$, $\boldsymbol{\beta}$ of $\mathbf{B}$, it computes a column $\boldsymbol{\alpha}$ of $\mathbf{A}$ by addressing

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \|\operatorname{diag}(\boldsymbol{\beta})(\mathbf{x} - \mathbf{D}\boldsymbol{\alpha})\|_2^2 \quad \text{s.t.} \quad \|\boldsymbol{\alpha}\|_0 \leq L, \tag{5}$$

or

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \|\boldsymbol{\alpha}\|_0 \quad \text{s.t.} \quad \|\operatorname{diag}(\boldsymbol{\beta})(\mathbf{x} - \mathbf{D}\boldsymbol{\alpha})\|_2^2 \leq \varepsilon \frac{\|\boldsymbol{\beta}\|_0}{m}, \tag{6}$$

or

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \frac{1}{2}\|\operatorname{diag}(\boldsymbol{\beta})(\mathbf{x} - \mathbf{D}\boldsymbol{\alpha})\|_2^2 + \lambda\|\boldsymbol{\alpha}\|_0. \tag{7}$$

where $\operatorname{diag}(\boldsymbol{\beta})$ is a diagonal matrix with the entries of $\boldsymbol{\beta}$ on the diagonal.

```
#
# Name: ompMask
#
# Usage: spams.ompMask(X,D,B,L=None,eps= None,lambda1 = None,return_reg_path = False,
#                numThreads = -1)
#
# Description:
#     ompMask is a variant of mexOMP that allow using
#     a binary mask B
#
#     for all columns x of X, and columns beta of B, it computes a column
#         alpha of A by addressing
#         min_{alpha} ||alpha||_0  s.t  ||diag(beta)*(x-Dalpha)||_2^2
#                                                 <= eps*||beta||_0/m
#         or
#         min_{alpha} ||diag(beta)*(x-Dalpha)||_2^2  s.t. ||alpha||_0 <= L
#         or
#         min_{alpha} 0.5||diag(beta)*(x-Dalpha)||_2^2  + lambda1||alpha||_0
#
# Inputs:
#       X:  double m x n matrix   (input signals)
#           m is the signal size
#           n is the number of signals to decompose
#       D:  double m x p matrix   (dictionary)
#          p is the number of elements in the dictionary
#          All the columns of D should have unit-norm !
#       B:  boolean m x n matrix   (mask)
#            p is the number of elements in the dictionary
#       return_reg_path:
#                          if true the function will return a tuple of matrices.
#       L: (optional, maximum number of elements in each decomposition,
#          min(m,p) by default)
```

```
#       eps: (optional, threshold on the squared l2-norm of the residual,
#          0 by default
#       lambda1: (optional, penalty parameter, 0 by default
#       numThreads: (optional, number of threads for exploiting
#       multi-core / multi-cpus. By default, it takes the value -1,
#       which automatically selects all the available CPUs/cores).
#
# Output:
#       A: double sparse p x n matrix (output coefficients)
#         path (optional): double dense p x L matrix
#                                    (regularization path of the first signal)
#                                    A = spams.ompMask(X,D,B,L,eps,return_reg_path = False
    ,...)
#                                (A,path) = spams.ompMask(X,D,B,L,eps,return_reg_path = True
    ,...)
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     this function admits a few experimental usages, which have not
#     been extensively tested:
#      - single precision setting (even though the output alpha is double
#        precision)
#      - Passing an int32 vector of length n to L provides
#        a different parameter L for each input signal x_i
#      - Passing a double vector of length n to eps and or lambda1
#        provides a different parameter eps (or lambda1) for each input signal x_i
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
print 'test ompMask'

########################################
# Decomposition of a large number of signals
########################################
X = np.asfortranarray(np.random.normal(size=(300,300)))
X = np.asfortranarray(X / np.tile(np.sqrt((X*X).sum(axis=0)),(X.shape[0],1)))
D = np.asfortranarray(np.random.normal(size=(300,50)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
mask = np.asfortranarray((X > 0))  # generating a binary mask
L = 20
eps = 0.1
numThreads=-1
tic = time.time()
alpha = spams.ompMask(X,D,mask,L = L,eps = eps,return_reg_path = False,numThreads = numThreads)
tac = time.time()
t = tac - tic
print "%f signals processed per second\n" %(float(X.shape[1]) / t)
```

4.3 Function spams.lasso

This is a fast implementation of the LARS algorithm [8] (variant for solving the Lasso) for solving the Lasso or Elastic-Net. Given a matrix of signals $\mathbf{X} = [\mathbf{x}^1, \ldots, \mathbf{x}^n]$ in $\mathbb{R}^{m \times n}$ and a dictionary $\mathbf{D}$ in $\mathbb{R}^{m \times p}$, depending on the input parameters, the algorithm returns a matrix of coefficients $\mathbf{A} = [\boldsymbol{\alpha}^1, \ldots, \boldsymbol{\alpha}^n]$ in $\mathbb{R}^{p \times n}$ such that for every column $\mathbf{x}$ of $\mathbf{X}$, the corresponding column $\boldsymbol{\alpha}$ of $\mathbf{A}$ is the solution of

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 \quad \text{s.t.} \quad \|\boldsymbol{\alpha}\|_1 \leq \lambda, \tag{8}$$

or

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \|\boldsymbol{\alpha}\|_1 \quad \text{s.t.} \quad \|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 \leq \lambda, \tag{9}$$

or

$$\min_{\boldsymbol{\alpha}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{x}-\mathbf{D}\boldsymbol{\alpha}\|_2^2 + \lambda\|\boldsymbol{\alpha}\|_1 + \frac{\lambda_2}{2}\|\boldsymbol{\alpha}\|_2^2. \tag{10}$$

For efficiency reasons, the method first compute the covariance matrix $\mathbf{D}^T\mathbf{D}$, then for each signal, it computes $\mathbf{D}^T\mathbf{x}$ and performs the decomposition with a Cholesky-based algorithm (see [8] for instance). The implementation has also an option to add **positivity constraints** on the solutions $\boldsymbol{\alpha}$. When the solution is very sparse and the problem size is reasonable, this approach can be very efficient. Moreover, it gives the solution with an exact precision, and its performance does not depend on the correlation of the dictionary elements, except when the solution is not unique (the algorithm breaks in this case).

Note that spams.lasso can return the whole regularization path of the first signal $\mathbf{x}_1$ and can handle implicitely the matrix $\mathbf{D}$ if the quantities $\mathbf{D}^T\mathbf{D}$ and $\mathbf{D}^T\mathbf{x}$ are passed as an argument, see below:

```
#
# Name: lasso
#
# Usage: spams.lasso(X,D= None,Q = None,q = None,return_reg_path = False,L= -1,lambda1= None,
#             lambda2= 0.,mode= spams_wrap.PENALTY,pos= False,ols= False,numThreads= -1,
#             max_length_path= -1,verbose=False,cholesky= False)
#
# Description:
#     lasso is an efficient implementation of the
#     homotopy-LARS algorithm for solving the Lasso.
#
#     If the function is called this way spams.lasso(X,D = D, Q = None,...),
#     it aims at addressing the following problems
#     for all columns x of X, it computes one column alpha of A
#     that solves
#       1) when mode=0
#         min_{alpha} ||x-Dalpha||_2^2 s.t. ||alpha||_1 <= lambda1
#       2) when mode=1
#         min_{alpha} ||alpha||_1 s.t. ||x-Dalpha||_2^2 <= lambda1
#       3) when mode=2
#         min_{alpha} 0.5||x-Dalpha||_2^2 + lambda1||alpha||_1 +0.5 lambda2||alpha||_2^2
#
#     If the function is called this way spams.lasso(X,D = None, Q = Q, q = q,...),
#     it solves the above optimisation problem, when Q=D'D and q=D'x.
#
#     Possibly, when pos=true, it solves the previous problems
#     with positivity constraints on the vectors alpha
#
# Inputs:
#       X:  double m x n matrix   (input signals)
#               m is the signal size
#               n is the number of signals to decompose
#       D:  double m x p matrix   (dictionary)
#             p is the number of elements in the dictionary
#       Q:              p x p double matrix (Q = D'D)
#       q:              p x n double matrix (q = D'X)
#       verbose:        verbose mode
#       return_reg_path:
#                           if true the function will return a tuple of matrices.
#       lambda1:  (parameter)
#       lambda2:  (optional parameter for solving the Elastic-Net)
#                     for mode=0 and mode=1, it adds a ridge on the Gram Matrix
#       L: (optional), maximum number of steps of the homotopy algorithm (can
#                be used as a stopping criterion)
#       pos: (optional, adds non-negativity constraints on the
#         coefficients, false by default)
#       mode: (see above, by default: 2)
#       numThreads: (optional, number of threads for exploiting
#         multi-core / multi-cpus. By default, it takes the value -1,
#         which automatically selects all the available CPUs/cores).
#       cholesky: (optional, default false),  choose between Cholesky
#         implementation or one based on the matrix inversion Lemma
#       ols: (optional, default false), perform an orthogonal projection
```

```
#         before returning the solution.
#       max_length_path: (optional) maximum length of the path, by default 4*p
#
# Output:
#       A: double sparse p x n matrix (output coefficients)
#       path: optional,  returns the regularisation path for the first signal
#       A = spams.lasso(X,return_reg_path = False,...)
#   (A,path) = spams.lasso(X,return_reg_path = True,...)
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     this function admits a few experimental usages, which have not
#     been extensively tested:
#         - single precision setting (even though the output alpha is double
#           precision)
#
# Examples:
#       import numpy as np
#       m = 5;n = 10;nD = 5
#       np.random.seed(0)
#       X = np.asfortranarray(np.random.normal(size=(m,n)))
#       X = np.asfortranarray(X / np.tile(np.sqrt((X*X).sum(axis=0)),(X.shape[0],1)))
#       D = np.asfortranarray(np.random.normal(size=(100,200)))
#       D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
#       alpha = spams.lasso(X,D = D,return_reg_path = FALSE,lambda1 = 0.15)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
print "test lasso"
##################################################
# Decomposition of a large number of signals
##################################################
# data generation
X = np.asfortranarray(np.random.normal(size=(100,100000)))
X = np.asfortranarray(X / np.tile(np.sqrt((X*X).sum(axis=0)),(X.shape[0],1)))
D = np.asfortranarray(np.random.normal(size=(100,200)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
# parameter of the optimization procedure are chosen
#param.L=20; # not more than 20 non-zeros coefficients (default: min(size(D,1),size(D,2)))
param = {
    'lambda1' : 0.15, # not more than 20 non-zeros coefficients
    'numThreads' : -1, # number of processors/cores to use; the default choice is -1
    # and uses all the cores of the machine
    'mode' : spams.PENALTY}        # penalized formulation

tic = time.time()
alpha = spams.lasso(X,D = D,return_reg_path = False,**param)
tac = time.time()
t = tac - tic
print "%f signals processed per second\n" %(float(X.shape[1]) / t)
##########################################
# Regularization path of a single signal
##########################################
X = np.asfortranarray(np.random.normal(size=(64,1)))
D = np.asfortranarray(np.random.normal(size=(64,10)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
(alpha,path) = spams.lasso(X,D = D,return_reg_path = True,**param)
```

4.4 Function spams.lassoWeighted

This is a fast implementation of a weighted version of LARS [8]. Given a matrix of signals $\mathbf{X} = [\mathbf{x}^1, \ldots, \mathbf{x}^n]$ in $\mathbb{R}^{m \times n}$, a matrix of weights $\mathbf{W} = [\mathbf{w}^1, \ldots, \mathbf{w}^n] \in \mathbb{R}^{p \times n}$, and a dictionary $\mathbf{D}$ in $\mathbb{R}^{m \times p}$, depending on the input parameters, the algorithm returns a matrix of coefficients $\mathbf{A} = [\boldsymbol{\alpha}^1, \ldots, \boldsymbol{\alpha}^n]$ in $\mathbb{R}^{p \times n}$, such that for every column $\mathbf{x}$ of $\mathbf{X}$, $\mathbf{w}$ of $\mathbf{W}$, it computes a column $\boldsymbol{\alpha}$ of $\mathbf{A}$, which is the solution of

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 \quad \text{s.t.} \quad \|\operatorname{diag}(\mathbf{w})\boldsymbol{\alpha}\|_1 \leq \lambda, \tag{11}$$

or

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \|\operatorname{diag}(\mathbf{w})\boldsymbol{\alpha}\|_1 \quad \text{s.t.} \quad \|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 \leq \lambda, \tag{12}$$

or

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \frac{1}{2}\|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 + \lambda \|\operatorname{diag}(\mathbf{w})\boldsymbol{\alpha}\|_1. \tag{13}$$

The implementation has also an option to add **positivity constraints** on the solutions $\boldsymbol{\alpha}$. This function is potentially useful for implementing efficiently the randomized Lasso of [25], or reweighted-ℓ_1 schemes [4].

```
#
# Name: lassoWeighted.
#
# Usage: spams.lassoWeighted(X,D,W,L= -1,lambda1= None,mode= spams_wrap.PENALTY,pos= False,
#                      numThreads= -1,verbose = False)
#
# Description:
#     lassoWeighted is an efficient implementation of the
#     LARS algorithm for solving the weighted Lasso. It is optimized
#     for solving a large number of small or medium-sized
#     decomposition problem (and not for a single large one).
#     It first computes the Gram matrix D'D and then perform
#     a Cholesky-based OMP of the input signals in parallel.
#     For all columns x of X, and w of W, it computes one column alpha of A
#     which is the solution of
#       1) when mode=0
#         min_{alpha} ||x-Dalpha||_2^2   s.t.
#                                    ||diag(w)alpha||_1 <= lambda1
#       2) when mode=1
#         min_{alpha} ||diag(w)alpha||_1  s.t.
#                                      ||x-Dalpha||_2^2 <= lambda1
#       3) when mode=2
#         min_{alpha} 0.5||x-Dalpha||_2^2  +
#                                       lambda1||diag(w)alpha||_1
#     Possibly, when pos=true, it solves the previous problems
#     with positivity constraints on the vectors alpha
#
# Inputs:
#       X:  double m x n matrix   (input signals)
#               m is the signal size
#               n is the number of signals to decompose
#       D:  double m x p matrix   (dictionary)
#             p is the number of elements in the dictionary
#       W:  double p x n matrix   (weights)
#       verbose:       verbose mode
#       lambda1:  (parameter)
#       L: (optional, maximum number of elements of each
#       decomposition)
#       pos: (optional, adds positivity constraints on the
#       coefficients, false by default)
#       mode: (see above, by default: 2)
#       numThreads: (optional, number of threads for exploiting
#       multi-core / multi-cpus. By default, it takes the value -1,
#       which automatically selects all the available CPUs/cores).
#
# Output:
#       A: double sparse p x n matrix (output coefficients)
#
```

```
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     this function admits a few experimental usages, which have not
#     been extensively tested:
#         - single precision setting (even though the output alpha is double
#           precision)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
print "test lasso weighted"
################################################
# Decomposition of a large number of signals
################################################
# data generation
X = np.asfortranarray(np.random.normal(size=(64,10000)))
X = np.asfortranarray(X / np.tile(np.sqrt((X*X).sum(axis=0)),(X.shape[0],1)))
D = np.asfortranarray(np.random.normal(size=(64,256)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
param = { 'L' : 20,
    'lambda1' : 0.15, 'numThreads' : 8, 'mode' : spams.PENALTY}
W = np.asfortranarray(np.random.random(size = (D.shape[1],X.shape[1])))
tic = time.time()
alpha = spams.lassoWeighted(X,D,W,**param)
tac = time.time()
t = tac - tic
print "%f signals processed per second\n" %(float(X.shape[1]) / t)
```

4.5 Function spams.lassoMask

This is a variant of spams.lasso with the possibility of adding a mask $\mathbf{B} = [\boldsymbol{\beta}^1, \ldots, \boldsymbol{\beta}^n]$, as in mexOMPMask. For every column $\mathbf{x}$ of $\mathbf{X}$, $\boldsymbol{\beta}$ of $\mathbf{B}$, it computes a column $\boldsymbol{\alpha}$ of $\mathbf{A}$, which is the solution of

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \| \operatorname{diag}(\boldsymbol{\beta})(\mathbf{x} - \mathbf{D}\boldsymbol{\alpha})\|_2^2 \quad \text{s.t.} \quad \|\boldsymbol{\alpha}\|_1 \leq \lambda, \tag{14}$$

or

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \|\boldsymbol{\alpha}\|_1 \quad \text{s.t.} \quad \| \operatorname{diag}(\boldsymbol{\beta})(\mathbf{x} - \mathbf{D}\boldsymbol{\alpha})\|_2^2 \leq \lambda \frac{\|\boldsymbol{\beta}\|_0}{m}, \tag{15}$$

or

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^p} \frac{1}{2} \| \operatorname{diag}(\boldsymbol{\beta})(\mathbf{x} - \mathbf{D}\boldsymbol{\alpha})\|_2^2 + \lambda \frac{\|\boldsymbol{\beta}\|_0}{m} \|\boldsymbol{\alpha}\|_1 + \frac{\lambda_2}{2} \|\boldsymbol{\alpha}\|_2^2. \tag{16}$$

```
#
# Name: lassoMask
#
# Usage: spams.lassoMask(X,D,B,L= -1,lambda1= None,lambda2= 0.,mode= spams_wrap.PENALTY,pos=
    False,
#                  numThreads= -1,verbose = False)
#
# Description:
#     lasso is a variant of lasso that handles
#     binary masks. It aims at addressing the following problems
#     for all columns x of X, and beta of B, it computes one column alpha of A
#     that solves
#       1) when mode=0
#         min_{alpha} ||diag(beta)(x-Dalpha)||_2^2 s.t. ||alpha||_1 <= lambda1
#       2) when mode=1
#         min_{alpha} ||alpha||_1 s.t. ||diag(beta)(x-Dalpha)||_2^2
#                                                       <= lambda1*||beta||_0/m
```

```
#       3) when mode=2
#         min_{alpha} 0.5||diag(beta)(x-Dalpha)||_2^2 +
#                                                 lambda1*(||beta||_0/m)*||alpha||_1 +
#                                                 (lambda2/2)||alpha||_2^2
#     Possibly, when pos=true, it solves the previous problems
#     with positivity constraints on the vectors alpha
#
# Inputs:
#       X:  double m x n matrix   (input signals)
#               m is the signal size
#               n is the number of signals to decompose
#       D:  double m x p matrix   (dictionary)
#             p is the number of elements in the dictionary
#       B:  boolean m x n matrix   (mask)
#             p is the number of elements in the dictionary
#       verbose:       verbose mode
#       lambda1:  (parameter)
#       L: (optional, maximum number of elements of each
#         decomposition)
#       pos: (optional, adds positivity constraints on the
#         coefficients, false by default)
#       mode: (see above, by default: 2)
#       lambda2:  (optional parameter for solving the Elastic-Net)
#                     for mode=0 and mode=1, it adds a ridge on the Gram Matrix
#       numThreads: (optional, number of threads for exploiting
#         multi-core / multi-cpus. By default, it takes the value -1,
#         which automatically selects all the available CPUs/cores).
#
# Output:
#       A: double sparse p x n matrix (output coefficients)
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     this function admits a few experimental usages, which have not
#     been extensively tested:
#         - single precision setting (even though the output alpha is double
#           precision)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
print "test lassoMask"
##################################################
# Decomposition of a large number of signals
##################################################
# data generation
X = np.asfortranarray(np.random.normal(size=(300,300)))
# X=X./repmat(sqrt(sum(X.^2)),[size(X,1) 1]);
X = np.asfortranarray(X / np.tile(np.sqrt((X*X).sum(axis=0)),(X.shape[0],1)))
D = np.asfortranarray(np.random.normal(size=(300,50)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
mask = np.asfortranarray((X > 0))  # generating a binary mask
param = {
    'lambda1' : 0.15, # not more than 20 non-zeros coefficients
    'numThreads' : -1, # number of processors/cores to use; the default choice is -1
    # and uses all the cores of the machine
    'mode' : spams.PENALTY}        # penalized formulation
tic = time.time()
alpha = spams.lassoMask(X,D,mask,**param)
tac = time.time()
t = tac - tic
print "%f signals processed per second\n" %(float(X.shape[1]) / t)
```

4.6 Function spams.cd

Coordinate-descent approach for solving Eq. (10) and Eq. (9). Note that unlike spams.lasso, it is not implemented to solve the Elastic-Net formulation. To solve Eq. (9), the algorithm solves a sequence of problems of the form (10) using simple heuristics. Coordinate descent is very simple and in practice very powerful. It performs better when the correlation between the dictionary elements is small.

```
#
# Name: cd
#
# Usage: spams.cd(X,D,A0,lambda1 = None,mode= spams_wrap.PENALTY,itermax=100,tol = 0.001,
#          numThreads =-1)
#
# Description:
#     cd addresses l1-decomposition problem with a
#     coordinate descent type of approach.
#     It is optimized for solving a large number of small or medium-sized
#     decomposition problem (and not for a single large one).
#     It first computes the Gram matrix D'D.
#     This method is particularly well adapted when there is low
#     correlation between the dictionary elements and when one can benefit
#     from a warm restart.
#     It aims at addressing the two following problems
#     for all columns x of X, it computes a column alpha of A such that
#       2) when mode=1
#         min_{alpha} ||alpha||_1 s.t. ||x-Dalpha||_2^2 <= lambda1
#         For this constraint setting, the method solves a sequence of
#         penalized problems (corresponding to mode=2) and looks
#         for the corresponding Lagrange multplier with a simple but
#         efficient heuristic.
#       3) when mode=2
#         min_{alpha} 0.5||x-Dalpha||_2^2 + lambda1||alpha||_1
#
# Inputs:
#       X:  double m x n matrix   (input signals)
#               m is the signal size
#               n is the number of signals to decompose
#       D:  double m x p matrix   (dictionary)
#             p is the number of elements in the dictionary
#             All the columns of D should have unit-norm !
#       A0:  double sparse p x n matrix   (initial guess)
#       lambda1:  (parameter)
#       mode: (optional, see above, by default 2)
#       itermax: (maximum number of iterations)
#       tol: (tolerance parameter)
#       numThreads: (optional, number of threads for exploiting
#       multi-core / multi-cpus. By default, it takes the value -1,
#       which automatically selects all the available CPUs/cores).
#
# Output:
#       A: double sparse p x n matrix (output coefficients)
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     this function admits a few experimental usages, which have not
#     been extensively tested:
#         - single precision setting (even though the output alpha
#           is double precision)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
X = np.asfortranarray(np.random.normal(size = (64,100)))
X = np.asfortranarray(X / np.tile(np.sqrt((X*X).sum(axis=0)),(X.shape[0],1)))
D = np.asfortranarray(np.random.normal(size = (64,100)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
# parameter of the optimization procedure are chosen
lambda1 = 0.015
mode = spams.PENALTY
tic = time.time()
alpha = spams.lasso(X,D,lambda1 = lambda1,mode = mode,numThreads = 4)
tac = time.time()
t = tac - tic
xd = X - D * alpha
E = np.mean(0.5 * (xd * xd).sum(axis=0) + lambda1 * np.abs(alpha).sum(axis=0))
print "%f signals processed per second for LARS" %(X.shape[1] / t)
print 'Objective function for LARS: %g' %E
tol = 0.001
itermax = 1000
tic = time.time()
#    A0 = ssp.csc_matrix(np.empty((alpha.shape[0],alpha.shape[1])))
A0 = ssp.csc_matrix((alpha.shape[0],alpha.shape[1]))
alpha2 = spams.cd(X,D,A0,lambda1 = lambda1,mode = mode,tol = tol, itermax = itermax,numThreads
    = 4)
tac = time.time()
t = tac - tic
print "%f signals processed per second for CD" %(X.shape[1] / t)
xd = X - D * alpha2
E = np.mean(0.5 * (xd * xd).sum(axis=0) + lambda1 * np.abs(alpha).sum(axis=0))
print 'Objective function for CD: %g' %E
print 'With Random Design, CD can be much faster than LARS'
```

4.7 Function spams.somp

This is a fast implementation of the Simultaneous Orthogonal Matching Pursuit algorithm. Given a set of matrices $\mathbf{X} = [\mathbf{X}^1, \ldots, \mathbf{X}^n]$ in $\mathbb{R}^{m \times N}$, where the $\mathbf{X}^i$'s are in $\mathbb{R}^{m \times n_i}$, and a dictionary $\mathbf{D}$ in $\mathbb{R}^{m \times p}$, the algorithm returns a matrix of coefficients $\mathbf{A} = [\mathbf{A}^1, \ldots, \mathbf{A}^n]$ in $\mathbb{R}^{p \times N}$ which is an approximate solution of the following NP-hard problem

$$\forall i \quad \min_{\mathbf{A}^i \in \mathbb{R}^{p \times n_i}} \|\mathbf{X}^i - \mathbf{D}\mathbf{A}^i\|_F^2 \quad \text{s.t.} \quad \|\mathbf{A}^i\|_{0,\infty} \leq L. \tag{17}$$

or

$$\forall i \quad \min_{\mathbf{A}^i \in \mathbb{R}^{p \times n_i}} \|\mathbf{A}^i\|_{0,\infty} \quad \text{s.t.} \quad \|\mathbf{X}^i - \mathbf{D}\mathbf{A}^i\|_F^2 \leq \varepsilon n_i. \tag{18}$$

To be efficient, the method first compute the covariance matrix $\mathbf{D}^T\mathbf{D}$, then for each signal, it computes $\mathbf{D}^T\mathbf{X}^i$ and performs the decomposition with a Cholesky-based algorithm.

```
#
# Name: somp
# (this function has not been intensively tested).
#
# Usage: spams.somp(X,D,list_groups,L = None,eps = 0.,numThreads = -1)
#
# Description:
#     somp is an efficient implementation of a
#     Simultaneous Orthogonal Matching Pursuit algorithm. It is optimized
#     for solving a large number of small or medium-sized
#     decomposition problem (and not for a single large one).
#     It first computes the Gram matrix D'D and then perform
#     a Cholesky-based OMP of the input signals in parallel.
#     It aims at addressing the following NP-hard problem
#
#     X is a matrix structured in groups of signals, which we denote
#     by X=[X_1,...,X_n]
```

```
#
#     for all matrices X_i of X,
#         min_{A_i} ||A_i||_{0,infty}  s.t  ||X_i-D A_i||_2^2 <= eps*n_i
#         where n_i is the number of columns of X_i
#
#         or
#
#         min_{A_i} ||X_i-D A_i||_2^2  s.t. ||A_i||_{0,infty} <= L
#
# Inputs:
#       X:  double m x N matrix   (input signals)
#            m is the signal size
#            N is the total number of signals
#       D:  double m x p matrix   (dictionary)
#          p is the number of elements in the dictionary
#          All the columns of D should have unit-norm !
#       list_groups : int32 vector containing the indices (starting at 0)
#          of the first elements of each groups.
#       L: (maximum number of elements in each decomposition)
#       eps: (threshold on the squared l2-norm of the residual
#       numThreads: (optional, number of threads for exploiting
#       multi-core / multi-cpus. By default, it takes the value -1,
#       which automatically selects all the available CPUs/cores).
#
# Output:
#       alpha: double sparse p x N matrix (output coefficients)
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     this function admits a few experimental usages, which have not
#     been extensively tested:
#      - single precision setting (even though the output alpha is double
#        precision)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
X = np.asfortranarray(np.random.normal(size = (64,10000)))
D = np.asfortranarray(np.random.normal(size = (64,200)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
ind_groups = np.array(xrange(0,10000,10),dtype=np.int32)
tic = time.time()
alpha = spams.somp(X,D,ind_groups,L = 10,eps = 0.1,numThreads=-1)
tac = time.time()
t = tac - tic
print "%f signals processed per second" %(X.shape[1] / t)
```

4.8 Function spams.l1L2BCD

This is a fast implementation of a simultaneous signal decomposition formulation. Given a set of matrices $\mathbf{X} = [\mathbf{X}^1, \ldots, \mathbf{X}^n]$ in $\mathbb{R}^{m \times N}$, where the $\mathbf{X}^i$'s are in $\mathbb{R}^{m \times n_i}$, and a dictionary $\mathbf{D}$ in $\mathbb{R}^{m \times p}$, the algorithm returns a matrix of coefficients $\mathbf{A} = [\mathbf{A}^1, \ldots, \mathbf{A}^n]$ in $\mathbb{R}^{p \times N}$ which is an approximate solution of the following NP-hard problem

$$\forall i \quad \min_{\mathbf{A}^i \in \mathbb{R}^{p \times n_i}} \|\mathbf{X}^i - \mathbf{D}\mathbf{A}^i\|_F^2 \quad \text{s.t.} \quad \|\mathbf{A}^i\|_{1,2} \leq \frac{\lambda}{n_i}. \tag{19}$$

or

$$\forall i \quad \min_{\mathbf{A}^i \in \mathbb{R}^{p \times n_i}} \|\mathbf{A}^i\|_{1,2} \quad \text{s.t.} \quad \|\mathbf{X}^i - \mathbf{D}\mathbf{A}^i\|_F^2 \leq \lambda n_i. \tag{20}$$

To be efficient, the method first compute the covariance matrix $\mathbf{D}^T\mathbf{D}$, then for each signal, it computes $\mathbf{D}^T\mathbf{X}^i$ and performs the decomposition with a Cholesky-based algorithm.

```
#
# Name: l1L2BCD
# (this function has not been intensively tested).
#
# Usage: spams.l1L2BCD(X,D,alpha0,list_groups,lambda1 = None,mode= spams_wrap.PENALTY,itermax =
    100,
#              tol = 1e-3,numThreads = -1)
#
# Description:
#     l1L2BCD is a solver for a
#     Simultaneous signal decomposition formulation based on block
#     coordinate descent.
#
#     X is a matrix structured in groups of signals, which we denote
#     by X=[X_1,...,X_n]
#
#     if mode=2, it solves
#         for all matrices X_i of X,
#         min_{A_i} 0.5||X_i-D A_i||_2^2 + lambda1/sqrt(n_i)||A_i||_{1,2}
#         where n_i is the number of columns of X_i
#     if mode=1, it solves
#         min_{A_i} ||A_i||_{1,2} s.t. ||X_i-D A_i||_2^2 <= n_i lambda1
#
# Inputs:
#       X:  double m x N matrix   (input signals)
#           m is the signal size
#           N is the total number of signals
#       D:  double m x p matrix   (dictionary)
#          p is the number of elements in the dictionary
#       alpha0: double dense p x N matrix (initial solution)
#       list_groups : int32 vector containing the indices (starting at 0)
#          of the first elements of each groups.
#       lambda1: (regularization parameter)
#       mode: (see above, by default 2)
#       itermax: (maximum number of iterations, by default 100)
#       tol: (tolerance parameter, by default 0.001)
#       numThreads: (optional, number of threads for exploiting
#       multi-core / multi-cpus. By default, it takes the value -1,
#       which automatically selects all the available CPUs/cores).
#
# Output:
#       alpha: double sparse p x N matrix (output coefficients)
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     this function admits a few experimental usages, which have not
#     been extensively tested:
#      - single precision setting (even though the output alpha is double
#        precision)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
X = np.asfortranarray(np.random.normal(size = (64,100)))
D = np.asfortranarray(np.random.normal(size = (64,200)))
D = np.asfortranarray(D / np.tile(np.sqrt((D*D).sum(axis=0)),(D.shape[0],1)))
ind_groups = np.array(xrange(0,X.shape[1],10),dtype=np.int32) #indices of the first signals in
    each group
# parameters of the optimization procedure are chosen
```

```
itermax = 100
tol = 1e-3
mode = spams.PENALTY
lambda1 = 0.15 # squared norm of the residual should be less than 0.1
numThreads = -1 # number of processors/cores to use the default choice is -1
                # and uses all the cores of the machine
alpha0 = np.zeros((D.shape[1],X.shape[1]),dtype=np.float64,order="FORTRAN")
tic = time.time()
alpha = spams.l1L2BCD(X,D,alpha0,ind_groups,lambda1 = lambda1,mode = mode,itermax = itermax,tol
    = tol,numThreads = numThreads)
tac = time.time()
t = tac - tic
print "%f signals processed per second" %(X.shape[1] / t)
```

4.9 Function spams.sparseProject

This is a multi-purpose function, implementing fast algorithms for projecting on convex sets, but it also solves the fused lasso signal approximation problem. The proposed method is detailed in [20]. The main problems addressed by this function are the following: Given a matrix $\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_n]$ in $\mathbb{R}^{m \times n}$, it finds a matrix $\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_n]$ in $\mathbb{R}^{m \times n}$ so that for all column $\mathbf{u}$ of $\mathbf{U}$, it computes a column $\mathbf{v}$ of $\mathbf{V}$ solving

$$\min_{\mathbf{v} \in \mathbb{R}^m} \|\mathbf{u} - \mathbf{v}\|_2^2 \quad \text{s.t.} \quad \|\mathbf{v}\|_1 \leq \tau, \tag{21}$$

or

$$\min_{\mathbf{v} \in \mathbb{R}^m} \|\mathbf{u} - \mathbf{v}\|_2^2 \quad \text{s.t.} \quad \lambda_1 \|\mathbf{v}\|_1 + \lambda_2 \|\mathbf{v}\|_2^2 \leq \tau, \tag{22}$$

or

$$\min_{\mathbf{v} \in \mathbb{R}^m} \|\mathbf{u} - \mathbf{v}\|_2^2 \quad \text{s.t.} \quad \lambda_1 \|\mathbf{v}\|_1 + \lambda_2 \|\mathbf{v}\|_2^2 + \lambda_3 FL(\mathbf{v}) \leq \tau, \tag{23}$$

or

$$\min_{\mathbf{v} \in \mathbb{R}^m} \frac{1}{2} \|\mathbf{u} - \mathbf{v}\|_2^2 + \lambda_1 \|\mathbf{v}\|_1 + \lambda_2 \|\mathbf{v}\|_2^2 + \lambda_3 FL(\mathbf{v}). \tag{24}$$

Note that for the two last cases, the method performs a small approximation. The method follows the regularization path, goes from one kink to another, and stop whenever the constraint is not satisfied anymore. The solution returned by the algorithm is the one obtained at the last kink of the regularization path, which is in practice close, but not exactly the same as the solution. This will be corrected in a future release of the toolbox.

```
#
# Name: sparseProject
#
# Usage: spams.sparseProject(U,thrs = 1.0,mode = 1,lambda1 = 0.0,lambda2 = 0.0,lambda3 = 0.0,
#                      pos = 0,numThreads = -1)
#
# Description:
#     sparseProject solves various optimization
#     problems, including projections on a few convex sets.
#     It aims at addressing the following problems
#     for all columns u of U in parallel
#       1) when mode=1 (projection on the l1-ball)
#           min_v ||u-v||_2^2  s.t.  ||v||_1 <= thrs
#       2) when mode=2
#           min_v ||u-v||_2^2  s.t. ||v||_2^2 + lamuda1||v||_1 <= thrs
#       3) when mode=3
#           min_v ||u-v||_2^2  s.t  ||v||_1 + 0.5lamuda1||v||_2^2 <= thrs
#       4) when mode=4
#           min_v 0.5||u-v||_2^2 + lamuda1||v||_1  s.t  ||v||_2^2 <= thrs
#       5) when mode=5
#           min_v 0.5||u-v||_2^2 + lamuda1||v||_1 +lamuda2 FL(v) + ...
#                                                 0.5lamuda_3 ||v||_2^2
#          where FL denotes a "fused lasso" regularization term.
#       6) when mode=6
#          min_v ||u-v||_2^2 s.t lamuda1||v||_1 +lamuda2 FL(v) + ...
#                                               0.5lamuda3||v||_2^2 <= thrs
```

```
#
#        When pos=true and mode <= 4,
#        it solves the previous problems with positivity constraints
#
# Inputs:
#       U:  double m x n matrix   (input signals)
#                m is the signal size
#                n is the number of signals to project
#       thrs: (parameter)
#       lambda1: (parameter)
#       lambda2: (parameter)
#       lambda3: (parameter)
#       mode: (see above)
#       pos: (optional, false by default)
#       numThreads: (optional, number of threads for exploiting
#         multi-core / multi-cpus. By default, it takes the value -1,
#         which automatically selects all the available CPUs/cores).
#
# Output:
#       V: double m x n matrix (output matrix)
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     this function admits a few experimental usages, which have not
#     been extensively tested:
#         - single precision setting
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
X = np.asfortranarray(np.random.normal(size = (20000,100)))
X = np.asfortranarray(X / np.tile(np.sqrt((X*X).sum(axis=0)),(X.shape[0],1)))
param = {'numThreads' : -1, # number of processors/cores to use (-1 => all cores)
         'pos' : False,
         'mode': 1, # projection on the l1 ball
         'thrs' : 2}
print "\n  Projection on the l1 ball"
tic = time.time()
X1 = spams.sparseProject(X,**param)
tac = time.time()
t = tac - tic
print "  Time : ", t
if (t != 0):
    print "%f signals of size %d projected per second" %((X.shape[1] / t),X.shape[0])
s = np.abs(X1).sum(axis=0)
print "Checking constraint: %f, %f" %(min(s),max(s))

print "\n  Projection on the Elastic-Net"
param['mode'] = 2  # projection on the Elastic-Net
param['lambda1'] = 0.15
tic = time.time()
X1 = spams.sparseProject(X,**param)
tac = time.time()
t = tac - tic
print "  Time : ", t
if (t != 0):
    print "%f signals of size %d projected per second" %((X.shape[1] / t),X.shape[0])
constraints = (X1*X1).sum(axis=0) + param['lambda1'] * np.abs(X1).sum(axis=0)
print 'Checking constraint: %f, %f (Projection is approximate : stops at a kink)' %(min(
    constraints),max(constraints))

print "\n  Projection on the FLSA"
```

```
param['mode'] = 6        # projection on the FLSA
param['lambda1'] = 0.7
param['lambda2'] = 0.7
param['lambda3'] = 1.0
X = np.asfortranarray(np.random.random(size = (2000,100)))
X = np.asfortranarray(X / np.tile(np.sqrt((X*X).sum(axis=0)),(X.shape[0],1)))
tic = time.time()
X1 = spams.sparseProject(X,**param)
tac = time.time()
t = tac - tic
print "  Time : ", t
if (t != 0):
    print "%f signals of size %d projected per second" %((X.shape[1] / t),X.shape[0])
constraints = 0.5 * param['lambda3'] * (X1*X1).sum(axis=0) + param['lambda1'] * np.abs(X1).sum(
    axis=0) + \
param['lambda2'] * np.abs(X1[2:,] - X1[1:-1,]).sum(axis=0)
print 'Checking constraint: %f, %f (Projection is approximate : stops at a kink)' %(min(
    constraints),max(constraints))
```

5 Proximal Toolbox

The previous toolbox we have presented is well adapted for solving a large number of small and medium-scale sparse decomposition problems with the square loss, which is typical from the classical dictionary learning framework. We now present a new software package that is adapted for solving a wide range of possibly large-scale learning problems, with several combinations of losses and regularization terms. The method implements the proximal methods of [1], and includes the proximal solvers for the tree-structured regularization of [14], and the solver of [21] for general structured sparse regularization. The solver for structured sparse regularization norms includes a C++ max-flow implementation of the push-relabel algorithm of [12], with heuristics proposed by [5].

This implementation also provides robust stopping criteria based on *duality gaps*, which are presented in Appendix A. It can handle intercepts (unregularized variables). The general formulation that our software can solve take the form

$$\min_{\mathbf{w}\in\mathbb{R}^p} [g(\mathbf{w}) \triangleq f(\mathbf{w}) + \lambda\psi(\mathbf{w})],$$

where f is a smooth loss function and ψ is a regularization function. When one optimizes a matrix $\mathbf{W}$ in $\mathbb{R}^{p\times r}$ instead of a vector $\mathbf{w}$ in $\mathbb{R}^p$, we will write

$$\min_{\mathbf{W}\in\mathbb{R}^{p\times r}} [g(\mathbf{W}) \triangleq f(\mathbf{W}) + \lambda\psi(\mathbf{W})].$$

Note that the software can possibly handle nonnegativity constraints.

We start by presenting the type of regularization implemented in the software

5.1 Regularization Functions

Our software can handle the following regularization functions ψ for vectors $\mathbf{w}$ in $\mathbb{R}^p$:

- **The Tikhonov regularization**: $\psi(\mathbf{w}) \triangleq \frac{1}{2}\|\mathbf{w}\|_2^2$.
- **The ℓ_1-norm**: $\psi(\mathbf{w}) \triangleq \|\mathbf{w}\|_1$.
- **The Elastic-Net**: $\psi(\mathbf{w}) \triangleq \|\mathbf{w}\|_1 + \gamma\|\mathbf{w}\|_2^2$.
- **The Fused-Lasso**: $\psi(\mathbf{w}) \triangleq \|\mathbf{w}\|_1 + \gamma\|\mathbf{w}\|_2^2 + \gamma_2 \sum_{i=1}^{p-1} |\mathbf{w}_{i+1} - \mathbf{w}_i|$.
- **The group Lasso**: $\psi(\mathbf{w}) \triangleq \sum_{g\in\mathcal{G}} \eta_g \|\mathbf{w}_g\|_2$, where $\mathcal{G}$ are groups of variables.
- **The group Lasso with ℓ_∞-norm**: $\psi(\mathbf{w}) \triangleq \sum_{g\in\mathcal{G}} \eta_g \|\mathbf{w}_g\|_\infty$, where $\mathcal{G}$ are groups of variables.
- **The sparse group Lasso**: same as above but with an additional ℓ_1 term.

- **The tree-structured sum of ℓ_2-norms**: $\psi(\mathbf{w}) \triangleq \sum_{g\in\mathcal{G}} \eta_g \|\mathbf{w}_g\|_2$, where $\mathcal{G}$ is a tree-structured set of groups [14], and the η_g are positive weights.
- **The tree-structured sum of ℓ_∞-norms**: $\psi(\mathbf{w}) \triangleq \sum_{g\in\mathcal{G}} \eta_g \|\mathbf{w}_g\|_\infty$. See [14]
- **General sum of ℓ_∞-norms**: $\psi(\mathbf{w}) \triangleq \sum_{g\in\mathcal{G}} \eta_g \|\mathbf{w}_g\|_\infty$, where no assumption are made on the groups $\mathcal{G}$.

Our software also handles regularization functions ψ on matrices $\mathbf{W}$ in $\mathbb{R}^{p\times r}$ (note that $\mathbf{W}$ can be transposed in these formulations). In particular,

- **The ℓ_1/ℓ_2-norm**: $\psi(\mathbf{W}) \triangleq \sum_{i=1}^p \|\mathbf{W}_i\|_2$, where $\mathbf{W}_i$ denotes the i-th row of $\mathbf{W}$.
- **The ℓ_1/ℓ_∞-norm**: $\psi(\mathbf{W}) \triangleq \sum_{i=1}^p \|\mathbf{W}_i\|_\infty$,
- **The ℓ_1/ℓ_2+ℓ_1-norm**: $\psi(\mathbf{W}) \triangleq \sum_{i=1}^p \|\mathbf{W}_i\|_2 + \lambda_2 \sum_{i,j} |\mathbf{W}_{ij}|$.
- **The ℓ_1/ℓ_∞+ℓ_1-norm**: $\psi(\mathbf{W}) \triangleq \sum_{i=1}^p \|\mathbf{W}_i\|_\infty + \lambda_2 \sum_{i,j} |\mathbf{W}_{ij}|$,
- **The ℓ_1/ℓ_∞-norm on rows and columns**: $\psi(\mathbf{W}) \triangleq \sum_{i=1}^p \|\mathbf{W}_i\|_\infty + \lambda_2 \sum_{j=1}^r \|\mathbf{W}^j\|_\infty$, where $\mathbf{W}^j$ denotes the j-th column of $\mathbf{W}$.
- **The multi-task tree-structured sum of ℓ_∞-norms**:

$$\psi(\mathbf{W}) \triangleq \sum_{i=1}^r \sum_{g\in\mathcal{G}} \eta_g \|\mathbf{w}_g^i\|_\infty + \gamma \sum_{g\in\mathcal{G}} \eta_g \max_{j\in g} \|\mathbf{W}_j\|_\infty, \tag{25}$$

  where the first double sums is in fact a sum of independent structured norms on the columns $\mathbf{w}^i$ of $\mathbf{W}$, and the right term is a tree-structured regularization norm applied to the ℓ_∞-norm of the rows of $\mathbf{W}$, thereby inducing the tree-structured regularization at the row level. $\mathcal{G}$ is here a tree-structured set of groups.
- **The multi-task general sum of ℓ_∞-norms** is the same as Eq. (25) except that the groups $\mathcal{G}$ are general overlapping groups.
- **The trace norm**: $\psi(\mathbf{W}) \triangleq \|\mathbf{W}\|_*$.

Non-convex regularizations are also implemented with the ISTA algorithm (no duality gaps are of course provided in these cases):

- **The ℓ_0-pseudo-norm**: $\psi(\mathbf{w}) \triangleq \|\mathbf{w}\|_0$.
- **The rank**: $\psi(\mathbf{W}) \triangleq \text{randk}(\mathbf{W})$.
- **The tree-structured ℓ_0-pseudo-norm**: $\psi(\mathbf{w}) \triangleq \sum_{g\in\mathcal{G}} \delta_{\mathbf{w}_g\neq 0}$.

All of these regularization terms for vectors or matrices can be coupled with nonnegativity constraints. It is also possible to add an intercept, which one wishes not to regularize, and we will include this possibility in the next sections. There are also a few hidden undocumented options which are available in the source code.

We now present 3 functions for computing proximal operators associated to the previous regularization functions.

5.2 Function spams.proximalFlat

This function computes the proximal operators associated to many regularization functions, for input signals $\mathbf{U} = [\mathbf{u}^1, \ldots, \mathbf{u}^n]$ in $\mathbb{R}^{p\times n}$, it finds a matrix $\mathbf{V} = [\mathbf{v}^1, \ldots, \mathbf{v}^n]$ in $\mathbb{R}^{p\times n}$ such that:

• If one chooses a regularization function on vectors, for every column $\mathbf{u}$ of $\mathbf{U}$, it computes one column $\mathbf{v}$ of $\mathbf{V}$ solving

$$\min_{\mathbf{v}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{u}-\mathbf{v}\|_2^2 + \lambda\|\mathbf{v}\|_0, \tag{26}$$

or

$$\min_{\mathbf{v}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{u}-\mathbf{v}\|_2^2 + \lambda\|\mathbf{v}\|_1, \tag{27}$$

or

$$\min_{\mathbf{v}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{u}-\mathbf{v}\|_2^2 + \frac{\lambda}{2}\|\mathbf{v}\|_2^2, \tag{28}$$

or

$$\min_{\mathbf{v}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{u}-\mathbf{v}\|_2^2 + \lambda\|\mathbf{v}\|_1 + \lambda_2\|\mathbf{v}\|_2^2, \tag{29}$$

or

$$\min_{\mathbf{v}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{u}-\mathbf{v}\|_2^2 + \lambda\sum_{j=1}^{p-1}|\mathbf{v}_{j+1}^i - \mathbf{v}_j^i| + \lambda_2\|\mathbf{v}\|_1 + \lambda_3\|\mathbf{v}\|_2^2, \tag{30}$$

or

$$\min_{\mathbf{v}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{u}-\mathbf{v}\|_2^2 + \lambda\sum_{g\in\mathcal{T}}\delta^g(\mathbf{v}), \tag{31}$$

where $\mathcal{T}$ is a tree-structured set of groups (see [15]), and $\delta^g(\mathbf{v}) = 0$ if $\mathbf{v}_g = 0$ and 1 otherwise. It can also solve

$$\min_{\mathbf{v}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{u}-\mathbf{v}\|_2^2 + \lambda\sum_{g\in\mathcal{T}}\eta^g\|\mathbf{v}_g\|_2, \tag{32}$$

or

$$\min_{\mathbf{v}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{u}-\mathbf{v}\|_2^2 + \lambda\sum_{g\in\mathcal{T}}\eta^g\|\mathbf{v}_g\|_\infty, \tag{33}$$

or

$$\min_{\mathbf{v}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{u}-\mathbf{v}\|_2^2 + \lambda\sum_{g\in\mathcal{G}}\eta^g\|\mathbf{v}_g\|_\infty, \tag{34}$$

where $\mathcal{G}$ is any kind of set of groups.

This function can also solve the following proximal operators on matrices

$$\min_{\mathbf{V}\in\mathbb{R}^{p\times n}} \frac{1}{2}\|\mathbf{U}-\mathbf{V}\|_F^2 + \lambda\sum_{i=1}^{p}\|\mathbf{V}_i\|_2, \tag{35}$$

where $\mathbf{V}_i$ is the i-th row of $\mathbf{V}$, or

$$\min_{\mathbf{V}\in\mathbb{R}^{p\times n}} \frac{1}{2}\|\mathbf{U}-\mathbf{V}\|_F^2 + \lambda\sum_{i=1}^{p}\|\mathbf{V}_i\|_\infty, \tag{36}$$

or

$$\min_{\mathbf{V}\in\mathbb{R}^{p\times n}} \frac{1}{2}\|\mathbf{U}-\mathbf{V}\|_F^2 + \lambda\sum_{i=1}^{p}\|\mathbf{V}_i\|_2 + \lambda_2\sum_{i=1}^{p}\sum_{j=1}^{n}|\mathbf{V}_{ij}|, \tag{37}$$

or

$$\min_{\mathbf{V}\in\mathbb{R}^{p\times n}} \frac{1}{2}\|\mathbf{U}-\mathbf{V}\|_F^2 + \lambda\sum_{i=1}^{p}\|\mathbf{V}_i\|_\infty + \lambda_2\sum_{i=1}^{p}\sum_{j=1}^{n}|\mathbf{V}_{ij}|, \tag{38}$$

or

$$\min_{\mathbf{V}\in\mathbb{R}^{p\times n}} \frac{1}{2}\|\mathbf{U}-\mathbf{V}\|_F^2 + \lambda\sum_{i=1}^{p}\|\mathbf{V}_i\|_\infty + \lambda_2\sum_{j=1}^{n}\|\mathbf{V}^j\|_\infty. \tag{39}$$

where $\mathbf{V}^j$ is the j-th column of $\mathbf{V}$.

See usage details below:

```
#
# Name: proximalFlat
#
# Usage: spams.proximalFlat(U,return_val_loss = False,numThreads =-1,lambda1=1.0,lambda2=0.,
#                     lambda3=0.,intercept=False,resetflow=False,regul="",verbose=False,
#                     pos=False,clever=True,size_group=1,groups = None,transpose=False)
#
# Description:
```

```
#     proximalFlat computes proximal operators. Depending
#        on the value of regul, it computes
#
#        Given an input matrix U=[u^1,\ldots,u^n], it computes a matrix
#        V=[v^1,\ldots,v^n] such that
#        if one chooses a regularization functions on vectors, it computes
#        for each column u of U, a column v of V solving
#        if regul='l0'
#            argmin 0.5||u-v||_2^2 + lambda1||v||_0
#        if regul='l1'
#            argmin 0.5||u-v||_2^2 + lambda1||v||_1
#        if regul='l2'
#            argmin 0.5||u-v||_2^2 + 0.5lambda1||v||_2^2
#        if regul='elastic-net'
#            argmin 0.5||u-v||_2^2 + lambda1||v||_1 + lambda1_2||v||_2^2
#        if regul='fused-lasso'
#            argmin 0.5||u-v||_2^2 + lambda1 FL(v) + ...
#                                ...  lambda1_2||v||_1 + lambda1_3||v||_2^2
#        if regul='linf'
#            argmin 0.5||u-v||_2^2 + lambda1||v||_inf
#        if regul='l1-constraint'
#            argmin 0.5||u-v||_2^2 s.t. ||v||_1 <= lambda1
#        if regul='l2-not-squared'
#            argmin 0.5||u-v||_2^2 + lambda1||v||_2
#        if regul='group-lasso-l2'
#            argmin 0.5||u-v||_2^2 + lambda1 sum_g ||v_g||_2
#            where the groups are either defined by groups or by size_group,
#        if regul='group-lasso-linf'
#            argmin 0.5||u-v||_2^2 + lambda1 sum_g ||v_g||_inf
#        if regul='sparse-group-lasso-l2'
#            argmin 0.5||u-v||_2^2 + lambda1 sum_g ||v_g||_2 + lambda1_2 ||v||_1
#            where the groups are either defined by groups or by size_group,
#        if regul='sparse-group-lasso-linf'
#            argmin 0.5||u-v||_2^2 + lambda1 sum_g ||v_g||_inf + lambda1_2 ||v||_1
#        if regul='trace-norm-vec'
#            argmin 0.5||u-v||_2^2 + lambda1 ||mat(v)||_*
#           where mat(v) has size_group rows
#
#        if one chooses a regularization function on matrices
#        if regul='l1l2', V=
#            argmin 0.5||U-V||_F^2 + lambda1||V||_{1/2}
#        if regul='l1linf', V=
#            argmin 0.5||U-V||_F^2 + lambda1||V||_{1/inf}
#        if regul='l1l2+l1', V=
#            argmin 0.5||U-V||_F^2 + lambda1||V||_{1/2} + lambda1_2||V||_{1/1}
#        if regul='l1linf+l1', V=
#            argmin 0.5||U-V||_F^2 + lambda1||V||_{1/inf} + lambda1_2||V||_{1/1}
#        if regul='l1linf+row-column', V=
#            argmin 0.5||U-V||_F^2 + lambda1||V||_{1/inf} + lambda1_2||V'||_{1/inf}
#        if regul='trace-norm', V=
#            argmin 0.5||U-V||_F^2 + lambda1||V||_*
#        if regul='rank', V=
#            argmin 0.5||U-V||_F^2 + lambda1 rank(V)
#        if regul='none', V=
#            argmin 0.5||U-V||_F^2
#
#        for all these regularizations, it is possible to enforce non-negativity constraints
#        with the option pos, and to prevent the last row of U to be regularized, with
#        the option intercept
#
# Inputs:
#       U:  double m x n matrix   (input signals)
#              m is the signal size
#       return_val_loss:
#              if true the function will return a tuple of matrices.
#       lambda1:  (regularization parameter)
#       regul: (choice of regularization, see above)
```

```
#       lambda2:  (optional, regularization parameter)
#       lambda3:  (optional, regularization parameter)
#       verbose: (optional, verbosity level, false by default)
#       intercept: (optional, last row of U is not regularized,
#         false by default)
#       transpose: (optional, transpose the matrix in the regularization function)
#       size_group: (optional, for regularization functions assuming a group
#         structure). It is a scalar. When groups is not specified, it assumes
#         that the groups are the sets of consecutive elements of size size_group
#       groups: (int32, optional, for regularization functions assuming a group
#         structure. It is an int32 vector of size m containing the group indices of the
#         variables (first group is 1).
#       pos: (optional, adds positivity constraints on the
#         coefficients, false by default)
#       numThreads: (optional, number of threads for exploiting
#         multi-core / multi-cpus. By default, it takes the value -1,
#         which automatically selects all the available CPUs/cores).
#       resetflow:    undocumented; modify at your own risks!
#       clever:    undocumented; modify at your own risks!
#
# Output:
#       V: double m x n matrix (output coefficients)
#       val_regularizer: double 1 x n vector (value of the regularization
#       term at the optimum).
#       val_loss:        vector of size U.shape[1]
#         alpha = spams.proximalFlat(U,return_val_loss = False,...)
#         (alpha,val_loss) = spams.proximalFlat(U,return_val_loss = True,...)
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     Valid values for the regularization parameter (regul) are:
#       "l0", "l1", "l2", "linf", "l2-not-squared", "elastic-net", "fused-lasso",
#       "group-lasso-l2", "group-lasso-linf", "sparse-group-lasso-l2",
#       "sparse-group-lasso-linf", "l1l2", "l1linf", "l1l2+l1", "l1linf+l1",
#       "tree-l0", "tree-l2", "tree-linf", "graph", "graph-ridge", "graph-l2",
#       "multi-task-tree", "multi-task-graph", "l1linf-row-column", "trace-norm",
#       "trace-norm-vec", "rank", "rank-vec", "none"
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
param = {'numThreads' : -1,'verbose' : True,
         'lambda1' : 0.1 }
m = 100;n = 1000
U = np.asfortranarray(np.random.normal(size = (m,n)))

# test L0
print "\nprox l0"
param['regul'] = 'l0'
param['pos'] = False       # false by default
param['intercept'] = False # false by default
alpha = spams.proximalFlat(U,False,**param)

# test L1
print "\nprox l1, intercept, positivity constraint"
param['regul'] = 'l1'
param['pos'] = True       # can be used with all the other regularizations
param['intercept'] = True # can be used with all the other regularizations
alpha = spams.proximalFlat(U,False,**param)

# test L2
print "\nprox squared-l2"
param['regul'] = 'l2'
```

```
param['pos'] = False
param['intercept'] = False
alpha = spams.proximalFlat(U,False,**param)

# test elastic-net
print "\nprox elastic-net"
param['regul'] = 'elastic-net'
param['lambda2'] = 0.1
alpha = spams.proximalFlat(U,**param)

# test fused-lasso
print "\nprox fused lasso"
param['regul'] = 'fused-lasso'
param['lambda2'] = 0.1
param['lambda3'] = 0.1
alpha = spams.proximalFlat(U,**param)

# test l1l2
print "\nprox mixed norm l1/l2"
param['regul'] = 'l1l2'
alpha = spams.proximalFlat(U,**param)

# test l1linf
print "\nprox mixed norm l1/linf"
param['regul'] = 'l1linf'
alpha = spams.proximalFlat(U,**param)

# test l1l2+l1
print "\nprox mixed norm l1/l2 + l1"
param['regul'] = 'l1l2+l1'
param['lambda2'] = 0.1
alpha = spams.proximalFlat(U,**param)

# test l1linf+l1
print "\nprox mixed norm l1/linf + l1"
param['regul'] = 'l1linf+l1'
param['lambda2'] = 0.1
alpha = spams.proximalFlat(U,**param)

# test l1linf-row-column
print "\nprox mixed norm l1/linf on rows and columns"
param['regul'] = 'l1linf-row-column'
param['lambda2'] = 0.1
alpha = spams.proximalFlat(U,**param)

# test none
print "\nprox no regularization"
param['regul'] = 'none'
alpha = spams.proximalFlat(U,**param)
```

5.3 Function spams.proximalTree

This function computes the proximal operators associated to tree-structured regularization functions, for input signals $\mathbf{U} = [\mathbf{u}^1, \ldots, \mathbf{u}^n]$ in $\mathbb{R}^{p \times n}$, and a tree-structured set of groups [14], it computes a matrix $\mathbf{V} = [\mathbf{v}^1, \ldots, \mathbf{v}^n]$ in $\mathbb{R}^{p \times n}$. When one uses a regularization function on vectors, it computes a column $\mathbf{v}$ of $\mathbf{V}$ for every column $\mathbf{u}$ of $\mathbf{U}$:

$$\min_{\mathbf{v} \in \mathbb{R}^p} \frac{1}{2}\|\mathbf{u} - \mathbf{v}\|_2^2 + \lambda \sum_{g \in \mathcal{T}} \eta^g \|\mathbf{v}_g\|_2, \tag{40}$$

or

$$\min_{\mathbf{v} \in \mathbb{R}^p} \frac{1}{2}\|\mathbf{u} - \mathbf{v}\|_2^2 + \lambda \sum_{g \in \mathcal{T}} \eta^g \|\mathbf{v}_g\|_\infty, \tag{41}$$

or

$$\min_{\mathbf{v} \in \mathbb{R}^p} \frac{1}{2}\|\mathbf{u} - \mathbf{v}\|_2^2 + \lambda \sum_{g \in \mathcal{T}} \delta^g(\mathbf{v}), \tag{42}$$

where $\delta^g(\mathbf{v}) = 0$ if $\mathbf{v}_g = 0$ and 1 otherwise (see appendix of [15]).

When the multi-task tree-structured regularization function is used, it solves

$$\min_{\mathbf{V}\in\mathbb{R}^{p\times n}} \frac{1}{2}\|\mathbf{U}-\mathbf{V}\|_F^2 + \lambda \sum_{i=1}^{n}\sum_{g\in\mathcal{T}} \eta^g \|\mathbf{v}_g^i\|_\infty + \lambda_2 \sum_{g\in\mathcal{T}} \max_{j\in g} \|\mathbf{v}_g^j\|_\infty, \tag{43}$$

which is a formulation presented in [21].

This function can also be used for computing the proximal operators addressed by spams.proximalFlat (it will just not take into account the tree structure). The way the tree is incoded is presented below, (and examples are given in the file test_ProximalTree.m, with more usage details:

```
#
# Name: proximalTree
#
# Usage: spams.proximalTree(U,tree,return_val_loss = False,numThreads =-1,lambda1=1.0,lambda2
    =0.,
#                    lambda3=0.,intercept=False,resetflow=False,regul="",verbose=False,
#                    pos=False,clever=True,size_group=1,transpose=False)
#
# Description:
#     proximalTree computes a proximal operator. Depending
#         on the value of regul, it computes
#
#         Given an input matrix U=[u^1,\ldots,u^n], and a tree-structured set of groups T,
#         it returns a matrix V=[v^1,\ldots,v^n]:
#
#         when the regularization function is for vectors,
#         for every column u of U, it compute a column v of V solving
#         if regul='tree-l0'
#             argmin 0.5||u-v||_2^2 + lambda1 \sum_{g \in T} \delta^g(v)
#         if regul='tree-l2'
#           for all i, v^i =
#             argmin 0.5||u-v||_2^2 + lambda1\sum_{g \in T} \eta_g||v_g||_2
#         if regul='tree-linf'
#           for all i, v^i =
#             argmin 0.5||u-v||_2^2 + lambda1\sum_{g \in T} \eta_g||v_g||_inf
#
#         when the regularization function is for matrices:
#         if regul='multi-task-tree'
#            V=argmin 0.5||U-V||_F^2 + lambda1 \sum_{i=1}^n\sum_{g \in T} \eta_g||v^i_g||_inf +
    ...
#                                                 lambda1_2 \sum_{g \in T} \eta_g max_{j in g}||
    V_j||_{inf}
#
#         it can also be used with any non-tree-structured regularization addressed by
    proximalFlat
#
#         for all these regularizations, it is possible to enforce non-negativity constraints
#         with the option pos, and to prevent the last row of U to be regularized, with
#         the option intercept
#
# Inputs:
#       U:  double m x n matrix   (input signals)
#               m is the signal size
#       tree: named list
#             with four fields, eta_g, groups, own_variables and N_own_variables.
#
#             The tree structure requires a particular organization of groups and variables
#                * Let us denote by N = |T|, the number of groups.
#                  the groups should be ordered T={g1,g2,\ldots,gN} such that if gi is included
#                  in gj, then j <= i. g1 should be the group at the root of the tree
#                  and contains every variable.
#                * Every group is a set of  contiguous indices for instance
#                  gi={3,4,5} or gi={4,5,6,7} or gi={4}, but not {3,5};
#                * We define root(gi) as the indices of the variables that are in gi,
#                  but not in its descendants. For instance for
```

```
#                 T={ g1={1,2,3,4},g2={2,3},g3={4} }, then, root(g1)={1},
#                 root(g2)={2,3}, root(g3)={4},
#                 We assume that for all i, root(gi) is a set of contigous variables
#               * We assume that the smallest of root(gi) is also the smallest index of gi.
#
#               For instance,
#                 T={ g1={1,2,3,4},g2={2,3},g3={4} }, is a valid set of groups.
#                 but we can not have
#                 T={ g1={1,2,3,4},g2={1,2},g3={3} }, since root(g1)={4} and 4 is not the
#                 smallest element in g1.
#
#            We do not lose generality with these assumptions since they can be fullfilled for
    any
#            tree-structured set of groups after a permutation of variables and a correct
   ordering of the
#            groups.
#            see more examples in test_ProximalTree.m of valid tree-structured sets of groups.
#
#            The first fields sets the weights for every group
#               tree['eta_g']            double N vector
#
#            The next field sets inclusion relations between groups
#            (but not between groups and variables):
#               tree['groups']           sparse (double or boolean) N x N matrix
#               the (i,j) entry is non-zero if and only if i is different than j and
#               gi is included in gj.
#               the first column corresponds to the group at the root of the tree.
#
#            The next field define the smallest index of each group gi,
#            which is also the smallest index of root(gi)
#            tree['own_variables']    int32 N vector
#
#            The next field define for each group gi, the size of root(gi)
#            tree['N_own_variables']  int32 N vector
#
#            examples are given in test_ProximalTree.m
#
#       return_val_loss:
#               if true the function will return a tuple of matrices.
#       lambda1:  (regularization parameter)
#       regul: (choice of regularization, see above)
#       lambda2:  (optional, regularization parameter)
#       lambda3:  (optional, regularization parameter)
#       verbose: (optional, verbosity level, false by default)
#       intercept: (optional, last row of U is not regularized,
#         false by default)
#       pos: (optional, adds positivity constraints on the
#         coefficients, false by default)
#       transpose: (optional, transpose the matrix in the regularization function)
#       size_group: (optional, for regularization functions assuming a group
#         structure). It is a scalar. When groups is not specified, it assumes
#         that the groups are the sets of consecutive elements of size size_group
#       numThreads: (optional, number of threads for exploiting
#         multi-core / multi-cpus. By default, it takes the value -1,
#         which automatically selects all the available CPUs/cores).
#       resetflow:    undocumented; modify at your own risks!
#       clever:    undocumented; modify at your own risks!
#
# Output:
#       V: double m x n matrix (output coefficients)
#       val_regularizer: double 1 x n vector (value of the regularization
#       term at the optimum).
#       val_loss:        vector of size U.shape[1]
#         alpha = spams.proximalTree(U,tree,return_val_loss = False,...)
#         (alpha,val_loss) = spams.proximalTree(U,tree,return_val_loss = True,...)
#
# Authors:
```

```
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     Valid values for the regularization parameter (regul) are:
#       "l0", "l1", "l2", "linf", "l2-not-squared", "elastic-net", "fused-lasso",
#       "group-lasso-l2", "group-lasso-linf", "sparse-group-lasso-l2",
#       "sparse-group-lasso-linf", "l1l2", "l1linf", "l1l2+l1", "l1linf+l1",
#       "tree-l0", "tree-l2", "tree-linf", "graph", "graph-ridge", "graph-l2",
#       "multi-task-tree", "multi-task-graph", "l1linf-row-column", "trace-norm",
#       "trace-norm-vec", "rank", "rank-vec", "none"
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
param = {'numThreads' : -1,'verbose' : True,
         'pos' : False, 'intercept' : False, 'lambda1' : 0.1 }
m = 10;n = 1000
U = np.asfortranarray(np.random.normal(size = (m,n)))
print 'First tree example'
# Example 1 of tree structure
# tree structured groups:
# g1= {0 1 2 3 4 5 6 7 8 9}
# g2= {2 3 4}
# g3= {5 6 7 8 9}
own_variables =  np.array([0,2,5],dtype=np.int32) # pointer to the first variable of each group
N_own_variables =  np.array([2,3,5],dtype=np.int32) # number of "root" variables in each group
# (variables that are in a group, but not in its descendants).
# for instance root(g1)={0,1}, root(g2)={2 3 4}, root(g3)={5 6 7 8 9}
eta_g = np.array([1,1,1],dtype=np.float64) # weights for each group, they should be non-zero to
    use fenchel duality
groups = np.asfortranarray([[0,0,0],
                            [1,0,0],
                            [1,0,0]],dtype = np.bool)
# first group should always be the root of the tree
# non-zero entriees mean inclusion relation ship, here g2 is a children of g1,
# g3 is a children of g1
groups = ssp.csc_matrix(groups,dtype=np.bool)
tree = {'eta_g': eta_g,'groups' : groups,'own_variables' : own_variables,
        'N_own_variables' : N_own_variables}
print '\ntest prox tree-l0'
param['regul'] = 'tree-l2'
alpha = spams.proximalTree(U,tree,False,**param)

print '\ntest prox tree-linf'
param['regul'] = 'tree-linf'
alpha = spams.proximalTree(U,tree,False,**param)

print 'Second tree example'
# Example 2 of tree structure
# tree structured groups:
# g1= {0 1 2 3 4 5 6 7 8 9}     root(g1) = { };
# g2= {0 1 2 3 4 5}             root(g2) = {0 1 2};
# g3= {3 4}                     root(g3) = {3 4};
# g4= {5}                       root(g4) = {5};
# g5= {6 7 8 9}                 root(g5) = { };
# g6= {6 7}                     root(g6) = {6 7};
# g7= {8 9}                     root(g7) = {8};
# g8 = {9}                      root(g8) = {9};
own_variables =  np.array([0, 0, 3, 5, 6, 6, 8, 9],dtype=np.int32)
N_own_variables =  np.array([0,3,2,1,0,2,1,1],dtype=np.int32)
eta_g = np.array([1,1,1,2,2,2,2.5,2.5],dtype=np.float64)
groups = np.asfortranarray([[0,0,0,0,0,0,0,0],
                [1,0,0,0,0,0,0,0],
                [0,1,0,0,0,0,0,0],
                [0,1,0,0,0,0,0,0],
```

```
                [1,0,0,0,0,0,0,0],
                [0,0,0,0,1,0,0,0],
                [0,0,0,0,1,0,0,0],
                [0,0,0,0,0,0,1,0]],dtype = np.bool)
groups = ssp.csc_matrix(groups,dtype=np.bool)
tree = {'eta_g': eta_g,'groups' : groups, 'own_variables' : own_variables,
        'N_own_variables' : N_own_variables}
print '\ntest prox tree-l0'
param['regul'] = 'tree-l0'
alpha = spams.proximalTree(U,tree,False,**param)

print '\ntest prox tree-l2'
param['regul'] = 'tree-l2'
alpha = spams.proximalTree(U,tree,False,**param)

print '\ntest prox tree-linf'
param['regul'] = 'tree-linf'
alpha = spams.proximalTree(U,tree,False,**param)

# mexProximalTree also works with non-tree-structured regularization functions
print '\nprox l1, intercept, positivity constraint'
param['regul'] = 'l1'
param['pos'] = True       # can be used with all the other regularizations
param['intercept'] = True # can be used with all the other regularizations
alpha = spams.proximalTree(U,tree,False,**param)

print '\nprox multi-task tree'
param['pos'] = False
param['intercept'] = False
param['lambda2'] = param['lambda1']
param['regul'] = 'multi-task-tree'
alpha = spams.proximalTree(U,tree,False,**param)
```

5.4 Function spams.proximalGraph

This function computes the proximal operators associated to structured sparse regularization, for input signals $\mathbf{U} = [\mathbf{u}^1, \ldots, \mathbf{u}^n]$ in $\mathbb{R}^{p \times n}$, and a set of groups [21], it returns a matrix $\mathbf{V} = [\mathbf{v}^1, \ldots, \mathbf{v}^n]$ in $\mathbb{R}^{p \times n}$. When one uses a regularization function on vectors, it computes a column $\mathbf{v}$ of $\mathbf{V}$ for every column $\mathbf{u}$ of $\mathbf{U}$:

$$\min_{\mathbf{v} \in \mathbb{R}^p} \frac{1}{2}\|\mathbf{u} - \mathbf{v}\|_2^2 + \lambda \sum_{g \in \mathcal{G}} \eta^g \|\mathbf{v}_g\|_\infty, \tag{44}$$

or with a regularization function on matrices, it computes $\mathbf{V}$ solving

$$\min_{\mathbf{V} \in \mathbb{R}^{p \times n}} \frac{1}{2}\|\mathbf{U} - \mathbf{V}\|_F^2 + \lambda \sum_{i=1}^{n} \sum_{g \in \mathcal{G}} \eta^g \|\mathbf{v}_g^i\|_\infty + \lambda_2 \sum_{g \in \mathcal{G}} \max_{j \in g} \|\mathbf{v}_g^j\|_\infty, \tag{45}$$

This function can also be used for computing the proximal operators addressed by spams.proximalFlat. The way the graph is incoded is presented below (and also in the example file test_ProximalGraph.m, with more usage details:

```
#
# Name: proximalGraph
#
# Usage: spams.proximalGraph(U,graph,return_val_loss = False,numThreads =-1,lambda1=1.0,lambda2
    =0.,
#                      lambda3=0.,intercept=False,resetflow=False,regul="",verbose=False,
#                      pos=False,clever=True,eval= None,size_group=1,transpose=False)
#
# Description:
#     proximalGraph computes a proximal operator. Depending
#         on the value of regul, it computes
#
#         Given an input matrix U=[u^1,\ldots,u^n], and a set of groups G,
```

```
#         it computes a matrix V=[v^1,\ldots,v^n] such that
#
#         if regul='graph'
#         for every column u of U, it computes a column v of V solving
#             argmin 0.5||u-v||_2^2 + lambda1\sum_{g \in G} \eta_g||v_g||_inf
#
#         if regul='graph+ridge'
#         for every column u of U, it computes a column v of V solving
#             argmin 0.5||u-v||_2^2 + lambda1\sum_{g \in G} \eta_g||v_g||_inf + lambda1_2||v||
    _2^2
#
#
#         if regul='multi-task-graph'
#            V=argmin 0.5||U-V||_F^2 + lambda1 \sum_{i=1}^n\sum_{g \in G} \eta_g||v^i_g||_inf +
     ...
#                                                 lambda1_2 \sum_{g \in G} \eta_g max_{j in g}||
    V_j||_{inf}
#
#         it can also be used with any regularization addressed by proximalFlat
#
#         for all these regularizations, it is possible to enforce non-negativity constraints
#         with the option pos, and to prevent the last row of U to be regularized, with
#         the option intercept
#
# Inputs:
#       U:  double p x n matrix   (input signals)
#                m is the signal size
#       graph: struct
#             with three fields, eta_g, groups, and groups_var
#
#             The first fields sets the weights for every group
#                graph.eta_g            double N vector
#
#             The next field sets inclusion relations between groups
#             (but not between groups and variables):
#                graph.groups           sparse (double or boolean) N x N matrix
#                the (i,j) entry is non-zero if and only if i is different than j and
#                gi is included in gj.
#
#             The next field sets inclusion relations between groups and variables
#                graph.groups_var       sparse (double or boolean) p x N matrix
#                the (i,j) entry is non-zero if and only if the variable i is included
#                in gj, but not in any children of gj.
#
#             examples are given in test_ProximalGraph.m
#
#       return_val_loss:
#               if true the function will return a tuple of matrices.
#       lambda1:  (regularization parameter)
#       regul: (choice of regularization, see above)
#       lambda2:  (optional, regularization parameter)
#       lambda3:  (optional, regularization parameter)
#       verbose: (optional, verbosity level, false by default)
#       intercept: (optional, last row of U is not regularized,
#         false by default)
#       pos: (optional, adds positivity constraints on the
#         coefficients, false by default)
#       numThreads: (optional, number of threads for exploiting
#         multi-core / multi-cpus. By default, it takes the value -1,
#         which automatically selects all the available CPUs/cores).
#       resetflow:    undocumented; modify at your own risks!
#       clever:    undocumented; modify at your own risks!
#       size_group:    undocumented; modify at your own risks!
#       transpose:    undocumented; modify at your own risks!
#
# Output:
#       V: double p x n matrix (output coefficients)
```

```
#       val_regularizer: double 1 x n vector (value of the regularization
#       term at the optimum).
#       val_loss:        vector of size U.shape[1]
#         alpha = spams.proximalGraph(U,graph,return_val_loss = False,...)
#         (alpha,val_loss) = spams.proximalGraph(U,graph,return_val_loss = True,...)
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     Valid values for the regularization parameter (regul) are:
#       "l0", "l1", "l2", "linf", "l2-not-squared", "elastic-net", "fused-lasso",
#       "group-lasso-l2", "group-lasso-linf", "sparse-group-lasso-l2",
#       "sparse-group-lasso-linf", "l1l2", "l1linf", "l1l2+l1", "l1linf+l1",
#       "tree-l0", "tree-l2", "tree-linf", "graph", "graph-ridge", "graph-l2",
#       "multi-task-tree", "multi-task-graph", "l1linf-row-column", "trace-norm",
#       "trace-norm-vec", "rank", "rank-vec", "none"
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
lambda1 = 0.1 # regularization parameter
num_threads = -1 # all cores (-1 by default)
verbose = True  # verbosity, false by default
pos = False      # can be used with all the other regularizations
intercept = False # can be used with all the other regularizations

U = np.asfortranarray(np.random.normal(size = (10,100)))
print 'First graph example'
# Example 1 of graph structure
# groups:
# g1= {0 1 2 3}
# g2= {3 4 5 6}
# g3= {6 7 8 9}
eta_g = np.array([1, 1, 1],dtype=np.float64)
groups = ssp.csc_matrix(np.zeros((3,3)),dtype = np.bool)
groups_var = ssp.csc_matrix(
    np.array([[1, 0, 0],
              [1, 0, 0],
              [1, 0, 0],
              [1, 1, 0],
              [0, 1, 0],
              [0, 1, 0],
              [0, 1, 1],
              [0, 0, 1],
              [0, 0, 1],
              [0, 0, 1]],dtype=np.bool),dtype=np.bool)
graph = {'eta_g': eta_g,'groups' : groups,'groups_var' : groups_var}

print '\ntest prox graph'
regul='graph'
alpha = spams.proximalGraph(U,graph,False,lambda1 = lambda1,numThreads  = num_threads ,verbose
    = verbose,pos = pos,intercept = intercept,regul = regul)

# Example 2 of graph structure
# groups:
# g1= {0 1 2 3}
# g2= {3 4 5 6}
# g3= {6 7 8 9}
# g4= {0 1 2 3 4 5}
# g5= {6 7 8}
eta_g = np.array([1, 1, 1, 1, 1],dtype=np.float64)
groups = ssp.csc_matrix(
    np.array([[0, 0, 0, 1, 0],
```

```
                [0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0],
                [0, 0, 1, 0, 0]],dtype=np.bool),dtype=np.bool)

groups_var = ssp.csc_matrix(
    np.array([[1, 0, 0, 0, 0],
                [1, 0, 0, 0, 0],
                [1, 0, 0, 0, 0],
                [1, 1, 0, 0, 0],
                [0, 1, 0, 1, 0],
                [0, 1, 0, 1, 0],
                [0, 1, 0, 0, 1],
                [0, 0, 0, 0, 1],
                [0, 0, 0, 0, 1],
                [0, 0, 1, 0, 0]],dtype=np.bool),dtype=np.bool)
graph = {'eta_g': eta_g,'groups' : groups,'groups_var' : groups_var}
print '\ntest prox graph'
alpha = spams.proximalGraph(U,graph,False,lambda1 = lambda1,numThreads  = num_threads ,verbose
    = verbose,pos = pos,intercept = intercept,regul = regul)
#
print '\ntest prox multi-task-graph'
regul = 'multi-task-graph'
lambda2 = 0.1
alpha = spams.proximalGraph(U,graph,False,lambda1 = lambda1,lambda2 = lambda2,numThreads  =
    num_threads ,verbose = verbose,pos = pos,intercept = intercept,regul = regul)
#
print '\ntest no regularization'
regul = 'none'
alpha = spams.proximalGraph(U,graph,False,lambda1 = lambda1,lambda2 = lambda2,numThreads  =
    num_threads ,verbose = verbose,pos = pos,intercept = intercept,regul = regul)
```

5.5 Function spams.proximalPathCoding

This function computes the proximal operators associated to the path coding penalties of [23].

```
#
# The python function is not yet implemented.
#
```

This function is associated to a function to evaluate the penalties:

5.6 Function spams.evalPathCoding

```
#
# The python function is not yet implemented.
#
```

After having presented the regularization terms which our software can handle, we present the various formulations that we address

5.7 Problems Addressed

We present here regression or classification formulations and their multi-task variants.

5.7.1 Regression Problems with the Square Loss

Given a training set $\{\mathbf{x}^i, y_i\}_{i=1}^n$, with $\mathbf{x}^i \in \mathbb{R}^p$ and $y_i \in \mathbb{R}$ for all i in $[1;n]$, we address

$$\min_{\mathbf{w}\in\mathbb{R}^p, b\in\mathbb{R}} \sum_{i=1}^{n} \frac{1}{2}(y_i - \mathbf{w}^\top \mathbf{x}^i - b)^2 + \lambda\psi(\mathbf{w}),$$

where b is an optional variable acting as an "intercept", which is not regularized, and ψ can be any of the regularization functions presented above. Let us consider the vector $\mathbf{y}$ in $\mathbb{R}^n$ that carries the entries y_i. The

problem without the intercept takes the following form, which we have already encountered in the previous toolbox, but with different notations:

$$\min_{\mathbf{w}\in\mathbb{R}^p} \frac{1}{2}\|\mathbf{y}-\mathbf{X}\mathbf{w}\|_2^2 + \lambda\psi(\mathbf{w}),$$

where the $\mathbf{X} = [\mathbf{x}^i, \ldots, \mathbf{x}^n]^T$ (the $\mathbf{x}^i$'s are here the rows of $\mathbf{X}$).

5.7.2 Classification Problems with the Logistic Loss

The next formulation that our software can solve is the regularized logistic regression formulation. We are again given a training set $\{\mathbf{x}^i, y_i\}_{i=1}^n$, with $\mathbf{x}^i \in \mathbb{R}^p$, but the variables y_i are now in $\{-1, +1\}$ for all i in $[1; n]$. The optimization problem we address is

$$\min_{\mathbf{w}\in\mathbb{R}^p, b\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^n \log(1 + e^{-y_i(\mathbf{w}^\top\mathbf{x}^i + b)} + \lambda\psi(\mathbf{w}),$$

with again ψ taken to be one of the regularization function presented above, and b is an optional intercept.

5.7.3 Multi-class Classification Problems with the Softmax Loss

We have also implemented a multi-class logistic classifier (or softmax). For a classification problem with r classes, we are given a training set $\{\mathbf{x}^i, y_i\}_{i=1}^n$, where the variables $\mathbf{x}^i$ are still vectors in $\mathbb{R}^p$, but the y_i's have integer values in $\{1, 2, \ldots, r\}$. The formulation we address is the following multi-class learning problem

$$\min_{\mathbf{W}\in\mathbb{R}^{p\times r}, \mathbf{b}\in\mathbb{R}^r} \frac{1}{n}\sum_{i=1}^n \log\Big(\sum_{j=1}^r e^{(\mathbf{w}^j - \mathbf{w}^{\mathbf{y}_i})^\top\mathbf{x}^i + \mathbf{b}_j - \mathbf{b}_{\mathbf{y}_i}}\Big) + \lambda\sum_{j=1}^r \psi(\mathbf{w}^j), \tag{46}$$

where $\mathbf{W} = [\mathbf{w}^1, \ldots, \mathbf{w}^r]$ and the optional vector $\mathbf{b}$ in $\mathbb{R}^r$ carries intercepts for each class.

5.7.4 Multi-task Regression Problems with the Square Loss

We are now considering a problem with r tasks, and a training set $\{\mathbf{x}^i, \mathbf{y}^i\}_{i=1}^n$, where the variables $\mathbf{x}^i$ are still vectors in $\mathbb{R}^p$, and $\mathbf{y}^i$ is a vector in $\mathbb{R}^r$. We are looking for r regression vectors $\mathbf{w}^j$, for $j \in [1; r]$, or equivalently for a matrix $\mathbf{W} = [\mathbf{w}^1, \ldots, \mathbf{w}^r]$ in $\mathbb{R}^{p\times r}$. The formulation we address is the following multi-task regression problem

$$\min_{\mathbf{W}\in\mathbb{R}^{p\times r}, \mathbf{b}\in\mathbb{R}^r} \sum_{j=1}^r\sum_{i=1}^n \frac{1}{2}(\mathbf{y}_j^i - \mathbf{w}^\top\mathbf{x}^i - \mathbf{b}_j)^2 + \lambda\psi(\mathbf{W}),$$

where ψ is any of the regularization function on matrices we have presented in the previous section. Note that by introducing the appropriate variables $\mathbf{Y}$, the problem without intercept could be equivalently rewritten

$$\min_{\mathbf{W}\in\mathbb{R}^{p\times r}} \frac{1}{2}\|\mathbf{Y} - \mathbf{X}\mathbf{W}\|_{\mathrm{F}}^2 + \lambda\psi(\mathbf{W}).$$

5.7.5 Multi-task Classification Problems with the Logistic Loss

The multi-task version of the logistic regression follows the same principle. We consider r tasks, and a training set $\{\mathbf{x}^i, \mathbf{y}^i\}_{i=1}^n$, with the $\mathbf{x}^i$'s in $\mathbb{R}^p$, and the $\mathbf{y}^i$'s are vectors in $\{-1, +1\}^r$. We look for a matrix $\mathbf{W} = [\mathbf{w}^1, \ldots, \mathbf{w}^r]$ in $\mathbb{R}^{p\times r}$. The formulation is the following multi-task regression problem

$$\min_{\mathbf{W}\in\mathbb{R}^{p\times r}, \mathbf{b}\in\mathbb{R}^r} \sum_{j=1}^r \frac{1}{n}\sum_{i=1}^n \log\Big(1 + e^{-\mathbf{y}_j^i(\mathbf{w}^\top\mathbf{x}^i + \mathbf{b}_j)}\Big) + \lambda\psi(\mathbf{W}).$$

5.7.6 Multi-task and Multi-class Classification Problems with the Softmax Loss

The multi-task/multi-class version directly follows from the formulation of Eq. (46), but associates with each class a task, and as a consequence, regularizes the matrix $\mathbf{W}$ in a particular way:

$$\min_{\mathbf{W}\in\mathbb{R}^{p\times r},\mathbf{b}\in\mathbb{R}^r} \frac{1}{n}\sum_{i=1}^{n}\log\Big(\sum_{j=1}^{r} e^{(\mathbf{w}^j-\mathbf{w}^{\mathbf{y}_i})^\top\mathbf{x}^i+\mathbf{b}_j-\mathbf{b}_{\mathbf{y}_i}}\Big)+\lambda\psi(\mathbf{W}).$$

How duality gaps are computed for any of these formulations is presented in Appendix A. We now present the main functions for solving these problems

5.8 Function spams.fistaFlat

Given a matrix $\mathbf{X}=[\mathbf{x}^1,\ldots,\mathbf{x}^p]^T$ in $\mathbb{R}^{m\times p}$, and a matrix $\mathbf{Y}=[\mathbf{y}^1,\ldots,\mathbf{y}^n]$, it solves the optimization problems presented in the previous section, with the same regularization functions as spams.proximalFlat. see usage details below:

```
#
# Name: fistaFlat
#
# Usage: spams.fistaFlat(Y,X,W0,return_optim_info = False,numThreads =-1,max_it =1000,L0=1.0,
#                 fixed_step=False,gamma=1.5,lambda1=1.0,delta=1.0,lambda2=0.,lambda3=0.,
#                 a=1.0,b=0.,c=1.0,tol=0.000001,it0=100,max_iter_backtracking=1000,
#                 compute_gram=False,lin_admm=False,admm=False,intercept=False,
#                 resetflow=False,regul="",loss="",verbose=False,pos=False,clever=False,
#                 log=False,ista=False,subgrad=False,logName="",is_inner_weights=False,
#                 inner_weights=np.array([0.]),size_group=1,groups = None,sqrt_step=True,
#                 transpose=False)
#
# Description:
#     fistaFlat solves sparse regularized problems.
#         X is a design matrix of size m x p
#         X=[x^1,...,x^n]', where the x_i's are the rows of X
#         Y=[y^1,...,y^n] is a matrix of size m x n
#         It implements the algorithms FISTA, ISTA and subgradient descent.
#
#           - if loss='square' and regul is a regularization function for vectors,
#             the entries of Y are real-valued,  W = [w^1,...,w^n] is a matrix of size p x n
#             For all column y of Y, it computes a column w of W such that
#               w = argmin 0.5||y- X w||_2^2 + lambda1 psi(w)
#
#           - if loss='square' and regul is a regularization function for matrices
#             the entries of Y are real-valued,  W is a matrix of size p x n.
#             It computes the matrix W such that
#               W = argmin 0.5||Y- X W||_F^2 + lambda1 psi(W)
#
#           - loss='square-missing' same as loss='square', but handles missing data
#             represented by NaN (not a number) in the matrix Y
#
#           - if loss='logistic' and regul is a regularization function for vectors,
#             the entries of Y are either -1 or +1, W = [w^1,...,w^n] is a matrix of size p x n
#             For all column y of Y, it computes a column w of W such that
#               w = argmin (1/m)sum_{j=1}^m log(1+e^(-y_j x^j' w)) + lambda1 psi(w),
#             where x^j is the j-th row of X.
#
#           - if loss='logistic' and regul is a regularization function for matrices
#             the entries of Y are either -1 or +1, W is a matrix of size p x n
#               W = argmin sum_{i=1}^n(1/m)sum_{j=1}^m log(1+e^(-y^i_j x^j' w^i)) + lambda1 psi
    (W)
#
#           - if loss='multi-logistic' and regul is a regularization function for vectors,
#             the entries of Y are in {0,1,...,N} where N is the total number of classes
#            W = [W^1,...,W^n] is a matrix of size p x Nn, each submatrix W^i is of size p x N
#             for all submatrix WW of W, and column y of Y, it computes
#               WW = argmin (1/m)sum_{j=1}^m log(sum_{j=1}^r e^(x^j'(ww^j-ww^{y_j}))) + lambda1
     sum_{j=1}^N psi(ww^j),
```

```
#             where ww^j is the j-th column of WW.
#
#           - if loss='multi-logistic' and regul is a regularization function for matrices,
#             the entries of Y are in {0,1,...,N} where N is the total number of classes
#             W is a matrix of size p x N, it computes
#               W = argmin (1/m)sum_{j=1}^m log(sum_{j=1}^r e^(x^j'(w^j-w^{y_j}))) + lambda1
    psi(W)
#             where ww^j is the j-th column of WW.
#
#           - loss='cur' useful to perform sparse CUR matrix decompositions,
#               W = argmin 0.5||Y-X*W*X||_F^2 + lambda1 psi(W)
#
#
#         The function psi are those used by proximalFlat (see documentation)
#
#         This function can also handle intercepts (last row of W is not regularized),
#         and/or non-negativity constraints on W, and sparse matrices for X
#
# Inputs:
#       Y:  double dense m x n matrix
#       X:  double dense or sparse m x p matrix
#       W0:  double dense p x n matrix or p x Nn matrix (for multi-logistic loss)
#            initial guess
#       return_optim_info:
#               if true the function will return a tuple of matrices.
#       loss: (choice of loss, see above)
#       regul: (choice of regularization, see function proximalFlat)
#       lambda1: (regularization parameter)
#       lambda2: (optional, regularization parameter, 0 by default)
#       lambda3: (optional, regularization parameter, 0 by default)
#       verbose: (optional, verbosity level, false by default)
#       pos: (optional, adds positivity constraints on the
#           coefficients, false by default)
#       transpose: (optional, transpose the matrix in the regularization function)
#       size_group: (optional, for regularization functions assuming a group
#            structure)
#       groups: (int32, optional, for regularization functions assuming a group
#            structure, see proximalFlat)
#       numThreads: (optional, number of threads for exploiting
#           multi-core / multi-cpus. By default, it takes the value -1,
#           which automatically selects all the available CPUs/cores).
#       max_it: (optional, maximum number of iterations, 100 by default)
#       it0: (optional, frequency for computing duality gap, every 10 iterations by default)
#       tol: (optional, tolerance for stopping criteration, which is a relative duality gap
#           if it is available, or a relative change of parameters).
#       gamma: (optional, multiplier for increasing the parameter L in fista, 1.5 by default)
#       L0: (optional, initial parameter L in fista, 0.1 by default, should be small enough)
#       fixed_step: (deactive the line search for L in fista and use L0 instead)
#       compute_gram: (optional, pre-compute X^TX, false by default).
#       intercept: (optional, do not regularize last row of W, false by default).
#       ista: (optional, use ista instead of fista, false by default).
#       subgrad: (optional, if not ista, use subradient descent instead of fista, false by
    default).
#       a:
#       b: (optional, if subgrad, the gradient step is a/(t+b)
#       also similar options as proximalFlat
#
#       the function also implements the ADMM algorithm via an option admm=true. It is not
    documented
#       and you need to look at the source code to use it.
#       delta:    undocumented; modify at your own risks!
#       c:    undocumented; modify at your own risks!
#       max_iter_backtracking:    undocumented; modify at your own risks!
#       lin_admm:    undocumented; modify at your own risks!
#       admm:    undocumented; modify at your own risks!
#       resetflow:    undocumented; modify at your own risks!
#       clever:    undocumented; modify at your own risks!
```

```
#       log:    undocumented; modify at your own risks!
#       logName:    undocumented; modify at your own risks!
#       is_inner_weights:    undocumented; modify at your own risks!
#       inner_weights:    undocumented; modify at your own risks!
#       sqrt_step:    undocumented; modify at your own risks!
#
# Output:
#       W:  double dense p x n matrix or p x Nn matrix (for multi-logistic loss)
#       optim: optional, double dense 4 x n matrix.
#           first row: values of the objective functions.
#           third row: values of the relative duality gap (if available)
#           fourth row: number of iterations
#       optim_info:         vector of size 4, containing information of the optimization.
#            W = spams.fistaFlat(Y,X,W0,return_optim_info = False,...)
#            (W,optim_info) = spams.fistaFlat(Y,X,W0,return_optim_info = True,...)
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     Valid values for the regularization parameter (regul) are:
#       "l0", "l1", "l2", "linf", "l2-not-squared", "elastic-net", "fused-lasso",
#       "group-lasso-l2", "group-lasso-linf", "sparse-group-lasso-l2",
#       "sparse-group-lasso-linf", "l1l2", "l1linf", "l1l2+l1", "l1linf+l1",
#       "tree-l0", "tree-l2", "tree-linf", "graph", "graph-ridge", "graph-l2",
#       "multi-task-tree", "multi-task-graph", "l1linf-row-column", "trace-norm",
#       "trace-norm-vec", "rank", "rank-vec", "none"
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
param = {'numThreads' : 1,'verbose' : True,
         'lambda1' : 0.05, 'it0' : 10, 'max_it' : 200,
         'L0' : 0.1, 'tol' : 1e-3, 'intercept' : False,
         'pos' : False}
np.random.seed(0)
m = 100;n = 200
X = np.asfortranarray(np.random.normal(size = (m,n)))
X = np.asfortranarray(X - np.tile(np.mean(X,0),(X.shape[0],1)))
X = spams.normalize(X)
Y = np.asfortranarray(np.random.normal(size = (m,1)))
Y = np.asfortranarray(Y - np.tile(np.mean(Y,0),(Y.shape[0],1)))
Y = spams.normalize(Y)
W0 = np.zeros((X.shape[1],Y.shape[1]),dtype=np.float64,order="FORTRAN")
# Regression experiments
# 100 regression problems with the same design matrix X.
print '\nVarious regression experiments'
param['compute_gram'] = True
print '\nFISTA + Regression l1'
param['loss'] = 'square'
param['regul'] = 'l1'
# param.regul='group-lasso-l2';
# param.size_group=10;
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
##     print "XX %s" %str(optim_info.shape);return None
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:],0),np.mean(optim_info[2,:],0),np.mean(optim_info[3,:],0))
###
print '\nISTA + Regression l1'
param['ista'] = True
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f\n' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))
##
print '\nSubgradient Descent + Regression l1'
```

```
param['ista'] = False
param['subgrad'] = True
param['a'] = 0.1
param['b'] = 1000 # arbitrary parameters
max_it = param['max_it']
it0 = param['it0']
param['max_it'] = 500
param['it0'] = 50
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f\n' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))
param['subgrad'] = False
param['max_it'] = max_it
param['it0'] = it0

###
print '\nFISTA + Regression l2'
param['regul'] = 'l2'
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f\n' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))
###
print '\nFISTA + Regression l2 + sparse feature matrix'
param['regul'] = 'l2';
(W, optim_info) = spams.fistaFlat(Y,ssp.csc_matrix(X),W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))
###########

print '\nFISTA + Regression Elastic-Net'
param['regul'] = 'elastic-net'
param['lambda2'] = 0.1
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:]),np.mean(optim_info
    [3,:]))

print '\nFISTA + Group Lasso L2'
param['regul'] = 'group-lasso-l2'
param['size_group'] = 2
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:],0),np.mean(optim_info[2,:],0),np.mean(optim_info[3,:],0))

print '\nFISTA + Group Lasso L2 with variable size of groups'
param['regul'] = 'group-lasso-l2'
param2=param.copy()
param2['groups'] = np.array(np.random.random_integers(1,5,X.shape[1]),dtype = np.int32)
param2['lambda1'] *= 10
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:],0),np.mean(optim_info[2,:],0),np.mean(optim_info[3,:],0))

print '\nFISTA + Trace Norm'
param['regul'] = 'trace-norm-vec'
param['size_group'] = 5
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:],0),np.mean(optim_info[3,:]))

####

print '\nFISTA + Regression Fused-Lasso'
param['regul'] = 'fused-lasso'
param['lambda2'] = 0.1
param['lambda3'] = 0.1; #
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
```

```
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:]),np.mean(optim_info
    [3,:]))

print '\nFISTA + Regression no regularization'
param['regul'] = 'none'
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:]),np.mean(optim_info
    [3,:]))


print '\nFISTA + Regression l1 with intercept '
param['intercept'] = True
param['regul'] = 'l1'
x1 = np.asfortranarray(np.concatenate((X,np.ones((X.shape[0],1))),1))
W01 = np.asfortranarray(np.concatenate((W0,np.zeros((1,W0.shape[1]))),0))
(W, optim_info) = spams.fistaFlat(Y,x1,W01,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))

print '\nFISTA + Regression l1 with intercept+ non-negative '
param['pos'] = True
param['regul'] = 'l1'
x1 = np.asfortranarray(np.concatenate((X,np.ones((X.shape[0],1))),1))
W01 = np.asfortranarray(np.concatenate((W0,np.zeros((1,W0.shape[1]))),0))
(W, optim_info) = spams.fistaFlat(Y,x1,W01,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:]),np.mean(optim_info
    [3,:]))
param['pos'] = False
param['intercept'] = False

print '\nISTA + Regression l0'
param['regul'] = 'l0'
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:]),np.mean(optim_info
    [3,:]))

# Classification

print '\nOne classification experiment'
Y = np.asfortranarray(2 * np.asarray(np.random.normal(size = (100,1)) > 0,dtype='float64') - 1)
print '\nFISTA + Logistic l1'
param['regul'] = 'l1'
param['loss'] = 'logistic'
param['lambda1'] = 0.01
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))
# can be used of course with other regularization functions, intercept,...
param['regul'] = 'l1'
param['loss'] = 'weighted-logistic'
param['lambda1'] = 0.01
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))
# can be used of course with other regularization functions, intercept,...

print '\nFISTA + Logistic l1 + sparse matrix'
param['loss'] = 'logistic'
(W, optim_info) = spams.fistaFlat(Y,ssp.csc_matrix(X),W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))
# can be used of course with other regularization functions, intercept,...


# Multi-Class classification
Y = np.asfortranarray(np.ceil(5 * np.random.random(size = (100,1000))) - 1)
param['loss'] = 'multi-logistic'
```

```
print '\nFISTA + Multi-Class Logistic l1'
nclasses = np.max(Y[:])+1
W0 = np.zeros((X.shape[1],nclasses * Y.shape[1]),dtype=np.float64,order="FORTRAN")
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)

print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))
# can be used of course with other regularization functions, intercept,...


# Multi-Task regression
Y = np.asfortranarray(np.random.normal(size = (100,100)))
Y = np.asfortranarray(Y - np.tile(np.mean(Y,0),(Y.shape[0],1)))
Y = spams.normalize(Y)
param['compute_gram'] = False
W0 = np.zeros((X.shape[1],Y.shape[1]),dtype=np.float64,order="FORTRAN")
param['loss'] = 'square'
print '\nFISTA + Regression l1l2 '
param['regul'] = 'l1l2'
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))

print '\nFISTA + Regression l1linf '
param['regul'] = 'l1linf'
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))


print '\nFISTA + Regression l1l2 + l1 '
param['regul'] = 'l1l2+l1'
param['lambda2'] = 0.1
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:]),np.mean(optim_info
    [3,:]))


print '\nFISTA + Regression l1linf + l1 '
param['regul'] = 'l1linf+l1'
param['lambda2'] = 0.1
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:]),np.mean(optim_info
    [3,:]))


print '\nFISTA + Regression l1linf + row + columns '
param['regul'] = 'l1linf-row-column'
param['lambda2'] = 0.1
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))

# Multi-Task Classification

print '\nFISTA + Logistic + l1l2 '
param['regul'] = 'l1l2'
param['loss'] = 'logistic'
Y = np.asfortranarray(2 * np.asarray(np.random.normal(size = (100,100)) > 1,dtype='float64') -
    1)
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))
# Multi-Class + Multi-Task Regularization


print '\nFISTA + Multi-Class Logistic l1l2 '
```

```
Y = np.asfortranarray(np.ceil(5 * np.random.random(size = (100,1000))) - 1)
Y = spams.normalize(Y)
param['loss'] = 'multi-logistic'
param['regul'] = 'l1l2'
nclasses = np.max(Y[:])+1
W0 = np.zeros((X.shape[1],nclasses * Y.shape[1]),dtype=np.float64,order="FORTRAN")
(W, optim_info) = spams.fistaFlat(Y,X,W0,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:]),np.mean(optim_info[2,:]),np.mean(optim_info[3,:]))
# can be used of course with other regularization functions, intercept,...


#############
```

5.9 Function spams.fistaTree

Given a matrix $\mathbf{X} = [\mathbf{x}^1, \ldots, \mathbf{x}^p]^T$ in $\mathbb{R}^{m \times p}$, and a matrix $\mathbf{Y} = [\mathbf{y}^1, \ldots, \mathbf{y}^n]$, it solves the optimization problems presented in the previous section, with the same regularization functions as spams.proximalTree. see usage details below:

```
#
# Name: fistaTree
#
# Usage: spams.fistaTree(Y,X,W0,tree,return_optim_info = False,numThreads =-1,max_it =1000,L0
    =1.0,
#                fixed_step=False,gamma=1.5,lambda1=1.0,delta=1.0,lambda2=0.,lambda3=0.,
#                a=1.0,b=0.,c=1.0,tol=0.000001,it0=100,max_iter_backtracking=1000,
#                compute_gram=False,lin_admm=False,admm=False,intercept=False,
#                resetflow=False,regul="",loss="",verbose=False,pos=False,clever=False,
#                log=False,ista=False,subgrad=False,logName="",is_inner_weights=False,
#                inner_weights=np.array([0.]),size_group=1,sqrt_step=True,transpose=False)
#
# Description:
#     fistaTree solves sparse regularized problems.
#         X is a design matrix of size m x p
#         X=[x^1,...,x^n]', where the x_i's are the rows of X
#         Y=[y^1,...,y^n] is a matrix of size m x n
#         It implements the algorithms FISTA, ISTA and subgradient descent for solving
#
#           min_W  loss(W) + lambda1 psi(W)
#
#         The function psi are those used by proximalTree (see documentation)
#         for the loss functions, see the documentation of fistaFlat
#
#         This function can also handle intercepts (last row of W is not regularized),
#         and/or non-negativity constraints on W and sparse matrices X
#
# Inputs:
#       Y:  double dense m x n matrix
#       X:  double dense or sparse m x p matrix
#       W0:  double dense p x n matrix or p x Nn matrix (for multi-logistic loss)
#            initial guess
#       tree: named list (see documentation of proximalTree)
#       return_optim_info:
#               if true the function will return a tuple of matrices.
#       loss: (choice of loss, see above)
#       regul: (choice of regularization, see function proximalFlat)
#       lambda1: (regularization parameter)
#       lambda2: (optional, regularization parameter, 0 by default)
#       lambda3: (optional, regularization parameter, 0 by default)
#       verbose: (optional, verbosity level, false by default)
#       pos: (optional, adds positivity constraints on the
#           coefficients, false by default)
#       transpose: (optional, transpose the matrix in the regularization function)
#       size_group: (optional, for regularization functions assuming a group
#            structure)
```

```
#       numThreads: (optional, number of threads for exploiting
#           multi-core / multi-cpus. By default, it takes the value -1,
#           which automatically selects all the available CPUs/cores).
#       max_it: (optional, maximum number of iterations, 100 by default)
#       it0: (optional, frequency for computing duality gap, every 10 iterations by default)
#       tol: (optional, tolerance for stopping criteration, which is a relative duality gap
#           if it is available, or a relative change of parameters).
#       gamma: (optional, multiplier for increasing the parameter L in fista, 1.5 by default)
#       L0: (optional, initial parameter L in fista, 0.1 by default, should be small enough)
#       fixed_step: (deactive the line search for L in fista and use L0 instead)
#       compute_gram: (optional, pre-compute X^TX, false by default).
#       intercept: (optional, do not regularize last row of W, false by default).
#       ista: (optional, use ista instead of fista, false by default).
#       subgrad: (optional, if not ista, use subradient descent instead of fista, false by
    default).
#       a:
#       b: (optional, if subgrad, the gradient step is a/(t+b)
#       also similar options as proximalTree
#
#       the function also implements the ADMM algorithm via an option admm=true. It is not
    documented
#       and you need to look at the source code to use it.
#       delta:    undocumented; modify at your own risks!
#       c:    undocumented; modify at your own risks!
#       max_iter_backtracking:    undocumented; modify at your own risks!
#       lin_admm:    undocumented; modify at your own risks!
#       admm:    undocumented; modify at your own risks!
#       resetflow:    undocumented; modify at your own risks!
#       clever:    undocumented; modify at your own risks!
#       log:    undocumented; modify at your own risks!
#       logName:    undocumented; modify at your own risks!
#       is_inner_weights:    undocumented; modify at your own risks!
#       inner_weights:    undocumented; modify at your own risks!
#       sqrt_step:    undocumented; modify at your own risks!
#
# Output:
#       W:  double dense p x n matrix or p x Nn matrix (for multi-logistic loss)
#       optim: optional, double dense 4 x n matrix.
#           first row: values of the objective functions.
#           third row: values of the relative duality gap (if available)
#           fourth row: number of iterations
#       optim_info:        vector of size 4, containing information of the optimization.
#            W = spams.fistaTree(Y,X,W0,tree,return_optim_info = False,...)
#            (W,optim_info) = spams.fistaTree(Y,X,W0,tree,return_optim_info = True,...)
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     Valid values for the regularization parameter (regul) are:
#       "l0", "l1", "l2", "linf", "l2-not-squared", "elastic-net", "fused-lasso",
#       "group-lasso-l2", "group-lasso-linf", "sparse-group-lasso-l2",
#       "sparse-group-lasso-linf", "l1l2", "l1linf", "l1l2+l1", "l1linf+l1",
#       "tree-l0", "tree-l2", "tree-linf", "graph", "graph-ridge", "graph-l2",
#       "multi-task-tree", "multi-task-graph", "l1linf-row-column", "trace-norm",
#       "trace-norm-vec", "rank", "rank-vec", "none"
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
param = {'numThreads' : -1,'verbose' : False,
         'lambda1' : 0.001, 'it0' : 10, 'max_it' : 200,
         'L0' : 0.1, 'tol' : 1e-5, 'intercept' : False,
         'pos' : False}
np.random.seed(0)
```

```
m = 100;n = 10
X = np.asfortranarray(np.random.normal(size = (m,n)))
X = np.asfortranarray(X - np.tile(np.mean(X,0),(X.shape[0],1)))
X = spams.normalize(X)
Y = np.asfortranarray(np.random.normal(size = (m,m)))
Y = np.asfortranarray(Y - np.tile(np.mean(Y,0),(Y.shape[0],1)))
Y = spams.normalize(Y)
W0 = np.zeros((X.shape[1],Y.shape[1]),dtype=np.float64,order="FORTRAN")
own_variables =  np.array([0,0,3,5,6,6,8,9],dtype=np.int32)
N_own_variables =  np.array([0,3,2,1,0,2,1,1],dtype=np.int32)
eta_g = np.array([1,1,1,2,2,2,2.5,2.5],dtype=np.float64)
groups = np.asfortranarray([[0, 0, 0, 0, 0, 0, 0, 0],
          [1, 0, 0, 0, 0, 0, 0, 0],
          [0, 1, 0, 0, 0, 0, 0, 0],
          [0, 1, 0, 0, 0, 0, 0, 0],
          [1, 0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 1, 0, 0, 0],
          [0, 0, 0, 0, 1, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 1, 0]],dtype = np.bool)
groups = ssp.csc_matrix(groups,dtype=np.bool)
tree = {'eta_g': eta_g,'groups' : groups,'own_variables' : own_variables,
        'N_own_variables' : N_own_variables}
print '\nVarious regression experiments'
param['compute_gram'] = True

print '\nFISTA + Regression tree-l2'
param['loss'] = 'square'
param['regul'] = 'tree-l2'
(W, optim_info) = spams.fistaTree(Y,X,W0,tree,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:],0),np.mean(optim_info
    [3,:],0))
###
print '\nFISTA + Regression tree-linf'
param['regul'] = 'tree-linf'
(W, optim_info) = spams.fistaTree(Y,X,W0,tree,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:],0),np.mean(optim_info[2,:]),np.mean(optim_info[3,:],0))
###
# works also with non tree-structured regularization. tree is ignored
print '\nFISTA + Regression Fused-Lasso'
param['regul'] = 'fused-lasso'
param['lambda2'] = 0.001
param['lambda3'] = 0.001
(W, optim_info) = spams.fistaTree(Y,X,W0,tree,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:],0),np.mean(optim_info
    [3,:],0))
###
print '\nISTA + Regression tree-l0'
param['regul'] = 'tree-l0'
(W, optim_info) = spams.fistaTree(Y,X,W0,tree,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:],0),np.mean(optim_info
    [3,:],0))
###
print '\nFISTA + Regression tree-l2 with intercept'
param['intercept'] = True
param['regul'] = 'tree-l2'
x1 = np.asfortranarray(np.concatenate((X,np.ones((X.shape[0],1))),1))
W01 = np.asfortranarray(np.concatenate((W0,np.zeros((1,W0.shape[1]))),0))
(W, optim_info) = spams.fistaTree(Y,x1,W01,tree,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:],0),np.mean(optim_info
    [3,:],0))
###
param['intercept'] = False

#    Classification

print '\nOne classification experiment'
```

```
Y = np.asfortranarray(2 * np.asarray(np.random.normal(size = (100,Y.shape[1])) > 0,dtype='
    float64') - 1)
print '\nFISTA + Logistic + tree-linf'
param['regul'] = 'tree-linf'
param['loss'] = 'logistic'
param['lambda1'] = 0.001
(W, optim_info) = spams.fistaTree(Y,X,W0,tree,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:],0),np.mean(optim_info[2,:]),np.mean(optim_info[3,:],0))
###
# can be used of course with other regularization functions, intercept,...

#  Multi-Class classification
Y = np.asfortranarray(np.ceil(5 * np.random.random(size = (100,Y.shape[1]))) - 1)
param['loss'] = 'multi-logistic'
param['regul'] = 'tree-l2'
print '\nFISTA + Multi-Class Logistic + tree-l2'
nclasses = np.max(Y[:])+1
W0 = np.zeros((X.shape[1],nclasses * Y.shape[1]),dtype=np.float64,order="FORTRAN")
(W, optim_info) = spams.fistaTree(Y,X,W0,tree,True,**param)
print 'mean loss: %f, number of iterations: %f' %(np.mean(optim_info[0,:],0),np.mean(optim_info
    [3,:],0))
# can be used of course with other regularization functions, intercept,...

# Multi-Task regression
Y = np.asfortranarray(np.random.normal(size = (100,100)))
Y = np.asfortranarray(Y - np.tile(np.mean(Y,0),(Y.shape[0],1)))
Y = spams.normalize(Y)
param['compute_gram'] = False
param['verbose'] = True;    # verbosity, False by default
W0 = np.zeros((X.shape[1],Y.shape[1]),dtype=np.float64,order="FORTRAN")
param['loss'] = 'square'
print '\nFISTA + Regression  multi-task-tree'
param['regul'] = 'multi-task-tree'
param['lambda2'] = 0.001
(W, optim_info) = spams.fistaTree(Y,X,W0,tree,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:],0),np.mean(optim_info[2,:]),np.mean(optim_info[3,:],0))

# Multi-Task Classification
print '\nFISTA + Logistic + multi-task-tree'
param['regul'] = 'multi-task-tree'
param['lambda2'] = 0.001
param['loss'] = 'logistic'
Y = np.asfortranarray(2 * np.asarray(np.random.normal(size = (100,Y.shape[1])) > 0,dtype='
    float64') - 1)
(W, optim_info) = spams.fistaTree(Y,X,W0,tree,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:],0),np.mean(optim_info[2,:]),np.mean(optim_info[3,:],0))

#  Multi-Class + Multi-Task Regularization
param['verbose'] = False
print '\nFISTA + Multi-Class Logistic +multi-task-tree'
Y = np.asfortranarray(np.ceil(5 * np.random.random(size = (100,Y.shape[1]))) - 1)
param['loss'] = 'multi-logistic'
param['regul'] = 'multi-task-tree'
nclasses = np.max(Y[:])+1
W0 = np.zeros((X.shape[1],nclasses * Y.shape[1]),dtype=np.float64,order="FORTRAN")
(W, optim_info) = spams.fistaTree(Y,X,W0,tree,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:],0),np.mean(optim_info[2,:]),np.mean(optim_info[3,:],0))
# can be used of course with other regularization functions, intercept,...

print '\nFISTA + Multi-Class Logistic +multi-task-tree + sparse matrix'
nclasses = np.max(Y[:])+1
W0 = np.zeros((X.shape[1],nclasses * Y.shape[1]),dtype=np.float64,order="FORTRAN")
X2 = ssp.csc_matrix(X)
```

```
(W, optim_info) = spams.fistaTree(Y,X2,W0,tree,True,**param)
print 'mean loss: %f, mean relative duality_gap: %f, number of iterations: %f' %(np.mean(
    optim_info[0,:],0),np.mean(optim_info[2,:]),np.mean(optim_info[3,:],0))
```

5.10 Function spams.fistaGraph

Given a matrix $\mathbf{X} = [\mathbf{x}^1, \ldots, \mathbf{x}^p]^T$ in $\mathbb{R}^{m \times p}$, and a matrix $\mathbf{Y} = [\mathbf{y}^1, \ldots, \mathbf{y}^n]$, it solves the optimization problems presented in the previous section, with the same regularization functions as spams.proximalGraph. see usage details below:

```
#
# Name: fistaGraph
#
# Usage: spams.fistaGraph(Y,X,W0,graph,return_optim_info = False,numThreads =-1,max_it =1000,L0
    =1.0,
#                  fixed_step=False,gamma=1.5,lambda1=1.0,delta=1.0,lambda2=0.,lambda3=0.,
#                  a=1.0,b=0.,c=1.0,tol=0.000001,it0=100,max_iter_backtracking=1000,
#                  compute_gram=False,lin_admm=False,admm=False,intercept=False,
#                  resetflow=False,regul="",loss="",verbose=False,pos=False,clever=False,
#                  log=False,ista=False,subgrad=False,logName="",is_inner_weights=False,
#                  inner_weights=np.array([0.]),size_group=1,sqrt_step=True,transpose=False)
#
# Description:
#     fistaGraph solves sparse regularized problems.
#         X is a design matrix of size m x p
#         X=[x^1,...,x^n]', where the x_i's are the rows of X
#         Y=[y^1,...,y^n] is a matrix of size m x n
#         It implements the algorithms FISTA, ISTA and subgradient descent.
#
#         It implements the algorithms FISTA, ISTA and subgradient descent for solving
#
#           min_W  loss(W) + lambda1 psi(W)
#
#         The function psi are those used by proximalGraph (see documentation)
#         for the loss functions, see the documentation of fistaFlat
#
#         This function can also handle intercepts (last row of W is not regularized),
#         and/or non-negativity constraints on W.
#
# Inputs:
#       Y:  double dense m x n matrix
#       X:  double dense or sparse m x p matrix
#       W0:  double dense p x n matrix or p x Nn matrix (for multi-logistic loss)
#            initial guess
#       graph: struct (see documentation of proximalGraph)
#       return_optim_info:
#               if true the function will return a tuple of matrices.
#       loss: (choice of loss, see above)
#       regul: (choice of regularization, see function proximalFlat)
#       lambda1: (regularization parameter)
#       lambda2: (optional, regularization parameter, 0 by default)
#       lambda3: (optional, regularization parameter, 0 by default)
#       verbose: (optional, verbosity level, false by default)
#       pos: (optional, adds positivity constraints on the
#           coefficients, false by default)
#       numThreads: (optional, number of threads for exploiting
#           multi-core / multi-cpus. By default, it takes the value -1,
#           which automatically selects all the available CPUs/cores).
#       max_it: (optional, maximum number of iterations, 100 by default)
#       it0: (optional, frequency for computing duality gap, every 10 iterations by default)
#       tol: (optional, tolerance for stopping criteration, which is a relative duality gap
#           if it is available, or a relative change of parameters).
#       gamma: (optional, multiplier for increasing the parameter L in fista, 1.5 by default)
#       L0: (optional, initial parameter L in fista, 0.1 by default, should be small enough)
#       fixed_step: (deactive the line search for L in fista and use L0 instead)
#       compute_gram: (optional, pre-compute X^TX, false by default).
```

```
#       intercept: (optional, do not regularize last row of W, false by default).
#       ista: (optional, use ista instead of fista, false by default).
#       subgrad: (optional, if not ista, use subradient descent instead of fista, false by
    default).
#       a:
#       b: (optional, if subgrad, the gradient step is a/(t+b)
#       also similar options as proximalTree
#
#       the function also implements the ADMM algorithm via an option admm=true. It is not
    documented
#       and you need to look at the source code to use it.
#       delta:    undocumented; modify at your own risks!
#       c:    undocumented; modify at your own risks!
#       max_iter_backtracking:    undocumented; modify at your own risks!
#       lin_admm:    undocumented; modify at your own risks!
#       admm:    undocumented; modify at your own risks!
#       resetflow:    undocumented; modify at your own risks!
#       clever:    undocumented; modify at your own risks!
#       log:    undocumented; modify at your own risks!
#       logName:    undocumented; modify at your own risks!
#       is_inner_weights:    undocumented; modify at your own risks!
#       inner_weights:    undocumented; modify at your own risks!
#       sqrt_step:    undocumented; modify at your own risks!
#       size_group:    undocumented; modify at your own risks!
#       transpose:    undocumented; modify at your own risks!
#
# Output:
#       W:  double dense p x n matrix or p x Nn matrix (for multi-logistic loss)
#       optim: optional, double dense 4 x n matrix.
#           first row: values of the objective functions.
#           third row: values of the relative duality gap (if available)
#           fourth row: number of iterations
#       optim_info:        vector of size 4, containing information of the optimization.
#            W = spams.fistaGraph(Y,X,W0,graph,return_optim_info = False,...)
#            (W,optim_info) = spams.fistaGraph(Y,X,W0,graph,return_optim_info = True,...)
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
# Note:
#     Valid values for the regularization parameter (regul) are:
#       "l0", "l1", "l2", "linf", "l2-not-squared", "elastic-net", "fused-lasso",
#       "group-lasso-l2", "group-lasso-linf", "sparse-group-lasso-l2",
#       "sparse-group-lasso-linf", "l1l2", "l1linf", "l1l2+l1", "l1linf+l1",
#       "tree-l0", "tree-l2", "tree-linf", "graph", "graph-ridge", "graph-l2",
#       "multi-task-tree", "multi-task-graph", "l1linf-row-column", "trace-norm",
#       "trace-norm-vec", "rank", "rank-vec", "none"
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
np.random.seed(0)
num_threads = -1 # all cores (-1 by default)
verbose = False   # verbosity, false by default
lambda1 = 0.1 # regularization ter
it0 = 1      # frequency for duality gap computations
max_it = 100 # maximum number of iterations
L0 = 0.1
tol = 1e-5
intercept = False
pos = False

eta_g = np.array([1, 1, 1, 1, 1],dtype=np.float64)

groups = ssp.csc_matrix(np.array([[0, 0, 0, 1, 0],
```

```
                    [0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0],
                    [0, 0, 1, 0, 0]],dtype=np.bool),dtype=np.bool)

groups_var = ssp.csc_matrix(np.array([[1, 0, 0, 0, 0],
                        [1, 0, 0, 0, 0],
                        [1, 0, 0, 0, 0],
                        [1, 1, 0, 0, 0],
                        [0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0],
                        [0, 1, 0, 0, 1],
                        [0, 0, 0, 0, 1],
                        [0, 0, 0, 0, 1],
                        [0, 0, 1, 0, 0]],dtype=np.bool),dtype=np.bool)

graph = {'eta_g': eta_g,'groups' : groups,'groups_var' : groups_var}

verbose = True
X = np.asfortranarray(np.random.normal(size = (100,10)))
X = np.asfortranarray(X - np.tile(np.mean(X,0),(X.shape[0],1)))
X = spams.normalize(X)
Y = np.asfortranarray(np.random.normal(size = (100,1)))
Y = np.asfortranarray(Y - np.tile(np.mean(Y,0),(Y.shape[0],1)))
Y = spams.normalize(Y)
W0 = np.zeros((X.shape[1],Y.shape[1]),dtype=np.float64,order="FORTRAN")
# Regression experiments
# 100 regression problems with the same design matrix X.
print '\nVarious regression experiments'
compute_gram = True
#
print '\nFISTA + Regression graph'
loss = 'square'
regul = 'graph'
tic = time.time()
(W, optim_info) = spams.fistaGraph(
    Y,X,W0,graph,True,numThreads = num_threads,verbose = verbose,
    lambda1 = lambda1,it0 = it0,max_it = max_it,L0 = L0,tol = tol,
    intercept = intercept,pos = pos,compute_gram = compute_gram,
    loss = loss,regul = regul)
tac = time.time()
t = tac - tic
print 'mean loss: %f, mean relative duality_gap: %f, time: %f, number of iterations: %f' %(np.
    mean(optim_info[0,:]),np.mean(optim_info[2,:]),t,np.mean(optim_info[3,:]))
#
print '\nADMM + Regression graph'
admm = True
lin_admm = True
c = 1
delta = 1
tic = time.time()
(W, optim_info) = spams.fistaGraph(
    Y,X,W0,graph,True,numThreads = num_threads,verbose = verbose,
    lambda1 = lambda1,it0 = it0,max_it = max_it,L0 = L0,tol = tol,
    intercept = intercept,pos = pos,compute_gram = compute_gram,
    loss = loss,regul = regul,admm = admm,lin_admm = lin_admm,c = c,delta = delta)
tac = time.time()
t = tac - tic
print 'mean loss: %f, mean relative duality_gap: %f, time: %f, number of iterations: %f' %(np.
    mean(optim_info[0,:]),np.mean(optim_info[2,:]),t,np.mean(optim_info[3,:]))
#
admm = False
max_it = 5
it0 = 1
tic = time.time()
(W, optim_info) = spams.fistaGraph(
    Y,X,W0,graph,True,numThreads = num_threads,verbose = verbose,
```

```
    lambda1 = lambda1, it0 = it0, max_it = max_it, L0 = L0, tol = tol,
    intercept = intercept, pos = pos, compute_gram = compute_gram,
    loss = loss, regul = regul, admm = admm, lin_admm = lin_admm, c = c, delta = delta)
tac = time.time()
t = tac - tic
print 'mean loss: %f, mean relative duality_gap: %f, time: %f, number of iterations: %f' %(np.
    mean(optim_info[0,:]),np.mean(optim_info[2,:]),t,np.mean(optim_info[3,:]))
#
#  works also with non graph-structured regularization. graph is ignored
print '\nFISTA + Regression Fused-Lasso'
regul = 'fused-lasso'
lambda2 = 0.01
lambda3 = 0.01
tic = time.time()
(W, optim_info) = spams.fistaGraph(
    Y,X,W0,graph,True,numThreads = num_threads,verbose = verbose,
    lambda1 = lambda1, it0 = it0, max_it = max_it, L0 = L0, tol = tol,
    intercept = intercept, pos = pos, compute_gram = compute_gram,
    loss = loss, regul = regul, admm = admm, lin_admm = lin_admm, c = c,
    lambda2 = lambda2, lambda3 = lambda3, delta = delta)
tac = time.time()
t = tac - tic
print 'mean loss: %f, time: %f, number of iterations: %f' %(np.mean(optim_info[0,:]),t,np.mean(
    optim_info[3,:]))
#
print '\nFISTA + Regression graph with intercept'
regul = 'graph'
intercept = True
tic = time.time()
(W, optim_info) = spams.fistaGraph(
    Y,X,W0,graph,True,numThreads = num_threads,verbose = verbose,
    lambda1 = lambda1, it0 = it0, max_it = max_it, L0 = L0, tol = tol,
    intercept = intercept, pos = pos, compute_gram = compute_gram,
    loss = loss, regul = regul, admm = admm, lin_admm = lin_admm, c = c,
    lambda2 = lambda2, lambda3 = lambda3, delta = delta)
tac = time.time()
t = tac - tic
print 'mean loss: %f, mean relative duality_gap: %f, time: %f, number of iterations: %f' %(np.
    mean(optim_info[0,:]),np.mean(optim_info[2,:]),t,np.mean(optim_info[3,:]))
intercept = False

# Classification
print '\nOne classification experiment'
Y = np.asfortranarray( 2 * np.asfortranarray(np.random.normal(size = (100,Y.shape[1])) > 0,
    dtype = np.float64) -1)
print '\nFISTA +  Logistic + graph-linf'
loss = 'logistic'
regul = 'graph'
lambda1 = 0.01
tic = time.time()
(W, optim_info) = spams.fistaGraph(
    Y,X,W0,graph,True,numThreads = num_threads,verbose = verbose,
    lambda1 = lambda1, it0 = it0, max_it = max_it, L0 = L0, tol = tol,
    intercept = intercept, pos = pos, compute_gram = compute_gram,
    loss = loss, regul = regul, admm = admm, lin_admm = lin_admm, c = c,
    lambda2 = lambda2, lambda3 = lambda3, delta = delta)
tac = time.time()
t = tac - tic
print 'mean loss: %f, mean relative duality_gap: %f, time: %f, number of iterations: %f' %(np.
    mean(optim_info[0,:]),np.mean(optim_info[2,:]),t,np.mean(optim_info[3,:]))
#
# can be used of course with other regularization functions, intercept,...

# Multi-Class classification

Y = np.asfortranarray(np.ceil(5 * np.random.random(size = (100,Y.shape[1]))) - 1)
loss = 'multi-logistic'
```

```
regul = 'graph'
print '\nFISTA + Multi-Class Logistic + graph'
nclasses = np.max(Y) + 1
W0 = np.zeros((X.shape[1],nclasses * Y.shape[1]),dtype=np.float64,order="FORTRAN")
tic = time.time()
(W, optim_info) = spams.fistaGraph(
    Y,X,W0,graph,True,numThreads = num_threads,verbose = verbose,
    lambda1 = lambda1,it0 = it0,max_it = max_it,L0 = L0,tol = tol,
    intercept = intercept,pos = pos,compute_gram = compute_gram,
    loss = loss,regul = regul,admm = admm,lin_admm = lin_admm,c = c,
    lambda2 = lambda2,lambda3 = lambda3,delta = delta)
tac = time.time()
t = tac - tic
print 'mean loss: %f, mean relative duality_gap: %f, time: %f, number of iterations: %f' %(np.
    mean(optim_info[0,:]),np.mean(optim_info[2,:]),t,np.mean(optim_info[3,:]))
#
# can be used of course with other regularization functions, intercept,...
# Multi-Task regression
Y = np.asfortranarray(np.random.normal(size = (100,Y.shape[1])))
Y = np.asfortranarray(Y - np.tile(np.mean(Y,0),(Y.shape[0],1)))
Y = spams.normalize(Y)
W0 = W0 = np.zeros((X.shape[1],Y.shape[1]),dtype=np.float64,order="FORTRAN")
compute_gram = False
verbose = True
loss = 'square'
print '\nFISTA + Regression multi-task-graph'
regul = 'multi-task-graph'
lambda2 = 0.01
tic = time.time()
(W, optim_info) = spams.fistaGraph(
    Y,X,W0,graph,True,numThreads = num_threads,verbose = verbose,
    lambda1 = lambda1,it0 = it0,max_it = max_it,L0 = L0,tol = tol,
    intercept = intercept,pos = pos,compute_gram = compute_gram,
    loss = loss,regul = regul,admm = admm,lin_admm = lin_admm,c = c,
    lambda2 = lambda2,lambda3 = lambda3,delta = delta)
tac = time.time()
t = tac - tic
print 'mean loss: %f, mean relative duality_gap: %f, time: %f, number of iterations: %f' %(np.
    mean(optim_info[0,:]),np.mean(optim_info[2,:]),t,np.mean(optim_info[3,:]))
#
# Multi-Task Classification
print '\nFISTA + Logistic + multi-task-graph'
regul = 'multi-task-graph'
lambda2 = 0.01
loss = 'logistic'
Y = np.asfortranarray( 2 * np.asfortranarray(np.random.normal(size = (100,Y.shape[1])) > 0,
    dtype = np.float64) -1)
tic = time.time()
(W, optim_info) = spams.fistaGraph(
    Y,X,W0,graph,True,numThreads = num_threads,verbose = verbose,
    lambda1 = lambda1,it0 = it0,max_it = max_it,L0 = L0,tol = tol,
    intercept = intercept,pos = pos,compute_gram = compute_gram,
    loss = loss,regul = regul,admm = admm,lin_admm = lin_admm,c = c,
    lambda2 = lambda2,lambda3 = lambda3,delta = delta)
tac = time.time()
t = tac - tic
print 'mean loss: %f, mean relative duality_gap: %f, time: %f, number of iterations: %f' %(np.
    mean(optim_info[0,:]),np.mean(optim_info[2,:]),t,np.mean(optim_info[3,:]))
# Multi-Class + Multi-Task Regularization
verbose = False
print '\nFISTA + Multi-Class Logistic +multi-task-graph'
Y = np.asfortranarray(np.ceil(5 * np.random.random(size = (100,Y.shape[1]))) - 1)
loss = 'multi-logistic'
regul = 'multi-task-graph'
nclasses = np.max(Y) + 1
W0 = np.zeros((X.shape[1],nclasses * Y.shape[1]),dtype=np.float64,order="FORTRAN")
tic = time.time()
```

```
(W, optim_info) = spams.fistaGraph(
    Y,X,W0,graph,True,numThreads = num_threads,verbose = verbose,
    lambda1 = lambda1,it0 = it0,max_it = max_it,L0 = L0,tol = tol,
    intercept = intercept,pos = pos,compute_gram = compute_gram,
    loss = loss,regul = regul,admm = admm,lin_admm = lin_admm,c = c,
    lambda2 = lambda2,lambda3 = lambda3,delta = delta)
tac = time.time()
t = tac - tic
print 'mean loss: %f, mean relative duality_gap: %f, time: %f, number of iterations: %f' %(np.
    mean(optim_info[0,:]),np.mean(optim_info[2,:]),t,np.mean(optim_info[3,:]))
# can be used of course with other regularization functions, intercept,...
```

5.11 Function spams.fistaPathCoding

Similarly, the toolbox handles the penalties of [23] with the following function

```
#
# The python function is not yet implemented.
#
```

6 Miscellaneous Functions

6.1 Function spams.conjGrad

Implementation of a conjugate gradient for solving a linear system $\mathbf{Ax} = \mathbf{b}$ when $\mathbf{A}$ is positive definite. In some cases, it is faster than the Matlab function `pcg`, especially when the library uses the Intel Math Kernel Library.

```
#
# Name: conjGrad
#
# Usage: spams.conjGrad(A,b,x0 = None,tol = 1e-10,itermax = None)
#
# Description:
#     Conjugate gradient algorithm, sometimes faster than the
#    equivalent python function solve. In order to solve Ax=b;
#
# Inputs:
#       A:  double square n x n matrix. HAS TO BE POSITIVE DEFINITE
#       b:  double vector of length n.
#       x0: double vector of length n. (optional) initial guess.
#       tol: (optional) tolerance.
#       itermax: (optional) maximum number of iterations.
#
# Output:
#       x: double vector of length n.
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
A = np.asfortranarray(np.random.normal(size = (5000,500)))
A = np.asfortranarray(np.dot(A.T,A))
b = np.ones((A.shape[1],),dtype=np.float64,order="FORTRAN")
x0 = b
tol = 1e-4
itermax = int(0.5 * len(b))

tic = time.time()
```

```
for i in xrange(0,20):
    y1 = np.linalg.solve(A,b)
tac = time.time()
print "  Time (numpy): ", tac - tic
x1 = np.abs(b - np.dot(A,y1))
print "Mean error on b : %f" %(x1.sum() / b.shape[0])

tic = time.time()
for i in xrange(0,20):
    y2 = spams.conjGrad(A,b,x0,tol,itermax)
tac = time.time()
print "  Time (spams): ", tac - tic
x1 = np.dot(A,y2)
x2 = np.abs(b - x1)
print "Mean error on b : %f" %(x2.sum() / b.shape[0])

err = abs(y1 - y2)
```

6.2 Function spams.bayer

Apply a Bayer pattern to an input image

```
#
# Name: conjGrad
#
# Usage: spams.conjGrad(A,b,x0 = None,tol = 1e-10,itermax = None)
#
# Description:
#     Conjugate gradient algorithm, sometimes faster than the
#    equivalent python function solve. In order to solve Ax=b;
#
# Inputs:
#       A:  double square n x n matrix. HAS TO BE POSITIVE DEFINITE
#       b:  double vector of length n.
#       x0: double vector of length n. (optional) initial guess.
#       tol: (optional) tolerance.
#       itermax: (optional) maximum number of iterations.
#
# Output:
#       x: double vector of length n.
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
A = np.asfortranarray(np.random.normal(size = (5000,500)))
A = np.asfortranarray(np.dot(A.T,A))
b = np.ones((A.shape[1],),dtype=np.float64,order="FORTRAN")
x0 = b
tol = 1e-4
itermax = int(0.5 * len(b))

tic = time.time()
for i in xrange(0,20):
    y1 = np.linalg.solve(A,b)
tac = time.time()
print "  Time (numpy): ", tac - tic
x1 = np.abs(b - np.dot(A,y1))
print "Mean error on b : %f" %(x1.sum() / b.shape[0])

tic = time.time()
for i in xrange(0,20):
```

```
    y2 = spams.conjGrad(A,b,x0,tol,itermax)
tac = time.time()
print "  Time (spams): ", tac - tic
x1 = np.dot(A,y2)
x2 = np.abs(b - x1)
print "Mean error on b : %f" %(x2.sum() / b.shape[0])

err = abs(y1 - y2)
```

6.3 Function spams.calcAAt

For an input sparse matrix $\mathbf{A}$, this function returns $\mathbf{A}\mathbf{A}^T$. For some reasons, when $\mathbf{A}$ has a lot more columns than rows, this function can be much faster than the equivalent matlab command `X*A'`.

```
#
# Name: calcAAt
#
# Usage: spams.calcAAt(A)
#
# Description:
#     Compute efficiently AAt = A*A', when A is sparse
#   and has a lot more columns than rows. In some cases, it is
#   up to 20 times faster than the equivalent python expression
#   AAt=A*A';
#
# Inputs:
#       A:  double sparse m x n matrix
#
# Output:
#       AAt: double m x m matrix
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
"""
test A * A'
"""
m=200; n = 200000; d= 0.05
A = ssprand(m,n,density=d,format='csc',dtype=np.float64)
result = spams.calcAAt(A)
```

6.4 Function spams.calcXAt

For an input sparse matrix $\mathbf{A}$ and a matrix $\mathbf{X}$, this function returns $\mathbf{X}\mathbf{A}^T$. For some reasons, when $\mathbf{A}$ has a lot more columns than rows, this function can be much faster than the equivalent matlab command `X*A'`.

```
#
# Name: calcXAt
#
# Usage: spams.calcXAt(X,A)
#
# Description:
#     Compute efficiently XAt = X*A', when A is sparse and has a
#   lot more columns than rows. In some cases, it is up to 20 times
#   faster than the equivalent python expression;
#
# Inputs:
#       X:  double m x n matrix
#       A:  double sparse p x n matrix
```

```
#
# Output:
#       XAt: double m x p matrix
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
m=200; n = 200000; d= 0.05
A = ssprand(m,n,density=d,format='csc',dtype=np.float64)
X = np.asfortranarray(np.random.normal(size = (64,n)))

result = spams.calcXAt(X,A)
```

6.5 Function spams.calcXY

For two input matrices $\mathbf{X}$ and $\mathbf{Y}$, this function returns $\mathbf{XY}$.

```
#
# Name: calcXY
#
# Usage: spams.calcXY(X,Y)
#
# Description:
#     Compute Z=XY using the BLAS library used by SPAMS.
#
# Inputs:
#       X:  double m x n matrix
#       Y:  double n x p matrix
#
# Output:
#       Z: double m x p matrix
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
X = np.asfortranarray(np.random.normal(size = (64,200)))
Y = np.asfortranarray(np.random.normal(size = (200,20000)))
result = spams.calcXY(X,Y)
```

6.6 Function spams.calcXYt

For two input matrices $\mathbf{X}$ and $\mathbf{Y}$, this function returns $\mathbf{XY}^T$.

```
#
# Name: calcXYt
#
# Usage: spams.calcXYt(X,Y)
#
# Description:
#     Compute Z=XY' using the BLAS library used by SPAMS.
#
# Inputs:
#       X:  double m x n matrix
#       Y:  double p x n matrix
```

```
#
# Output:
#       Z: double m x p matrix
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
X = np.asfortranarray(np.random.normal(size = (64,200)))
Y = np.asfortranarray(np.random.normal(size = (20000,200)))
result = spams.calcXYt(X,Y)
```

6.7 Function spams.calcXtY

For two input matrices $\mathbf{X}$ and $\mathbf{Y}$, this function returns $\mathbf{X}^T\mathbf{Y}$.

```
#
# Name: calcXtY
#
# Usage: spams.calcXtY(X,Y)
#
# Description:
#     Compute Z=X'Y using the BLAS library used by SPAMS.
#
# Inputs:
#       X:  double n x m matrix
#       Y:  double n x p matrix
#
# Output:
#       Z: double m x p matrix
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
X = np.asfortranarray(np.random.normal(size = (200,64)))
Y = np.asfortranarray(np.random.normal(size = (200,20000)))
result = spams.calcXtY(X,Y)
```

6.8 Function spams.invSym

For an input symmetric matrices $\mathbf{A}$ in $\mathbb{R}^{n\times n}$, this function returns $\mathbf{A}^{-1}$.

```
#
# Name: invSym
#
# Usage: spams.invSym(A)
#
# Description:
#     returns the inverse of a symmetric matrix A
#
# Inputs:
#       A:  double n x n matrix
#
# Output:
#       B: double n x n matrix
```

```
#
# Authors:
# Julien MAIRAL, 2009 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
A = np.asfortranarray(np.random.random(size = (1000,1000)))
A =np.asfortranarray( np.dot(A.T,A))
result = spams.invSym(A)
```

6.9 Function spams.normalize

```
#
# Name: normalize
#
# Usage: spams.normalize(A)
#
# Description:
#     rescale the columns of X so that they have
#        unit l2-norm.
#
# Inputs:
#       X:  double m x n matrix
#
# Output:
#       Y: double m x n matrix
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
A = np.asfortranarray(np.random.random(size = (100,1000)))
res2 = spams.normalize(A)
```

6.10 Function spams.sort

```
#
# Name: sort
#
# Usage: spams.sort(X,mode=True)
#
# Description:
#     sort the elements of X using quicksort
#
# Inputs:
#       X:  double vector of size n
#       mode: false for decreasing order (true by default)
#
# Output:
#       Y: double  vector of size n
#
# Authors:
# Julien MAIRAL, 2010 (spams, matlab interface and documentation)
# Jean-Paul CHIEZE 2011-2012 (python interface)
#
```

The following piece of code contains usage examples:

```
import spams
import numpy as np
n = 2000000
X = np.random.normal(size = (n,))
result = spams.sort(X,True)
```

6.11 Function mexDisplayPatches

Print to the screen a matrix containing as columns image patches.

6.12 Function spams.countPathsDAG

This function counts the number of paths in a DAG using dynamic programming.

```
#
# The python function is not yet implemented.
#
```

6.13 Function spams.removeCyclesGraph

One heuristic to remove cycles from a graph.

```
#
# The python function is not yet implemented.
#
```

6.14 Function spams.countConnexComponents

Count the number of connected components of a subgraph from a graph.

```
#
# The python function is not yet implemented.
#
```

A Duality Gaps with Fenchel Duality

This section is taken from the appendix D of Julien Mairal's PhD thesis [18]. We are going to use intensively Fenchel Duality (see [2]). Let us consider the problem

$$\min_{\mathbf{w}\in\mathbb{R}^p} [g(\mathbf{w}) \triangleq f(\mathbf{w}) + \lambda\psi(\mathbf{w})], \tag{47}$$

We first notice that for all the formulations we have been interested in, $g(\mathbf{w})$ can be rewritten

$$g(\mathbf{w}) = \tilde{f}(\mathbf{X}^\top \mathbf{w}) + \lambda\psi(\mathbf{w}), \tag{48}$$

where $\mathbf{X} = [\mathbf{x}^1, \ldots, \mathbf{x}^n]$ are training vectors, and $\tilde{f}$ is an appropriated smooth real-valued function of $\mathbb{R}^n$, and ψ one of the regularization functions we have introduced.

Given a primal variable $\mathbf{w}$ in $\mathbb{R}^p$ and a dual variable $\boldsymbol{\kappa}$ in $\mathbb{R}^n$, we obtain using classical Fenchel duality rules [2], that the following quantity is a duality gap for problem (47):

$$\delta(\mathbf{w}, \boldsymbol{\kappa}) \triangleq g(\mathbf{w}) + \tilde{f}^*(\boldsymbol{\kappa}) + \lambda\psi^*(-\mathbf{X}\boldsymbol{\kappa}/\lambda),$$

where $\tilde{f}^*$ and ψ^* are respectively the Fenchel conjugates of $\tilde{f}$ and ψ. Denoting by $\mathbf{w}^\star$ the solution of Eq. (47), the duality gap is interesting in the sense that it upperbounds the difference with the optimal value of the function:

$$\delta(\mathbf{w}, \boldsymbol{\kappa}) \geq g(\mathbf{w}) - g(\mathbf{w}^\star) \geq 0.$$

Similarly, we will consider pairs of primal-dual variables $(\mathbf{W}, \mathbf{K})$ when dealing with matrices.

During the optimization, sequences of primal variables $\mathbf{w}$ are available, and one wishes to exploit duality gaps for estimating the difference $g(\mathbf{w}) - g(\mathbf{w}^\star)$. This requires the following components:

- being able to efficiently compute $\tilde{f}^*$ and ψ^*.
- being able to obtain a "good" dual variable $\boldsymbol{\kappa}$ given a primal variable $\mathbf{w}$, such that $\delta(\mathbf{w}, \boldsymbol{\kappa})$ is close to $g(\mathbf{w}) - g(\mathbf{w}^\star)$.

We suppose that the first point is satisfied (we will detail these computations for every loss and regularization functions in the sequel), and explain how to choose $\boldsymbol{\kappa}$ in general (details will also be given in the sequel).

Let us first consider the choice that associates with a primal variable $\mathbf{w}$, the dual variable

$$\boldsymbol{\kappa}(\mathbf{w}) \triangleq \nabla \tilde{f}(\mathbf{X}^\top \mathbf{w}), \tag{49}$$

and let us compute $\delta(\mathbf{w}, \boldsymbol{\kappa}(\mathbf{w}))$. First, easy computations show that for all vectors $\mathbf{z}$ in $\mathbb{R}^n$, $\tilde{f}^*(\nabla \tilde{f}(\mathbf{z})) = \mathbf{z}^\top \nabla \tilde{f}(\mathbf{z}) - \tilde{f}(\mathbf{z})$, which gives

$$\begin{aligned} \delta(\mathbf{w}, \boldsymbol{\kappa}(\mathbf{w})) &= \tilde{f}(\mathbf{X}^\top \mathbf{w}) + \lambda \psi(\mathbf{w}) + \tilde{f}^*(\nabla \tilde{f}(\mathbf{X}^\top \mathbf{w})) + \lambda \psi^*(-\mathbf{X} \nabla \tilde{f}(\mathbf{X}^\top \mathbf{w})/\lambda), & (50) \\ &= \lambda \psi(\mathbf{w}) + \mathbf{w}^\top \mathbf{X} \nabla \tilde{f}(\mathbf{X}^\top \mathbf{w}) + \lambda \psi^*(-\mathbf{X} \nabla \tilde{f}(\mathbf{X}^\top \mathbf{w})/\lambda). & (51) \end{aligned}$$

We now use the classical Fenchel-Young inequality (see, Proposition 3.3.4 of [2]) on the function ψ, which gives

$$\delta(\mathbf{w}, \boldsymbol{\kappa}(\mathbf{w})) \geq \mathbf{w}^\top \mathbf{X} \nabla \tilde{f}(\mathbf{X}^\top \mathbf{w}) - \mathbf{w}^\top \mathbf{X} \nabla \tilde{f}(\mathbf{X}^\top \mathbf{w}) = 0,$$

with equality if and only if $-\mathbf{X} \nabla \tilde{f}(\mathbf{X}^\top \mathbf{w})$ belongs to $\partial \psi(\mathbf{w})$. Interestingly, we now that first-order optimality conditions for Eq. (48) gives that $-\mathbf{X} \nabla \tilde{f}(\mathbf{X}^\top \mathbf{w}^\star) \in \partial \psi(\mathbf{w}^\star)$. We have now in hand a non-negative function $\mathbf{w} \mapsto \delta(\mathbf{w}, \boldsymbol{\kappa}(\mathbf{w}))$ of $\mathbf{w}$, that upperbounds $g(\mathbf{w}) - g(\mathbf{w}^\star)$ and satisfying $\delta(\mathbf{w}^\star, \boldsymbol{\kappa}(\mathbf{w}^\star)) = 0$.

This is however not a sufficient property to make it a good measure of the quality of the optimization, and further work is required, that will be dependent on $\tilde{f}$ and ψ. We have indeed proven that $\delta(\mathbf{w}^\star, \boldsymbol{\kappa}(\mathbf{w}^\star))$ is always 0. However, for $\mathbf{w}$ different than $\mathbf{w}^\star$, $\delta(\mathbf{w}^\star, \boldsymbol{\kappa}(\mathbf{w}^\star))$ can be infinite, making it a non-informative duality-gap. The reasons for this can be one of the following:

- The term $\psi^*(-\mathbf{X} \nabla \tilde{f}(\mathbf{X}^\top \mathbf{w})/\lambda)$ might have an infinite value.
- Intercepts make the problem more complicated. One can write the formulation with an intercept by adding a row to $\mathbf{X}$ filled with the value 1, add one dimension to the vector $\mathbf{w}$, and consider a regularization function ψ that does regularize the last entry of $\mathbf{w}$. This further complexifies the computation of ψ^* and its definition, as shown in the next section.

Let us now detail how we proceed to solve these problems, but first without considering the intercept. The analysis is similar when working with matrices $\mathbf{W}$ instead of vectors $\mathbf{w}$.

A.0.1 Duality Gaps without Intercepts

Let us show how to compute the Fenchel conjugate of the functions we have introduced. We now present the Fenchel conjugate of the loss functions $\tilde{f}$.

- **The square loss**
$$\begin{aligned} \tilde{f}(\mathbf{z}) &= \tfrac{1}{2n} \|\mathbf{y} - \mathbf{z}\|_2^2, \\ \tilde{f}^*(\boldsymbol{\kappa}) &= \tfrac{n}{2} \|\boldsymbol{\kappa}\|_2^2 + \boldsymbol{\kappa}^\top \mathbf{y}. \end{aligned}$$
- **The logistic loss**
$$\begin{aligned} \tilde{f}(\mathbf{z}) &= \tfrac{1}{n} \textstyle\sum_{i=1}^n \log(1 + e^{-y_i \mathbf{z}_i}) \\ \tilde{f}^*(\boldsymbol{\kappa}) &= \begin{cases} +\infty \text{ if } \exists\ i \in [1; n] \text{ s.t. } y_i \boldsymbol{\kappa}_i \notin [-1, 0], \\ \sum_{i=1}^n (1 + y_i \boldsymbol{\kappa}_i) \log(1 + y_i \boldsymbol{\kappa}_i) - y_i \boldsymbol{\kappa}_i \log(-y_i \boldsymbol{\kappa}_i) \text{ otherwise.} \end{cases} \end{aligned}$$
- **The multiclass logistic loss (or softmax)**. The primal variable is now a matrix $\mathbf{Z}$, in $\mathbb{R}^{n \times r}$, which represents the product $\mathbf{X}^\top \mathbf{W}$. We denote by $\mathbf{K}$ the dual variable in $\mathbb{R}^{n \times r}$.
$$\begin{aligned} \tilde{f}(\mathbf{Z}) &= \tfrac{1}{n} \textstyle\sum_{i=1}^n \log\Big(\sum_{j=1}^r e^{\mathbf{Z}_{ij} - \mathbf{Z}_{i\mathbf{y}_i}} \Big) \\ \tilde{f}^*(\mathbf{K}) &= \begin{cases} +\infty \text{ if } \exists i \in [1; n] \text{ s.t. } \{\mathbf{K}_{ij} < 0 \text{ and } j \neq \mathbf{y}_i\} \text{ or } \mathbf{K}_{i\mathbf{y}_i} < -1, \\ \sum_{i=1}^n \Big[\sum_{j \neq \mathbf{y}_i} \mathbf{K}_{ij} \log(\mathbf{K}_{ij}) + (1 + \mathbf{K}_{i\mathbf{y}_i}) \log(1 + \mathbf{K}_{i\mathbf{y}_i}) \Big]. \end{cases} \end{aligned}$$

Our first remark is that the choice Eq. (49) ensures that $\tilde{f}(\boldsymbol{\kappa})$ is not infinite.

As for the regularization function, except for the Tikhonov regularization which is self-conjugate (it is equal to its Fenchel conjugate), we have considered functions that are norms. There exists therefore a norm $\|.\|$ such that $\psi(\mathbf{w}) = \|\mathbf{w}\|$, and we denote by $\|.\|_*$ its dual-norm. In such a case, the Fenchel conjugate of ψ for a vector $\boldsymbol{\gamma}$ in $\mathbb{R}^p$ takes the form

$$\psi^*(\boldsymbol{\gamma}) = \begin{cases} 0 & \text{if } \|\boldsymbol{\gamma}\|_* \leq 1, \\ +\infty & \text{otherwise.} \end{cases}$$

It turns out that for almost all the norms we have presented, there exists (i) either a closed form for the dual-norm or (ii) there exists an efficient algorithm evaluating it. The only one which does not conform to this statement is the tree-structured sum of ℓ_2-norms, for which we do not know how to evaluate it efficiently.

One can now slightly modify the definition of $\boldsymbol{\kappa}$ to ensure that $\psi^*(-\mathbf{X}\boldsymbol{\kappa}/\lambda) \neq +\infty$. A natural choice is

$$\boldsymbol{\kappa}(\mathbf{w}) \triangleq \min\Big(1, \frac{\lambda}{\|\mathbf{X}\nabla\tilde{f}(\mathbf{X}^\top\mathbf{w})\|_*}\Big)\nabla\tilde{f}(\mathbf{X}^\top\mathbf{w}),$$

which is the one we have implemented. With this new choice, it is easy to see that for all vectors $\mathbf{w}$ in $\mathbb{R}^p$, we still have $\tilde{f}^*(\boldsymbol{\kappa}) \neq +\infty$, and finally, we also have $\delta(\mathbf{w}, \boldsymbol{\kappa}(\mathbf{w})) < +\infty$ and $\delta(\mathbf{w}^\star, \boldsymbol{\kappa}(\mathbf{w}^\star)) = 0$, making it potentially a good duality gap.

A.0.2 Duality Gaps with Intercepts

Even though adding an intercept does seem a simple modification to the original problem, it induces difficulties for finding good dual variables.

We recall that having an intercept is equivalent to having a problem of the type (48), by adding a row to $\mathbf{X}$ filled with the value 1, adding one dimension to the vector $\mathbf{w}$ (or one row for matrices $\mathbf{W}$), and by using a regularization function that does not depend on the last entry of $\mathbf{w}$ (or the last row of $\mathbf{W}$).

Suppose that we are considering a problem of type (48) of dimension $p+1$, but we are using a regularization function $\tilde{\psi} : \mathbb{R}^{p+1} \to \mathbb{R}$, such that for a vector $\mathbf{w}$ in $\mathbb{R}^{p+1}$, $\tilde{\psi}(\mathbf{w}) \triangleq \psi(\mathbf{w}_{[1;p]})$, where $\psi : \mathbb{R}^p \to \mathbb{R}$ is one of the regularization function we have introduced. Then, considering a primal variable $\mathbf{w}$, a dual variable $\boldsymbol{\kappa}$, and writing $\boldsymbol{\gamma} \triangleq -\mathbf{X}\boldsymbol{\kappa}/\lambda$, we are interested in computing

$$\tilde{\psi}^*(\boldsymbol{\gamma}) = \begin{cases} +\infty \text{ if } \boldsymbol{\gamma}_{p+1} \neq 0 \\ \psi^*(\boldsymbol{\gamma}_{[1;p]}) \text{ otherwise,} \end{cases}$$

which means that in order the duality gap not to be infinite, one needs in addition to ensure that $\boldsymbol{\gamma}_{p+1}$ be zero. Since the last row of $\mathbf{X}$ is filled with ones, this writes down $\sum_{i=1}^{p+1} \boldsymbol{\kappa}_i = 0$. For the formulation with matrices $\mathbf{W}$ and $\mathbf{K}$, the constraint is similar but for every column of $\mathbf{K}$.

Let us now detail how we proceed for every loss function to find a "good" dual variable $\boldsymbol{\kappa}$ satisfying this additional constraint, given a primal variable $\mathbf{w}$ in $\mathbb{R}^{p+1}$, we first define the auxiliary function

$$\boldsymbol{\kappa}'(\mathbf{w}) \triangleq \nabla\tilde{f}(\mathbf{X}^\top\mathbf{w}),$$

(which becomes $\mathbf{K}'(\mathbf{W}) \triangleq \nabla\tilde{f}(\mathbf{X}^\top\mathbf{W})$ for matrices), and then define another auxiliary function $\boldsymbol{\kappa}''(\mathbf{w})$ as follows, to take into account the additional constraint $\sum_{i=1}^{p+1} \boldsymbol{\kappa}_i = 0$.

- **For the square loss**, we define another auxiliary function:

$$\boldsymbol{\kappa}''(\mathbf{w}) \triangleq \boldsymbol{\kappa}'(\mathbf{w}) - \frac{1}{n}\mathbf{1}_{p+1}^\top\boldsymbol{\kappa}'(\mathbf{w})\mathbf{1}_{p+1}$$

  where $\mathbf{1}_{p+1}$ is a vector of size $p+1$ filled with ones. This step, ensures that $\sum_{i=1}^{p+1} \boldsymbol{\kappa}''(\mathbf{w})_i = 0$.

- **For the logistic loss**, the situation is slightly more complicated since additional constraints are involved in the definition of $\tilde{f}^*$.

$$\boldsymbol{\kappa}''(\mathbf{w}) \triangleq \operatorname*{arg\,min}_{\boldsymbol{\kappa}\in\mathbb{R}^n} \|\boldsymbol{\kappa} - \boldsymbol{\kappa}'(\mathbf{w})\|_2^2 \;\text{ s.t. }\; \sum_{i=1}^{n} \boldsymbol{\kappa}_i = 0 \text{ and } \forall i \in [1;n],\ \boldsymbol{\kappa}_i \in [-1, 0].$$

  This problem can be solved in linear-time [3] using a similar algorithm as for the projection onto the ℓ_1-ball, since it is an instance of a *quadratic knapsack problem.*

- **For the multi-class logistic loss**, we proceed in a similar way, for every column $\mathbf{K}^j$ of $\mathbf{K}$, $j \in [1; r]$:

$$\mathbf{K}''^j(\mathbf{w}) \triangleq \arg\min_{\boldsymbol{\kappa} \in \mathbb{R}^n} \|\mathbf{K}'^j - \boldsymbol{\kappa}'(\mathbf{w})\|_2^2 \text{ s.t. } \sum_{i=1}^n \boldsymbol{\kappa}_i = 0 \text{ and } \forall i \in [1; n],\ \{\boldsymbol{\kappa}_i \geq 0 \text{ if } j \neq \mathbf{y}_i\} \text{ and } \{\boldsymbol{\kappa}_i \geq -1 \text{ if } \mathbf{y}_i = j\}.$$

When the function ψ is the Tykhonov regularization function, we end the process by setting $\boldsymbol{\kappa}(\mathbf{w}) = \boldsymbol{\kappa}''(\mathbf{w})$. When it is a norm, we choose, as before for taking into account the constraint $\|\mathbf{X}\boldsymbol{\kappa}\|_* \leq \lambda$,

$$\boldsymbol{\kappa}(\mathbf{w}) \triangleq \min\Big(1, \frac{\lambda}{\|\mathbf{X}\boldsymbol{\kappa}''(\mathbf{w})\|_*}\Big)\boldsymbol{\kappa}''(\mathbf{w}),$$

with a similar formulation for matrices $\mathbf{W}$ and $\mathbf{K}$.

Even though finding dual variables while taking into account the intercept requires quite a lot of engineering, notably implementing a quadratic knapsack solver, it can be done efficiently.

References

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

[2] J. M. Borwein and A. S. Lewis. *Convex analysis and nonlinear optimization: Theory and examples*. Springer, 2006.

[3] P. Brucker. An O(n) algorithm for quadratic knapsack problems. 3:163–166, 1984.

[4] E. J. Candès, M. Wakin, and S. Boyd. Enhancing sparsity by reweighted l1 minimization. *Journal of Fourier Analysis and Applications*, 14:877–905, 2008.

[5] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[6] S. F. Cotter, J. Adler, B. Rao, and K. Kreutz-Delgado. Forward sequential algorithms for best basis selection. In *IEEE Proceedings of Vision Image and Signal Processing*, pages 235–244, 1999.

[7] J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra. Efficient projections onto the ℓ_1-ball for learning in high dimensions. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2008.

[8] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *Annals of statistics*, 32(2):407–499, 2004.

[9] J. Friedman, T. Hastie, H. Hölfling, and R. Tibshirani. Pathwise coordinate optimization. *Annals of statistics*, 1(2):302–332, 2007.

[10] J. Friedman, T. Hastie, and R. Tibshirani. A note on the group lasso and a sparse group lasso. Technical report, Preprint arXiv:1001.0736, 2010.

[11] W. J. Fu. Penalized regressions: The bridge versus the Lasso. *Journal of computational and graphical statistics*, 7:397–416, 1998.

[12] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proc. of ACM Symposium on Theory of Computing*, pages 136–146, 1986.

[13] P. O. Hoyer. Non-negative sparse coding. In *Proc. IEEE Workshop on Neural Networks for Signal Processing*, 2002.

[14] R. Jenatton, J. Mairal, G. Obozinski, and F. Bach. Proximal methods for sparse hierarchical dictionary learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2010.

[15] R. Jenatton, J. Mairal, G. Obozinski, and F. Bach. Proximal methods for hierarchical sparse coding. *Journal of Machine Learning Research*, 12:2297–2334, 2011.

[16] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *Advances in Neural Information Processing Systems*, 2001.

[17] N. Maculan and J. R. G. Galdino de Paula. A linear-time median-finding algorithm for projecting a vector on the simplex of Rn. *Operations research letters*, 8(4):219–222, 1989.

[18] J. Mairal. *Sparse coding for machine learning, image processing and computer vision*. PhD thesis, Ecole Normale Supérieure, Cachan, 2010.

[19] J. Mairal, F. Bach, J. Ponce, and G. Sapiro. Online dictionary learning for sparse coding. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2009.

[20] J. Mairal, F. Bach, J. Ponce, and G. Sapiro. Online learning for matrix factorization and sparse coding. *Journal of Machine Learning Research*, 11:19–60, 2010.

[21] J. Mairal, R. Jenatton, G. Obozinski, and F. Bach. Network flow algorithms for structured sparsity. In *Advances in Neural Information Processing Systems*, 2010.

[22] J. Mairal, R. Jenatton, G. Obozinski, and F. Bach. Convex and network flow optimization for structured sparsity. *Journal of Machine Learning Research*, 12:2649–2689, 2011.

[23] J. Mairal and B. Yu. Supervised feature selection in graphs with path coding penalties and network flows. Technical report, Preprint arXiv:1204.4539, 2012.

[24] S. Mallat and Z. Zhang. Matching pursuit in a time-frequency dictionary. *IEEE Transactions on Signal Processing*, 41(12):3397–3415, 1993.

[25] N. Meinshausen and P. Buehlmann. Stability selection. Technical report. ArXiv:0809.2932.

[26] G. Obozinski, B. Taskar, and M.I. Jordan. Joint covariate selection and joint subspace selection for multiple classification problems. *Statistics and Computing*, pages 1–22.

[27] M. R. Osborne, B. Presnell, and B. A. Turlach. On the Lasso and its dual. *Journal of Computational and Graphical Statistics*, 9(2):319–37, 2000.

[28] P. Sprechmann, I. Ramirez, G. Sapiro, and Y. C. Eldar. Collaborative hierarchical sparse modeling. Technical report, 2010. Preprint arXiv:1003.0400v1.

[29] R. Tibshirani, M. Saunders, S. Rosset, J. Zhu, and K. Knight. Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistical Society Series B*, 67(1):91–108, 2005.

[30] J. A. Tropp. Algorithms for simultaneous sparse approximation. part ii: Convex relaxation. *Signal Processing, special issue "Sparse approximations in signal and image processing"*, 86:589–602, April 2006.

[31] J. A. Tropp, A. C. Gilbert, and M. J. Strauss. Algorithms for simultaneous sparse approximation. part i: Greedy pursuit. *Signal Processing, special issue "sparse approximations in signal and image processing"*, 86:572–588, April 2006.

[32] S. Weisberg. *Applied Linear Regression*. Wiley, New York, 1980.

[33] T. T. Wu and K. Lange. Coordinate descent algorithms for Lasso penalized regression. *Annals of Applied Statistics*, 2(1):224–244, 2008.

[34] M. Yuan and Y. Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society Series B*, 68:49–67, 2006.

[35] H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society Series B*, 67(2):301–320, 2005.