SPORCO: A Python package for standard and convolutional sparse representations

Brendt Wohlberg[‡*]

Abstract—SParse Optimization Research COde (SPORCO) is an open-source Python package for solving optimization problems with sparsity-inducing regularization, consisting primarily of sparse coding and dictionary learning, for both standard and convolutional forms of sparse representation. In the current version, all optimization problems are solved within the Alternating Direction Method of Multipliers (ADMM) framework. SPORCO was developed for applications in signal and image processing, but is also expected to be useful for problems in computer vision, statistics, and machine learning.

Index Terms—sparse representations, convolutional sparse representations, sparse coding, convolutional sparse coding, dictionary learning, convolutional dictionary learning, alternating direction method of multipliers

Introduction

SPORCO is an open-source Python package for solving inverse problems with sparsity-inducing regularization [MBP14]. This type of regularization has become one of the leading techniques in signal and image processing, with applications including image denoising, inpainting, deconvolution, superresolution, and compressed sensing, to name only a few. It is also a prominent method in machine learning and computer vision, with applications including image classification, video background modeling, collaborative filtering, and genomic data analysis, and is widely used in statistics as a regression technique.

SPORCO was initially a Matlab library, but the implementation language was switched to Python for a number of reasons, including (i) the substantial cost of Matlab licenses (particularly in an environment that does not qualify for an academic discount), and the difficulty of running large scale experiments on multiple hosts with a limited supply of toolbox licenses, (ii) the greater maintainability and flexibility of the object-oriented design possible in Python, (iii) the flexibility provided by NumPy in indexing arrays of arbitrary numbers of dimensions (essentially impossible in Matlab), and (iv) the superiority of Python as a general-purpose programming language.

SPORCO supports a variety of inverse problems, including Total Variation [ROF92] [All92] denoising and deconvolution, and Robust PCA [CCS10], but the primary focus is on sparse coding and dictionary learning, for solving problems with sparse representations [MBP14]. Both standard and convolutional forms of sparse representations are supported. In the standard form the dictionary is a matrix, which limits the sizes of signals, images, etc. that can be directly represented; the usual strategy is to compute independent representations for a set of overlapping blocks, as illustrated in Figure 1. In the convolutional form [LS99][ZKTF10][Woh16d] the dictionary is a set of linear filters, making it feasible to directly represent an entire signal or image. The convolutional form is equivalent to sparse coding with a structured dictionary constructed from translations of a smaller generating dictionary, as illustrated in Figure 2. The support for the convolutional form is one of the major strengths of SPORCO since it is the only Python package to provide such a breadth of options for convolutional sparse coding and dictionary learning. Some features are not available in any other open-source package, including support for representation of multi-channel images (e.g. RGB color images) [Woh16b], and representation of arrays of

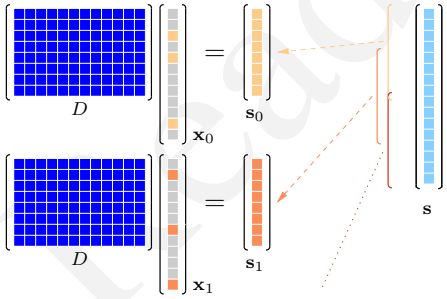

Fig. 1: *Independent sparse coding of overlapping blocks*

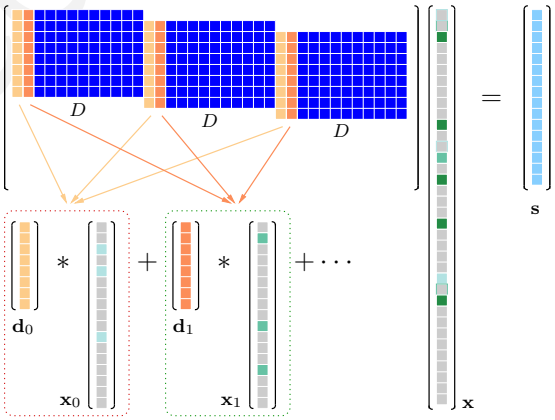

Fig. 2: *Convolutional sparse coding of an entire signal*

∗ *Corresponding author: brendt@ieee.org*
‡ *Theoretical Division, Los Alamos National Laboratory*

arbitrary numbers of dimensions, allowing application to one-dimensional signals, images, and video and volumetric data.

In the current version, all optimization problems are solved within the Alternating Direction Method of Multipliers (ADMM) [BPC+10] framework, which is implemented as flexible class hierarchy designed to minimize the additional code that has to be written to solve a specific problem. This design also simplifies the process of deriving algorithms for solving variants of existing problems, in some cases only requiring overriding one or two methods, involving a few additional lines of code.

The remainder of this paper provides a more detailed overview of the SPORCO library. A brief introduction to the ADMM optimization approach is followed by a discussion of the design of the classes that implement it. This is followed by a discussion of both standard and convolutional forms of sparse coding and dictionary learning, and some comments on the selection of parameters for the inverse problems supported by SPORCO. The next section addresses the installation of SPORCO, and is followed by some usage examples. The remaining sections consist of a discussion of the derivation of extensions of supported problems, a list of useful support modules in SPORCO, and closing remarks.

ADMM

The ADMM [BPC+10] framework addresses optimization problems of the form

$$\operatorname{argmin}_{\mathbf{x},\mathbf{y}} \; f(\mathbf{x}) + g(\mathbf{y}) \;\text{ such that }\; A\mathbf{x} + B\mathbf{y} = \mathbf{c} \;. \tag{1}$$

This general problem is solved by iterating over the following three update steps:

$$\begin{aligned}
\mathbf{x}^{(j+1)} &= \operatorname{argmin}_{\mathbf{x}} \; f(\mathbf{x}) + \frac{\rho}{2} \left\| A\mathbf{x} - \left(-B\mathbf{y}^{(j)} + \mathbf{c} - \mathbf{u}^{(j)} \right) \right\|_2^2 \\
\mathbf{y}^{(j+1)} &= \operatorname{argmin}_{\mathbf{y}} \; g(\mathbf{y}) + \frac{\rho}{2} \left\| B\mathbf{y} - \left(-A\mathbf{x}^{(j+1)} + \mathbf{c} - \mathbf{u}^{(j)} \right) \right\|_2^2 \\
\mathbf{u}^{(j+1)} &= \mathbf{u}^{(j)} + A\mathbf{x}^{(j+1)} + B\mathbf{y}^{(j+1)} - \mathbf{c}
\end{aligned}$$

which we will refer to as the **x**, **y**, and **u**, steps respectively.

The feasibility conditions (see Sec. 3.3 [BPC+10]) for the ADMM problem are

$$\begin{aligned}
& A\mathbf{x}^* + B\mathbf{y}^* - \mathbf{c} = 0 \\
& 0 \in \partial f(\mathbf{x}^*) + \rho^{-1} A^T \mathbf{u}^* \\
& 0 \in \partial g(\mathbf{u}^*) + \rho^{-1} B^T \mathbf{u}^* \;,
\end{aligned}$$

where ∂ denotes the subdifferential operator. It can be shown that the last feasibility condition above is always satisfied by the solution of the **y** step. The primal and dual residuals [BPC+10]

$$\begin{aligned}
\mathbf{r} &= A\mathbf{x}^{(j+1)} + B\mathbf{y}^{(j+1)} - \mathbf{c} \\
\mathbf{s} &= \rho A^T B(\mathbf{y}^{(j+1)} - \mathbf{y}^{(j)}) \;,
\end{aligned}$$

which can be derived from the feasibility conditions, provide a convenient measure of convergence, and can be used to define algorithm stopping criteria. The **u** step can be written in terms of the primal residual as

$$\mathbf{u}^{(j+1)} = \mathbf{u}^{(j)} + \mathbf{r}^{(j+1)} \;.$$

It is often preferable to use normalized versions of these residuals [Woh17], obtained by dividing the definitions above by their corresponding normalization factors

$$\begin{aligned}
r_{\mathrm{n}} &= \max(\|A\mathbf{x}^{(j+1)}\|_2, \|B\mathbf{y}^{(j+1)}\|_2, \|\mathbf{c}\|_2) \\
s_{\mathrm{n}} &= \rho \|A^T \mathbf{u}^{(j+1)}\|_2 \;.
\end{aligned}$$

These residuals can also be used in a heuristic scheme [Woh17] for selecting the critical *penalty parameter* ρ.

SPORCO ADMM Classes

SPORCO provides a flexible set of classes for solving problems within the ADMM framework. All ADMM algorithms are derived from class `admm.admm.ADMM`, which provides much of the infrastructure required for solving a problem, so that the user need only override methods that define the constraint components A, B, and $\mathbf{c}$, and that compute the **x** and **y** steps. This infrastructure includes the computation of the primal and dual residuals, which are used as convergence measures on which termination of the iterations can be based.

These residuals are also used within the heuristic scheme, referred to above for, automatically setting the penalty parameter. This scheme is controlled by the `AutoRho` entry in the algorithm options dictionary object that is used to specify algorithm options and parameters. For example, to enable or disable it, set `opt['AutoRho', 'Enabled']` to `True` or `False` respectively, where `opt` is an instance of `admm.admm.ADMM.Options` or one of its derived classes. It should be emphasized that this method is not always successful, and can result in oscillations or divergence of the optimization. The scheme is enabled by default for classes for which it is expected to give reasonable performance, and disabled for those for which it is not, but these default settings should not be considered to be particularly reliable, and the user is advised to explicitly select whether the method is enabled to disabled.

Additional class attributes and methods can be defined to customize the calculation of diagnostic information, such as the functional value, at each iteration. The SPORCO documentation includes a detailed description of the required and optional methods to be overridden in defining a class for solving a specific optimization problem.

The `admm.admm` module also includes classes that are derived from `admm.admm.ADMM` to specialize to less general cases; for example, class `admm.admm.ADMMEqual` assumes that $A = I$, $B = -I$, and $\mathbf{c} = \mathbf{0}$, which is a very frequently occurring case, allowing derived classes to avoid overriding methods that specify the constraint. The most complex partial specialization is `admm.admm.ADMMTwoBlockCnstrnt`, which implements the commonly-occurring ADMM problem form with a block-structured **y** variable,

$$\begin{aligned}
&\operatorname{argmin}_{\mathbf{x},\mathbf{y}_0,\mathbf{y}_1} \; f(\mathbf{x}) + g_0(\mathbf{y}_0) + g_0(\mathbf{y}_1) \\
&\text{such that } \begin{pmatrix} A_0 \\ A_1 \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_0 \\ \mathbf{c}_1 \end{pmatrix} \;,
\end{aligned}$$

for solving problems that have the form

$$\operatorname{argmin}_{\mathbf{x}} f(\mathbf{x}) + g_0(A_0\mathbf{x}) + g_1(A_1\mathbf{x})$$

prior to variable splitting. The block components of the **y** variable are concatenated into a single NumPy array, with access to the individual components provided by methods `block_sep0` and `block_sep1`.

Defining new classes derived from `admm.admm.ADMM` or one of its partial specializations provides complete flexibility in constructing a new ADMM algorithm, while reducing the amount of code that has to be written compared with implementing the entire ADMM algorithm from scratch. When a new ADMM algorithm is closely related to an existing algorithm, it is often

much easier to derived the new class from that of the existing algorithm, as described in the section *Extending SPORCO*.

Sparse Coding

Sparse coding in SPORCO is based on the Basis Pursuit DeNoising (BPDN) problem [CDS98]

$$\text{argmin}_X \ (1/2)\|DX - S\|_F^2 + \lambda \|X\|_1 \ ,$$

were D is the dictionary, S is the signal matrix, each column of which is a distinct signal, X is the sparse representation, and λ is the regularization parameter controlling the sparsity of the solution. BPDN is solved via the equivalent ADMM problem

$$\text{argmin}_X \ (1/2)\|DX - S\|_F^2 + \lambda \|Y\|_1 \quad \text{such that} \quad X = Y \ .$$

This algorithm is effective because the Y step can be solved in closed form, and is computationally relatively cheap. The main computational cost is in solving the X step, which involves solving the potentially-large linear system

$$(D^T D + \rho I)X = D^T S + \rho(Y - U) \ .$$

SPORCO solves this system efficiently by precomputing an LU factorization of $(D^T D + \rho I)$ which enables a rapid direct-method solution at every iteration (see Sec. 4.2.3 in [BPC+10]). In addition, if $(DD^T + \rho I)$ is smaller than $(D^T D + \rho I)$, the matrix inversion lemma is used to reduce the size of the system that is actually solved (see Sec. 4.2.4 in [BPC+10]).

The solution of the BPDN problem is implemented by class `admm.bpdn.BPDN`. A number of variations on this problem are supported by other classes in module `admm.bpdn`.

Dictionary Learning

Dictionary learning is based on the problem

$$\text{argmin}_{D,X} \ (1/2)\|DX - S\|_F^2 + \lambda \|X\|_1 \ \text{ s.t } \ \|\mathbf{d}_m\|_2 = 1 \ ,$$

which is solved by alternating between a sparse coding stage, as above, and a constrained dictionary update obtained by solving the problem

$$\text{argmin}_D (1/2)\|DX - S\|_2^2 \ \text{ s.t } \ \|\mathbf{d}_m\|_2 = 1 \ .$$

This approach is implemented by class `admm.bpdndl.DictLearn`. An unusual feature of this dictionary learning algorithm is the adoption from convolutional dictionary learning [BEL13] [Woh16d] [GCW17] of the very effective strategy of alternating between a single step of each of the sparse coding and dictionary update algorithms. To the best of this author's knowledge, this strategy has not previously been applied to standard (non-convolutional) dictionary learning.

Convolutional Sparse Coding

Convolutional sparse coding (CSC) is based on a convolutional form of BPDN, referred to as Convolutional BPDN (CBPDN) [Woh16d]

$$\text{argmin}_{\{\mathbf{x}_m\}} \ \frac{1}{2} \left\| \sum_m \mathbf{d}_m * \mathbf{x}_m - \mathbf{s} \right\|_2^2 + \lambda \sum_m \|\mathbf{x}_m\|_1 \ ,$$

which is implemented by class `admm.cbpdn.ConvBPDN`. Module `admm.cbpdn` also contains a number of other classes implementing variations on this basic form. As in the case of standard BPDN, the main computational cost of this algorithm is in solving the $\mathbf{x}$ step, which can be solved very efficiently by exploiting the Sherman-Morrison formula [Woh14]. SPORCO provides support for solving the basic form above, as well as a number of variants, including one with a gradient penalty, and two different approaches for solving a variant with a spatial mask W [HHW15][Woh16a]

$$\text{argmin}_{\{\mathbf{x}_m\}} \ \frac{1}{2} \left\| W \left(\sum_m \mathbf{d}_m * \mathbf{x}_m - \mathbf{s} \right) \right\|_2^2 + \lambda \sum_m \|\mathbf{x}_m\|_1 \ .$$

SPORCO also supports two different methods for convolutional sparse coding of multi-channel (e.g. color) images [Woh16b]. The one represents a multi-channel input with channels $\mathbf{s}_c$ with single-channel dictionary filters $\mathbf{d}_m$ and multi-channel coefficient maps $\mathbf{x}_{c,m}$,

$$\text{argmin}_{\{\mathbf{x}_{c,m}\}} \ \frac{1}{2} \sum_c \left\| \sum_m \mathbf{d}_m * \mathbf{x}_{c,m} - \mathbf{s}_c \right\|_2^2 + \lambda \sum_c \sum_m \|\mathbf{x}_{c,m}\|_1 \ ,$$

and the other uses multi-channel dictionary filters $\mathbf{d}_{c,m}$ and single-channel coefficient maps $\mathbf{x}_m$,

$$\text{argmin}_{\{\mathbf{x}_m\}} \ \frac{1}{2} \sum_c \left\| \sum_m \mathbf{d}_{c,m} * \mathbf{x}_m - \mathbf{s}_c \right\|_2^2 + \lambda \sum_m \|\mathbf{x}_m\|_1 \ .$$

In the former case the representation of each channel is completely independent unless they are coupled via an $\ell_{2,1}$ norm term [Woh16b], which is supported by class `admm.cbpdn.ConvBPDNJoint`.

An important issue that has received surprisingly little attention in the literature is the need to explicitly consider the representation of the smooth/low frequency image component when constructing convolutional sparse representations. If this component is not properly taken into account, convolutional sparse representations tend to give poor results. As briefly mentioned in [Woh16d] (Sec. I), the simplest approach is to lowpass filter the image to be represented, computing the sparse representation on the highpass residual. In this approach the lowpass component forms part of the complete image representation, and should, of course, be added to the reconstruction from the sparse representation in order to reconstruct the image being represented. SPORCO supports this separation of an image into lowpass/highpass components via the function `util.tikhonov_filter`, which computes the lowpass component of $\mathbf{s}$ as the solution of the problem

$$\text{argmin}_{\mathbf{x}} \ (1/2) \|\mathbf{x} - \mathbf{s}\|_2^2 + (\lambda/2) \sum_i \|G_i \mathbf{x}\|_2^2 \ ,$$

where G_i is an operator computing the derivative along axis i of the array represented as vector $\mathbf{x}$, and λ is a parameter controlling the amount of smoothing. In some cases it is not feasible to handle the lowpass component via such a pre-processing strategy, making it necessary to include the lowpass component in the CSC optimization problem itself. The simplest approach to doing so is to append an impulse filter to the dictionary and include a gradient regularization term on corresponding coefficient map in the functional [Woh16c] (Sec. 3). This approach is supported by class `admm.cbpdn.ConvBPDNGradReg`, the use of which is demonstrated in section *Removal of Impulse Noise via CSC*.

Convolutional Dictionary Learning

Convolutional dictionary learning is based on the problem

$$\text{argmin}_{\{\mathbf{d}_m\},\{\mathbf{x}_{k,m}\}} \ \frac{1}{2} \sum_k \left\| \sum_m \mathbf{d}_m * \mathbf{x}_{k,m} - \mathbf{s}_k \right\|_2^2 + \lambda \sum_k \sum_m \|\mathbf{x}_{k,m}\|_1 \ \text{ s.t } \ \mathbf{d}_m \in C \ ,$$

where C is the feasible set, consisting of filters with unit norm and constrained support [Woh16d]. It is solved by alternating between a convolutional sparse coding stage, as described in the previous section, and a constrained dictionary update obtained by solving the problem

$$\operatorname{argmin}_{\{\mathbf{d}_m\}} \frac{1}{2} \sum_k \left\| \sum_m \mathbf{d}_m * \mathbf{x}_{k,m} - \mathbf{s}_k \right\|_2^2 \text{ s.t. } \mathbf{d}_m \in C \ .$$

This approach is implemented by class `ConvBPDNDictLearn` in module `admm.cbpdndl`. Dictionary learning with a spatial mask W,

$$\operatorname{argmin}_{\{\mathbf{d}_m\},\{\mathbf{x}_{k,m}\}} \frac{1}{2} \sum_k \left\| W \left(\sum_m \mathbf{d}_m * \mathbf{x}_{k,m} - \mathbf{s}_k \right) \right\|_2^2 + \lambda \sum_k \sum_m \|\mathbf{x}_{k,m}\|_1 \quad \text{s.t } \mathbf{d}_m \in C$$

is also supported by class `ConvBPDNMaskDcplDictLearn` in module `admm.cbpdndl`.

Convolutional Representations

SPORCO convolutional representations are stored within NumPy arrays of `dimN` + 3 dimensions, where `dimN` is the number of spatial/temporal dimensions in the data to be represented. This value defaults to 2 (i.e. images), but can be set to any other reasonable value, such as 1 (i.e. one-dimensional signals) or 3 (video or volumetric data). The roles of the axes in these multi-dimensional arrays are required to follow a fixed order: first spatial/temporal axes, then an axis for multiple channels (singleton in the case of single-channel data), then an axis for multiple input signals (singleton in the case of only one input signal), and finally the axis corresponding to the index of the filters in the dictionary.

Sparse Coding

For the convenience of the user, the `D` (dictionary) and `S` (signal) arrays provided to the convolutional sparse coding classes need not follow this strict format, but they are internally reshaped to this format for computational efficiency. This internal reshaping is largely transparent to the user, but must be taken into account when passing weighting arrays to optimization classes (e.g. option `L1Weight` for class `admm.cbpdn.ConvBPDN`). When performing the reshaping into internal array layout, it is necessary to infer the intended roles of the axes of the input arrays, which is performed by class `admm.cbpdn.ConvRepIndexing` (this class is expected to be moved to a different module in a future version of SPORCO). The inference rules, which are described in detail in the documentation for class `admm.cbpdn.ConvRepIndexing`, are relatively complex, depending on both the number of dimensions in the `D` and `S` arrays, and on parameters `dimK` and `dimN`.

Dictionary Update

The handling of convolutional representations by the dictionary update classes in module `admm.ccmod` are similar to those for sparse coding, the primary difference being the the dictionary update classes expect that the sparse representation inputs `X` are already in the standard layout as described above since they are usually obtained as the output of one of the sparse coding classes, and therefore already have the required layout. The inference of internal dimensions for these classes is handled by class `admm.ccmod.ConvRepIndexing` (which is also expected to be moved to a different module in a future version of SPORCO).

Problem Parameters

Most of the inverse problems supported by SPORCO have at least one problem parameter (e.g. regularization parameter λ in the BPDN and CBPDN problems) that determines the balance between the different terms in the functional to be minimized. Of these, the only problem that has a relatively reliable default value for its parameter is RPCA (see class `admm.rpca.RobustPCA`). Most of the classes implementing BPDN and CBPDN problems do have default values for regularization parameter λ, but these defaults should not be expected to provide even close to optimal performance for specific applications, and may be removed in future versions.

SPORCO does not support any statistical parameter estimation techniques such as GCV [GHW79] or SURE [Ste81], but the grid search function `util.grid_search` can be very helpful in selecting problem parameters when a suitable data set with ground truth is available. This function efficiently evaluates a user-specified performance measure, in parallel, over a single- or multi-dimensional grid sampling the parameter space. Usage of this function is illustrated in the example scripts `examples/stdsparse/demo_bpdn.py` and `examples/stdsparse/demo_bpdnjnt.py`, which "cheat" by evaluating performance by using the ground truth for the actual problem being solved. In a more realistic setting, one would optimize the parameters using the ground truth for a separate set of data with the same properties as those of the data for the test problem.

Installing SPORCO

The primary requirements for SPORCO are Python itself (version 2.7 or 3.x), and modules numpy, scipy, future, pyfftw, and matplotlib. Module numexpr is not required, but some functions will be faster if it is installed. If module mpldatacursor is installed, `plot.plot` and `plot.imview` will support the data cursor that it provides. Additional information on the requirements are provided in the installation instructions.

SPORCO is available on GitHub and can be installed via `pip`:

```
pip install sporco
```

SPORCO can also be installed from source, either from the development version from GitHub, or from a release source package downloaded from PyPI.

To install the development version from GitHub do

```
git clone https://github.com/bwohlberg/sporco.git
```

followed by

```
cd sporco
python setup.py build
python setup.py test
python setup.py install
```

The install command will usually have to be performed with root permissions, e.g. on Ubuntu Linux

```
sudo python setup.py install
```

The procedure for installing from a source package downloaded from PyPI is similar.

A summary of the most significant changes between SPORCO releases can be found in the `CHANGES.rst` file. It is strongly recommended to consult this summary when updating from a previous version.

SPORCO includes a large number of usage examples, some of which make use of a set of standard test images, which can be installed using the `sporco_get_images` script. To download these images from the root directory of the source distribution (i.e. prior to installation) do

```
bin/sporco_get_images --libdest
```

after setting the `PYTHONPATH` environment variable to point to the root directory of the source distribution; for example, in a `bash` shell

```
export PYTHONPATH=$PYTHONPATH:`pwd`
```

from the root directory of the package. To download the images as part of a package that has already been installed, do

```
sporco_get_images --libdest
```

which will usually have to be performed with root privileges.

Using SPORCO

The simplest way to use SPORCO is to make use of one of the many existing classes for solving problems that are already supported, but SPORCO is also designed to be easy to extend to solve custom problems, in some cases requiring only a few lines of additional code to extend an existing class to solve a new problem. This latter, more advanced usage is described in the section *Extending SPORCO*.

Detailed documentation is available. The distribution includes a large number of example scripts and a selection of Jupyter notebook demos, which can be viewed online via nbviewer, or run interactively via mybinder.

A Simple Usage Example

Each optimization algorithm is implemented as a separate class. Solving a problem is straightforward, as illustrated in the following example, which assumes that we wish to solve the BPDN problem

$$\operatorname{argmin}_{\mathbf{x}} (1/2)\|D\mathbf{x} - \mathbf{s}\|_F^2 + \lambda\|\mathbf{x}\|_1$$

for a given dictionary D and signal vector $\mathbf{s}$, represented by NumPy arrays `D` and `s` respectively. After importing the appropriate modules

```
import numpy as np
from sporco.admm import bpdn
```

we construct a synthetic problem consisting of a random dictionary and a test signal that is generated so that it has a very sparse representation, `x0`, on that dictionary

```
np.random.seed(12345)
D = np.random.randn(8, 16)
x0 = np.zeros((16, 1))
x0[[3,11]] = np.random.randn(2,1)
s = D.dot(x0)
```

Now we create an object representing the desired algorithm options

```
opt = bpdn.BPDN.Options({'Verbose' : True,
                         'MaxMainIter' : 500,
                         'RelStopTol' : 1e-6})
```

initialize the solver object

```
lmbda = 1e-2
b = bpdn.BPDN(D, s, lmbda, opt)
```

and call the `solve` method

```
x = b.solve()
```

leaving the result in NumPy array `x`. Since the optimizer objects retain algorithm state, calling `solve` again gives a warm start on an additional set of iterations for solving the same problem (e.g. if the first solve terminated because it reached the maximum number of iterations, but the desired solution accuracy was not reached).

Removal of Impulse Noise via CSC

We now consider a more detailed and realistic usage example, based on using CSC to remove impulse noise from a color image. First we need to import some modules, including `print_function` for Python 2/3 compatibility, NumPy, and a number of modules from SPORCO:

```
from __future__ import print_function

import numpy as np
from scipy.misc import imsave

from sporco import util
from sporco import plot
from sporco import metric
from sporco.admm import cbpdn
```

Boundary artifacts are handled by performing a symmetric extension on the image to be denoised and then cropping the result to the original image support. This approach is simpler than the boundary handling strategies described in [HHW15] and [Woh16a], and for many problems gives results of comparable quality. The functions defined here implement symmetric extension and cropping of images.

```
def pad(x, n=8):

  if x.ndim == 2:
      return np.pad(x, n, mode='symmetric')
  else:
      return np.pad(x, ((n, n), (n, n), (0, 0)),
                    mode='symmetric')


def crop(x, n=8):

  return x[n:-n, n:-n]
```

Now we load a reference image (see the discussion on the script for downloading standard test images in section *Installing SPORCO*), and corrupt it with 33% salt and pepper noise. (The call to `np.random.seed` ensures that the pseudo-random noise is reproducible.)

```
img = util.ExampleImages().image('standard',
      'monarch.png', zoom=0.5, scaled=True,
      idxexp=np.s_[:, 160:672])
np.random.seed(12345)
imgn = util.spnoise(img, 0.33)
```

We use a color dictionary, as described in [Woh16b]. The impulse denoising problem is solved by appending some additional filters to the learned dictionary `D0`, which is one of those distributed with SPORCO. The first of these additional components is a set of three impulse filters, one per color channel, that will represent the impulse noise, and the second is an identical set of impulse filters that will represent the low frequency image components when used together with a gradient penalty on the coefficient maps, as discussed below.

```
D0 = util.convdicts()['RGB:8x8x3x64']
Di = np.zeros(D0.shape[0:2] + (3, 3))
np.fill_diagonal(Di[0, 0], 1.0)
D = np.concatenate((Di, Di, D0), axis=3)
```

The problem is solved using class `ConvBPDNGradReg` in module `admm.cbpdn`, which implements the form of CBPDN with an additional gradient regularization term,

$$\mathrm{argmin}_{\{\mathbf{x}_m\}} \frac{1}{2} \left\| \sum_m \mathbf{d}_m * \mathbf{x}_m - \mathbf{s} \right\|_2^2 + \lambda \sum_m \|\mathbf{x}_m\|_1 + \frac{\mu}{2} \sum_i \sum_m \|G_i \mathbf{x}_m\|_2^2$$

where G_i is an operator computing the derivative along index i, as described in [Woh16c]. The regularization parameters for the ℓ_1 and gradient terms are `lmbda` and `mu` respectively. Setting correct weighting arrays for these regularization terms is critical to obtaining good performance. For the ℓ_1 norm, the weights on the filters that are intended to represent the impulse noise are tuned to an appropriate value for the impulse noise density (this value sets the relative cost of representing an image feature by one of the impulses or by one of the filters in the learned dictionary), the weights on the filters that are intended to represent low frequency components are set to zero (we only want them penalized by the gradient term), and the weights of the remaining filters are set to unity. For the gradient penalty, all weights are set to zero except for those corresponding to the filters intended to represent low frequency components, which are set to unity.

```
lmbda = 2.8e-2
mu = 3e-1
w1 = np.ones((1, 1, 1, 1, D.shape[-1]))
w1[..., 0:3] = 0.33
w1[..., 3:6] = 0.0
wg = np.zeros((D.shape[-1]))
wg[..., 3:6] = 1.0
opt = cbpdn.ConvBPDNGradReg.Options(
       {'Verbose': True, 'MaxMainIter': 100,
        'RelStopTol': 5e-3, 'AuxVarObj': False,
        'L1Weight': w1, 'GradWeight': wg})
```

Now we initialize the `cbpdn.ConvBPDNGradReg` object and call the `solve` method.

```
b = cbpdn.ConvBPDNGradReg(D, pad(imgn), lmbda, mu,
                          opt=opt, dimK=0)
X = b.solve()
```

The denoised estimate of the image is just the reconstruction from all coefficient maps except those that represent the impulse noise, which is why we subtract the slice of `X` corresponding the impulse noise representing filters from the result of `reconstruct`.

```
imgdp = b.reconstruct().squeeze() \
        - X[..., 0, 0:3].squeeze()
imgd = crop(imgdp)
```

Now we print the PSNR of the noisy and denoised images, and display the reference, noisy, and denoised images. These images are shown in Figures 3, 4, and 5 respectively.

```
print('%.3f dB   %.3f dB' % (sm.psnr(img, imgn),
      sm.psnr(img, imgd)))

fig = plot.figure(figsize=(21, 7))
plot.subplot(1,3,1)
plot.imview(img, fgrf=fig, title='Reference')
plot.subplot(1,3,2)
plot.imview(imgn, fgrf=fig, title='Noisy')
plot.subplot(1,3,3)
plot.imview(imgd, fgrf=fig, title='CSC Result')
fig.show()
```

Fig. 3: *Reference image*

Fig. 4: *Noisy image*

Finally, we save the low frequency image component estimate as an NPZ file, for use in a subsequent example.

```
imglp = X[..., 0, 3:6].squeeze()
np.savez('implslpc.npz', imglp=imglp)
```

Extending SPORCO

We illustrate the ease of extending or modifying existing algorithms in SPORCO by constructing an alternative approach to removing impulse noise via CSC. The previous method gave good results, but the weight on the filter representing the impulse noise is an additional parameter that has to be tuned. This parameter can be avoided by switching to an ℓ_1 data fidelity term instead of including dictionary filters to represent the impulse noise, as in the problem [Woh16c]

$$\mathrm{argmin}_{\{\mathbf{x}_m\}} \left\| \sum_m \mathbf{d}_m * \mathbf{x}_m - \mathbf{s} \right\|_1 + \lambda \sum_m \|\mathbf{x}_m\|_1 \;. \tag{2}$$

Ideally we would also include a gradient penalty term to assist in the representation of the low frequency image component. While this relatively straightforward, it is a bit more complex to

Fig. 5: *Denoised image (first method)*

implement, and is omitted from this example. Instead of including a representation of the low frequency image component within the optimization, we use the low frequency component estimated by the previous example, subtracting it from the signal passed to the CSC algorithm, and adding it back to the solution of this algorithm.

An algorithm for the problem in Equation (2) is not included in SPORCO, but there is an existing algorithm that can easily be adapted. CBPDN with mask decoupling, with mask array W,

$$\operatorname{argmin}_{\{\mathbf{x}_m\}} \frac{1}{2} \left\| W \left(\sum_m \mathbf{d}_m * \mathbf{x}_m - \mathbf{s} \right) \right\|_2^2 + \lambda \sum_m \|\mathbf{x}_m\|_1 \;, \tag{3}$$

is solved via the ADMM problem

$$\begin{aligned} &\operatorname{argmin}_{\mathbf{x},\mathbf{y}_0,\mathbf{y}_1} (1/2)\|W\mathbf{y}_0\|_2^2 + \lambda \|\mathbf{y}_1\|_1 \\ &\text{such that } \begin{pmatrix} D \\ I \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{s} \\ \mathbf{0} \end{pmatrix} \;, \end{aligned} \tag{4}$$

where $\mathbf{x} = \begin{pmatrix} \mathbf{x}_0^T & \mathbf{x}_1^T & \dots \end{pmatrix}^T$ and $D\mathbf{x} = \sum_m \mathbf{d}_m * \mathbf{x}_m$. We can express Equation (2) using the same variable splitting, as

$$\begin{aligned} &\operatorname{argmin}_{\mathbf{x},\mathbf{y}_0,\mathbf{y}_1} \|W\mathbf{y}_0\|_1 + \lambda \|\mathbf{y}_1\|_1 \\ &\text{such that } \begin{pmatrix} D \\ I \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{s} \\ \mathbf{0} \end{pmatrix} \;. \end{aligned} \tag{5}$$

(We don't need the W for the immediate problem at hand, but there isn't a good reason for discarding it.) Since Equation (5) has no $f(\mathbf{x})$ term (see Equation (1)), and has the same constraint as Equation (4), the $\mathbf{x}$ and $\mathbf{u}$ steps for these two problems are the same. The $\mathbf{y}$ step for Equation (4) decomposes into the two independent subproblems

$$\begin{aligned} \mathbf{y}_0^{(j+1)} &= \operatorname{argmin}_{\mathbf{y}_0} \frac{1}{2} \|W\mathbf{y}_0\|_2^2 + \frac{\rho}{2} \left\| \mathbf{y}_0 - (D\mathbf{x}^{(j+1)} - \mathbf{s} + \mathbf{u}_0^{(j)}) \right\|_2^2 \\ \mathbf{y}_1^{(j+1)} &= \operatorname{argmin}_{\mathbf{y}_1} \lambda \|\mathbf{y}_1\|_1 + \frac{\rho}{2} \left\| \mathbf{y}_1 - (\mathbf{x}^{(j+1)} + \mathbf{u}_1^{(j)}) \right\|_2^2 . \end{aligned}$$

The only difference between the ADMM algorithms for Equations (4) and (5) is in the $\mathbf{y}_0$ subproblem, which becomes

$$\mathbf{y}_0^{(j+1)} = \operatorname{argmin}_{\mathbf{y}_0} \|W\mathbf{y}_0\|_1 + \frac{\rho}{2} \left\| \mathbf{y}_0 - (D\mathbf{x}^{(j+1)} - \mathbf{s} + \mathbf{u}_0^{(j)}) \right\|_2^2 .$$

Therefore, the only modifications we expect to make to the class implementing the problem in Equation (3) are changing the computation of the functional value, and part of the $\mathbf{y}$ step.

We turn now to the implementation for this example. The module import statements and definitions of functions `pad` and `crop` are the same as for the example in section *Removal of Impulse Noise via CSC*, and are not repeated here. Our main task is to modify `cbpdn.ConvBPDNMaskDcpl`, the class for solving the problem in Equation (3), to replace the ℓ_2 norm data fidelity term with an ℓ_1 norm. The $\mathbf{y}$ step of this class is

```
def ystep(self):
    AXU = self.AX + self.U
    Y0 = (self.rho*(self.block_sep0(AXU) - self.S)) \
         / (self.W**2 + self.rho)
    Y1 = sl.shrink1(self.block_sep1(AXU),
                    (self.lmbda/self.rho)*self.wl1)
    self.Y = self.block_cat(Y0, Y1)

    super(ConvBPDNMaskDcpl, self).ystep()
```

where the `Y0` and `Y1` blocks of `Y` respectively represent $\mathbf{y}_0$ and $\mathbf{y}_1$ in Equation (5). All we need do to change the data fidelity term to an ℓ_1 norm is to modify the calculation of `Y0` to be a soft thresholding instead of the calculation derived from the existing ℓ_2 norm. We also need to override method `obfn_g0` so that the functional values are calculated correctly, taking into account the change of the data fidelity term. We end up with a definition of our class solving Equation (2) consisting of only a few lines of additional code

```
class ConvRepL1L1(cbpdn.ConvBPDNMaskDcpl):

  def ystep(self):

    AXU = self.AX + self.U
    Y0 = sl.shrink1(self.block_sep0(AXU) - self.S,
                    (1.0/self.rho)*self.W)
    Y1 = sl.shrink1(self.block_sep1(AXU),
                    (self.lmbda/self.rho)*self.wl1)
    self.Y = self.block_cat(Y0, Y1)

    super(cbpdn.ConvBPDNMaskDcpl, self).ystep()


  def obfn_g0(self, Y0):

    return np.sum(np.abs(self.W *
                         self.obfn_g0var()))
```

To solve the impulse denoising problem we load the reference image and dictionary, and construct the test image as before. We also need to load the low frequency component saved by the previous example

```
imglp = np.load('implslpc.npz')['imglp']
```

Now we initialize an instance of our new class, solve, and reconstruct the denoised estimate

```
lmbda = 3.0
b = ConvRepL1L1(D, pad(imgn) - imglp, lmbda,
                opt=opt, dimK=0)
X = b.solve()
imgdp = b.reconstruct().squeeze() + imglp
imgd = crop(imgdp)
```

The resulting denoised image is displayed in Figure 6.

Support Functions and Classes

In addition to the main set of classes for solving inverse problems, SPORCO provides a number of supporting functions and classes, within the following modules:

Fig. 6: *Denoised image (second method)*

- `util`: Various utility functions and classes, including a parallel-processing grid search for parameter optimization, access to a set of pre-learned convolutional dictionaries, and access to a set of example images.
- `plot`: Functions for plotting graphs or 3D surfaces and visualizing images, providing simplified access to Matplotlib functionality.
- `linalg`: Linear algebra and related functions, including solvers for specific forms of linear system and filters for computing image gradients.
- `metric`: Image quality metrics including standard metrics such as MSE, SNR, and PSNR.
- `cdict`: A constrained dictionary class that constrains the allowed dict keys, and also initializes the dict with default content on instantiation. All of the inverse problem algorithm options classes are derived from this class.

Conclusion

SPORCO is an actively maintained and thoroughly documented open source Python package for computing with sparse representations. While the primary design goal is ease of use and flexibility with respect to extensions of the supported algorithms, it is also intended to be computationally efficient and able to solve at least medium-scale problems. Standard sparse representations are supported, but the main focus is on convolutional sparse representations, for which SPORCO provides a wider range of features than any other publicly available library. The set of ADMM classes on which the optimization algorithms are based is also potentially useful for a much broader range of convex optimization problems.

Acknowledgment

Development of SPORCO was supported by the U.S. Department of Energy through the LANL/LDRD Program.

References

[All92] Stefano Alliney. Digital filters as absolute norm regularizers. *IEEE Transactions on Signal Processing*, 40(6):1548–1562, June 1992. doi:10.1109/78.139258.

[BEL13] Hilton Bristow, Anders Eriksson, and Simon Lucey. Fast convolutional sparse coding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 391–398, June 2013. doi:10.1109/CVPR.2013.57.

[BPC+10] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2010. doi:10.1561/2200000016.

[CCS10] Jian-Feng Cai, Emmanuel J. Candès, and Zuowei Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20(4):1956–1982, 2010. doi:10.1137/080738970.

[CDS98] Scott Shaobing Chen, David L. Donoho, and Michael A. Saunders. Atomic decomposition by basis pursuit. *SIAM Journal on Scientific Computing*, 20(1):33–61, 1998. doi:10.1137/S1064827596304010.

[GCW17] Cristina Garcia-Cardona and Brendt Wohlberg. Subproblem coupling in convolutional dictionary learning. In *Proceedings of IEEE International Conference on Image Processing (ICIP)*, September 2017. Accepted for presentation.

[GHW79] Gene H. Golub, Michael Heath, and Grace Wahba. Generalized cross-validation as a method for choosing a good ridge parameter. *Technometrics*, 21(2):215–223, May 1979. doi:10.1080/00401706.1979.10489751.

[HHW15] Felix Heide, Wolfgang Heidrich, and Gordon Wetzstein. Fast and flexible convolutional sparse coding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5135–5143, 2015. doi:10.1109/CVPR.2015.7299149.

[LS99] Michael S. Lewicki and Terrence J. Sejnowski. Coding time-varying signals using sparse, shift-invariant representations. In *Advances in Neural Information Processing Systems*, volume 11, pages 730–736, 1999.

[MBP14] Julien Mairal, Francis Bach, and Jean Ponce. Sparse modeling for image and vision processing. *Foundations and Trends in Computer Graphics and Vision*, 8(2-3):85–283, 2014. doi:10.1561/0600000058.

[ROF92] Leonid I. Rudin, Stanley Osher, and Emad Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60(1–4):259–268, 1992. doi:10.1016/0167-2789(92)90242-F.

[Ste81] Charles M. Stein. Estimation of the mean of a multivariate normal distribution. *The Annals of Statistics*, pages 1135–1151, November 1981.

[Woh14] Brendt Wohlberg. Efficient convolutional sparse coding. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 7173–7177, May 2014. doi:10.1109/ICASSP.2014.6854992.

[Woh16a] Brendt Wohlberg. Boundary handling for convolutional sparse representations. In *Proceedings of IEEE International Conference on Image Processing (ICIP)*, September 2016. doi:10.1109/ICIP.2016.7532675.

[Woh16b] Brendt Wohlberg. Convolutional sparse representation of color images. In *Proceedings of the IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI)*, pages 57–60, March 2016. doi:10.1109/SSIAI.2016.7459174.

[Woh16c] Brendt Wohlberg. Convolutional sparse representations as an image model for impulse noise restoration. In *Proceedings of the IEEE Image, Video, and Multidimensional Signal Processing Workshop (IVMSP)*, July 2016. doi:10.1109/IVMSPW.2016.7528229.

[Woh16d] Brendt Wohlberg. Efficient algorithms for convolutional sparse representations. *IEEE Transactions on Image Processing*, 25(1):301–315, January 2016. doi:10.1109/TIP.2015.2495260.

[Woh17] Brendt Wohlberg. ADMM penalty parameter selection by residual balancing, 2017. arXiv:1704.06209.

[ZKTF10] Matthew D. Zeiler, Dilip Krishnan, Graham W. Taylor, and Rob Fergus. Deconvolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2528–2535, June 2010. doi:10.1109/cvpr.2010.5539957.