SoftwareX 11 (2020) 100361

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

PyLops—A linear-operator Python library for scalable algebra and optimization

Matteo Ravasi [a,*], Ivan Vasconcelos [b]

[a] Equinor ASA - Sandslivegen 90, Sandsli, 5254, Norway
[b] Utrecht University, Vening Meineszgebouw A, Princetonlaan 8a, Room 242, 3584 CB, Utrecht, The Netherlands

ARTICLE INFO

Article history:
Received 4 April 2019
Received in revised form 13 November 2019
Accepted 19 November 2019

Keywords:
Python
Linear algebra
Inverse problems
Optimization
Linear operator

ABSTRACT

Linear operators and optimization are at the core of many algorithms used in signal and image processing, remote sensing, and inverse problems. For small to medium-scale problems, existing software packages (e.g., MATLAB, Python NumPy and SciPy) allow to explicitly build dense or sparse matrices and perform algebraic operations with syntax that closely represents their equivalent mathematical notation. However, many real-application, large-scale operators do not lend themselves to explicit matrix representations, usually forcing practitioners to forego the convenient linear-algebra syntax available for their explicit-matrix counterparts. PyLops is an open-source Python library providing a flexible framework for the creation and combination of so-called linear operators, class-based entities that represent matrices and inherit their associated syntax convenience, but do not rely on the creation of explicit matrices. We show that PyLops operators can dramatically reduce the memory load and CPU computations compared to explicit-matrix calculations, while still allowing users to seamlessly use their existing knowledge of compact matrix-based syntax that scales to any problem size because no explicit matrices are required.

Code metadata

Current code version	v1.7.0
Permanent link to code/repository used of this code version	https://github.com/ElsevierSoftwareX/SOFTX_2019_106
Legal Code License	LGPL-3.0
Code versioning system used	git
Software code languages, tools, and services used	Python; CI travis; Azure Pipelines; readthedocs; codacy; Docker
Compilation requirements, operating environments & dependencies	Python $\geq$ 3.5; Linux, OSX, Windows; Requirements provided in requirement.txt or environment.yml files in repository.
If available Link to developer documentation/manual	pylops.readthedocs.io
Support email for questions	matteoravasi@gmail.com

1. Introduction

Numerical linear algebra is at the core of many problems in signal processing [1], image processing [2], inverse problems [3,4] with applications to remote sensing [5], geophysics [6], medical imaging [7], and even some areas of machine learning such as deep neural networks [8].

Commonly used within these disciplines is the notion of *linear operator*, mapping vectors from one space, generally referred to as the model space, to another space, referred to as the data or observation space. Conversely, an *inverse problem* is the process of estimating from a set of observations the causal factors that produced them, that is, the underlying model (e.g., [3]).

Three alternative approaches can be identified for solving an inverse problem: direct solvers with explicit (dense or sparse) matrices, iterative solvers with explicit matrices, and iterative solvers with linear operators. For problems of limited size, one can first create a matrix and subsequently exploit the power of direct solvers to factorize the system of equations or use analytical pseudo-inverse formulas to invert such a matrix. In the latter case, the inverse matrix is multiplied to the observation vector to obtain an estimate of the model. However, this route is not always viable and iterative solvers such as gradient-descent

* Corresponding author.
E-mail address: matteoravasi@gmail.com (M. Ravasi).

https://doi.org/10.1016/j.softx.2019.100361

methods are usually employed. A clear advantage of these solvers is that one does not need direct access to the matrix, rather it only needs to be able to compute the forward and sometimes adjoint operations.

Nowadays, several software packages provide core functionalities for dealing with arrays and matrices, as well as a suite of direct and iterative numerical solvers: this is for example the case of MATLAB, the Python scientific libraries NumPy and SciPy, as well as the more low level libraries BLAS and LAPACK. Moreover, several open-source projects provide high-level, easy-to-use routines for the numerical treatment of ill-posed problems, such as the Regularization Tools package [9]. The widespread need to perform core linear algebra operations as efficiently and fast as possible has also led to large computing technology investments in the last two decades, focused on the search for efficient implementations on CPUs using both multi-threading as well as multi-core paradigms, GPUs, and more recently TPUs [10]. For example the field of machine learning has massively benefited from such advances; this is especially the case for *deep learning* where highly complex, deep neural networks with millions of weights (i.e., model parameters) can now be solved in a matter of hours, mostly because of the speedup in matrix-matrix and matrix–vector computations that GPUs and TPUs provide when compared to CPUs. Frameworks such as Theano [11], TensorFlow [12] or PyTorch [13] have been developed to specifically satisfy such a need and take advantage of advances in hardware components.

Nevertheless, when dealing with a large variety of physics-based inverse problems, the underlying linear operators are often far from being dense matrices (as opposed to, for example, dense layers in a neural network). Instead, they can be represented via sparse, structured matrices with fewer non-zero elements compared to the zero ones. By taking advantage of this property, we can write computer code for the linear operator resulting in a more efficient application of the forward and adjoint operations that scale with problem size. Such computer code can be written in a manner that inherits the syntax convenience of analytical linear algebra, by simply representing forward and adjoint operations via class-defined methods that reproduce the result of otherwise explicit matrix–vector products. This construct not only serves both sparse (e.g. physics-based) and dense operators (e.g., convolution with Green's functions or neural-network layers) equally well, but it also provides full functionality for the use of iterative solvers. Conveniently, the Python library `SciPy` provides a barebone, generic class for the definition and application of linear operators, which we leverage from and build on within the PyLops package as discussed below. Other examples of currently available software packages that provide a general interface to linear operators are the C++ `Rice Vector Library` [14], the MATLAB `Spot` and `Waveform` packages [15,16], the Python `fastmat` library [17], and the Julia `LinearMaps.jl` and `JOLI.jl` packages [18,19]. Moreover, some open-source software packages that employ a similar construct to solve domain-specific inverse problems are the `ASTRA-toolbox` [20], `Seplib` [6,21], `Madagascar` [22], and `Devito` [23]. Many of the packages mentioned above however tend to prioritize the ability of solving large inverse problems efficiently in exchange for a loss in the convenient linear-algebra syntax. To the best of our knowledge, only the MATLAB `Spot` package, and to a lesser degree the Python `fastmat` library, achieve the best of both worlds. PyLops is a Python library that accomplishes the very same goal while at the same time being more tightly connected to the Python ecosystem by directly building on top of the linear operator definition within the `SciPy` library.

2. A brief tour of linear operators

A discrete linear operator can be formally represented as a matrix–vector multiplication:

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{1}$$

where $\mathbf{A}$ in $\mathbb{R}^{(N\times M)}$ is an operator that maps a model vector $\mathbf{x}$ belonging to the real space $\mathbb{R}^M$ into a data vector $\mathbf{y}$ belonging to the real space $\mathbb{R}^N$. Note that the same theory is also valid for complex spaces.

The linear mapping from a known set of input parameters ($\mathbf{x}$) into a vector in the data space ($\mathbf{y}$) is generally referred to as *modeling* or the *forward problem*. Similarly, we can define the mapping from a vector in the data space to a vector in the model space as *adjoint modeling*. Finally, the process of undoing the effect of the modeling operator from a data vector is referred to as *inverse modeling* or the *inverse problem*.

As already mentioned, several linear mappings tend to obey to a certain structure and exploiting such a structure when applying them to a vector can usually lead to a significant gain in both computation speed and memory efficiency. This is for example the case of operators that can be expressed in terms of a convolution (correlation) between the model (data) vector and a compact kernel. Operators of such a kind can be implemented by creating a Toeplitz matrix that contains the elements of the kernel, followed by a matrix–vector multiplication with the model or data vector. When the kernel is compact, such matrix is a very sparse, band matrix with few non-zero elements around the main diagonal and zeros elsewhere. Performing the matrix–vector multiplication leads to poor performance, as many multiplications and summations with zero elements are performed. For example, imagine we want to apply a first-order derivative to a vector $\mathbf{x}$: the first-order derivative, in its simplest form, can be approximated by a two-sample forward difference stencil $[1/\Delta x, -1/\Delta x]$ applied to each pair of samples of the input vector; as for any convolutional operator with a generic kernel, the very same operation can be performed in three different ways:

1. Create a dense matrix with $1/\Delta x$ along the main diagonal and $-1/\Delta x$ along the first lower diagonal (and zero elsewhere), followed by a matrix–vector multiplication,
2. Convolve the input signal by the stencil,
3. Subtract each sample of the input vector by the previous sample, i.e., $y_i = (x_{i+1} - x_i)/\Delta x$.

This very last approach is the one adopted in the PyLops implementation of a first-order derivative as it does not only remove the need for storing $-1/\Delta x$ and $1/\Delta x$ values, but it also reduces the number of operations to one summation and one multiplication for each sample of the output vector. More in general, PyLops aims to provide efficient implementations of operators by exploiting their specific structure, and reduce both memory usage and computational cost.

An additional benefit of using linear operators becomes evident when attempting to solve an inverse problem. Without loss in generality, we consider the least-squares solution to an over-determined inverse problem ($n > m$):

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \; (J(\mathbf{x}) = \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2) \rightarrow \hat{\mathbf{x}} = (\mathbf{A}^H\mathbf{A})^{-1}\mathbf{A}^H\mathbf{y} \tag{2}$$

Notice how the solution $\hat{\mathbf{x}}$ that minimizes the cost function J requires both the operator $\mathbf{A}$ and its adjoint $\mathbf{A}^H$. This is not only the case when an explicit solution is used, but also when solving the problem by means of an iterative gradient-based solver. Working with explicit matrices requires creating and storing also the adjoint matrix, doubling the amount of data in memory. On

```python
import numpy as np
import pylops

# input signal parameters
ifreqs = [41, 25, 66]
amps = [1., 1., 1.]
nt = 200
nfft = 2**11
dt = 0.004
t = np.arange(nt)*dt
f = np.fft.rfftfreq(nfft, dt)

# input signal in frequency domain
X = np.zeros(nfft//2+1, dtype='complex128')
X[ifreqs] = amps

# input signal in time domain
FFTop = 10*pylops.signalprocessing.FFT(nt, nfft=nfft, real=True)
x = FFTop.H*X

# sampling locations
perc_subsampling = 0.2
ntsub = int(np.round(nt*perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(nt))[:ntsub])

# create operator
Rop = pylops.Restriction(nt, iava, dtype='float64')

# apply forward
y = Rop*x
ymask = Rop.mask(x)

# apply adjoint
xadj = Rop.H*y

# apply inverse
xinv = Rop / y

# regularized inversion
D2op = pylops.SecondDerivative(nt, dims=None, dtype='float64')

epsR = np.sqrt(0.1)
epsI = np.sqrt(1e-4)

xne = \
    pylops.optimization.leastsquares.NormalEquationsInversion(Rop, [D2op], y,
                                                              epsI=epsI,
                                                              epsRs=[epsR],
                                                              returninfo=False,
                                                              **dict(maxiter=50))

# sparse inversion
pfista, niterf, costf = \
    pylops.optimization.sparsity.FISTA(Rop*FFTop.H, y, niter=1000,
                                       eps=0.1, tol=1e-7, returninfo=True)
xfista = FFTop.H*pfista
```

Fig. 1. Code snippet for creation and application of forward, adjoint and inverse `Restriction` operator to a vector.

the other hand, linear operators can implement the adjoint in a similar fashion as the forward by exploiting the structure of the operator itself, leading to limited or in most cases no additional storage being required.

3. Code example

In this section we present a pedagogic example showing how the PyLops library can be used to frame and solve an interpolation problem by using linear operators. More specifically, we aim at interpolating onto a regular grid a one dimensional signal composed of three sinusoids that has been sampled at irregularly and coarsely spaced positions along the time axis. This is obtained by inverting the so-called restriction operator **R**, an operator that extracts a subset of N values at locations $\mathbf{l} = [l_1, l_2, l_M]$ (referred to as `iava` in the code) from an input (or model) vector $\mathbf{x}$ in forward mode:

$$y_i = x_{l_i} \quad \forall i = 1, 2, \dots, M \tag{3}$$

In adjoint mode, the action of the operator is such that values in the data vector $\mathbf{y}$ are placed at locations $\mathbf{l}$ in the model vector:

$$x_{l_i} = y_i \quad \forall i = 1, 2, \dots, M \tag{4}$$

where $x_j = 0 \quad \forall j = 1, 2, \dots, N \quad (j \notin \mathbf{l})$ (i.e., at all other locations in input vector).

In the following we present a complete code snippet and guide the reader through some of PyLops' code patterns. A more detailed description of the software package and its implementation details follows in Section 4.

3.1. Sample code snippet

Fig. 1 shows the code snippet used to solve the interpolation problem discussed above. We start by creating an input signal composed of three sinusoids in the frequency domain (lines 5–15), convert it to time domain using the `pylops.signalprocessing.FFT` linear operator (18–19), define indices for sampling the signal at irregular locations (22–24), create the `pylops.Restriction` operator (27), apply it in forward mode to the input signal (30–31), apply its adjoint to the calculated data (34), and finally invert the operator (37).

Fig. 2 shows that the operator **R** is ill-posed and the inverse problem cannot be successfully solved by simply employing the / operator. Such method does in fact implement the vanilla least-squares inversion (Eq. (1)) by means of the `scipy.sparse.linalg.lsqr` solver.

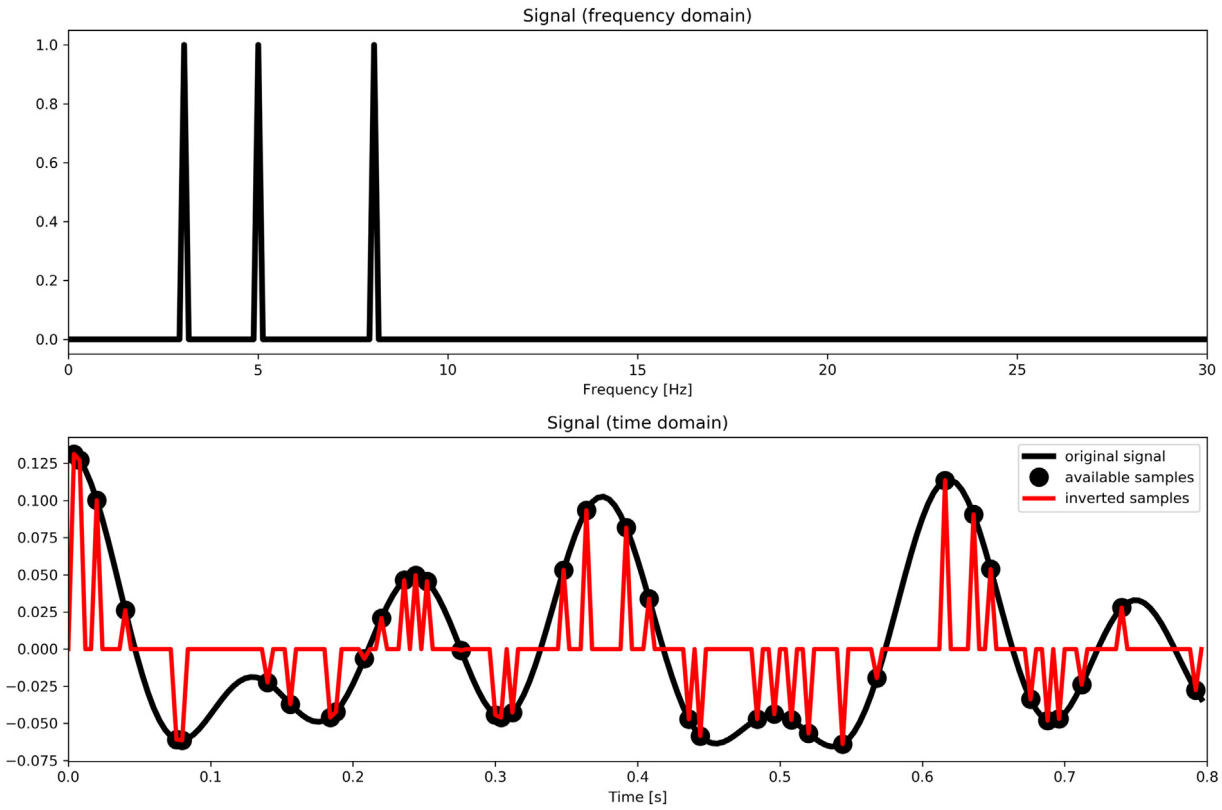

Fig. 2. Input signal in (a) frequency domain `X` and (b) time domain `x`. Sampled signal (`y` - green dots) and inverted signal (`xinv` - red) are also shown in panel (b).

In this example we show how we can improve our estimate by either (i) including a regularization term that favors a smooth model by penalizing its second-order derivative ($\mathbf{D}^2\mathbf{x}$), implemented via the `pylops.SecondDerivative` operator or (ii) taking advantage of the sparsity of the model in the frequency domain and by using a sparsity promoting solver such as `pylops.optimization.sparsity.FISTA` [24]. As shown in Fig. 3, the estimate of the input signal is very much improved in both cases.

Finally, while this example shows the potential of our library to solve inverse problems of any kind, several domain specific examples are created using Sphinx-Gallery and are available as part of the official documentation.[1] At this point in time, PyLops is particularly used within the geophysical imaging community as linear operators are ideally suited for solving multi-dimensional convolutional integral equations which are ubiquitous in processing of large multi-channel, time-series-based datasets (e.g., seismic data).[2]

4. Software framework

PyLops' goal is to provide an easy-to-use Application Programming Interface (API) to create and solve inverse problems by means of linear operators and express them in a way that mimics as closely as possible the mathematical linear algebra formalism used to describe the problem in the first place. Moreover, the library is suited to solve problems of any size, as shown in the benchmarking tests in Section 5. To achieve this goal, each linear operator is a class-based entity which can be used independently, combined together with other operators by means of basic mathematical operations (e.g., $+$, $-$, $*$; see below for more details), or fed directly into various solvers. Taking a modular approach to the creation of linear operators, the library makes it easy for other developers to implement new ones and to seamlessly include them in the framework. This ultimately enables the combination of any new and existing operators, providing an easy and quick way to experiment with novel problems.

The API can be loosely seen as composed of three interconnected units as shown in Fig. 4.

4.1. Linear operators

The first unit contains the entire suite of linear operators. `pylops.LinearOperator` is the main class of the library which is used as parent class for all other linear operators, such that they inherit its various internal methods as described below. Submodules are used to create an organized stack of operators and separate basic operators that are used within several applications from more domain-specific ones, such as those used within the `signalprocessing` submodule.

`pylops.LinearOperator` creates a generic interface for matrix–vector (and matrix-matrix) products that can ultimately be used to solve any forward or inverse problem of the form $\mathbf{y} = \mathbf{Ax}$. This is achieved by overloading the SciPy class `scipy.sparse.linalg.LinearOperator`, on top of which additional properties and methods are defined. Forward and adjoint matrix–vector operations are achieved by implementing the method `_matvec` for the forward, and `_rmatvec` for the adjoint. The attributes `shape` (tuple of two integers) and `dtype` must also be provided during initialization to identify the shape and data type of the operator itself. Moreover, `pylops.LinearOperator`

[1] The official documentation is hosted at pylops.readthedocs.io. Moreover, more in-depth code examples can be found at github.com/mrava87/pylops/_notebooks.

[2] Examples of such applications can be found at https://pylops.readthedocs.io/en/latest/tutorials/mdd.html#sphx-glr-tutorials-mdd-py and https://pylops.readthedocs.io/en/latest/tutorials/marchenko.html#sphx-glr-tutorials-marchenko-py.

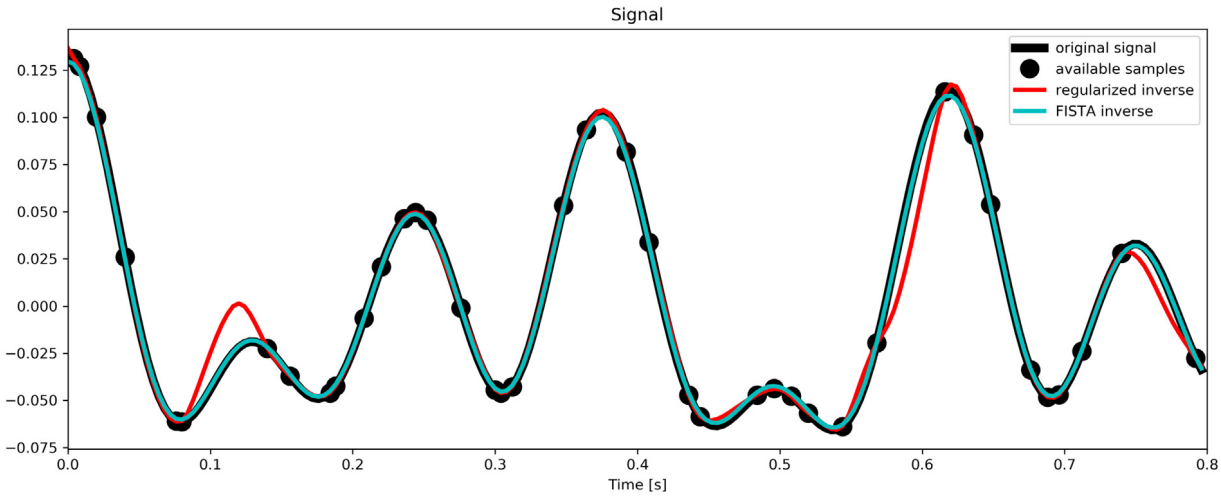

Fig. 3. Recovered signal using regularized least-squares inversion (red) and sparsity-promoting inversion (cyan), and input signal (black). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

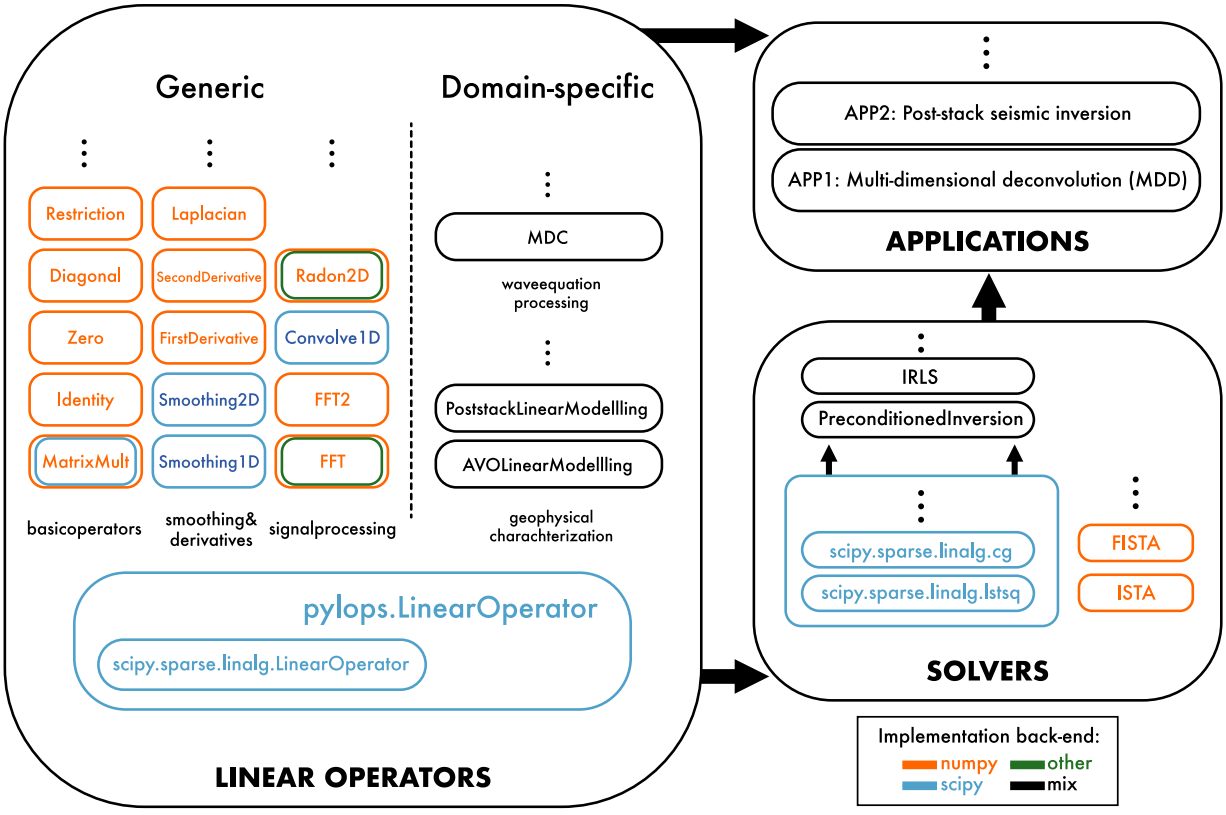

Fig. 4. Schematic representation of the software API. Colors indicate which library is used in the back-end of a linear operator (or solver).

requires an additional boolean attribute `explicit`, which identifies whether the operator has an explicit or implicit matrix representation. This allows the inference of the most appropriate solver to be used when invoking the `__truediv__` method as explained below.

As a linear operator is both concept- and syntax-wise equivalent to a matrix, enabling users to combine those operators actually reduces to implementing the following five elementary operations: sum, multiply by a scalar, multiply (or chain) operators, stack vertically and stack horizontally. In this context, vertical stacking is equivalent to creating a list that contains multiple operators, applying each operator to the model vector and concatenating the resulting data. On the other hand, horizontal stacking requires applying each operator to a portion of the model (of equal size to the operator column span) and summing the resulting data vectors.

To be able to write code that resembles as much as possible the underlying mathematical equations, we take advantage of the ability of Python to perform *operator overloading* of various magic methods – those being Python methods with the double underscores at the beginning and the end – to allow using mathematical symbols such as $+$, $-$, $*$, and $/$ to perform those elementary operations. More specifically, the following operator overloads are implemented:

- `__matmul__` (called via @) or `__mul__` (*): when applied to a NumPy `ndarray` vector, executes the forward computation of matrix-matrix or matrix–vector multiplication, respectively;
- `__mul__` (*): when applied to a scalar, left- or right-multiplies the operator by a scalar, while when applied to another `LinearOperator`, chains the two operators;
- `__add__` (+): when applied to another `LinearOperator`, sums the two operators;
- `H` and `T`: creates the transpose (or hermitian operator) and performs the adjoint computation when combined with a * (i.e., `.H*`);
- `__truediv__` (\): when applied to a NumPy `ndarray` vector, solves the inverse problem $\mathbf{y} = \mathbf{Ax}$ with either explicit or iterative solver.

Additionally, two other convenience methods are implemented within the `pylops.LinearOperator` class:

- `eigs`: estimate the singular values of the operator using the SciPy wrapper of ARPACK Fortran package [25].
- `cond`: use the `eigs` method to compute the conditioning number (the ratio of the largest-to-smallest eigenvalues).

4.2. Solvers

Solving a linear problem by means of an off-the-shelf least-squares cost function as in Eq. (1) may not always provide a good estimate of the input model (e.g., [3]). This is always the case in the presence of noisy data and for ill-posed linear operators that cannot be inverted directly such as the `Restriction` operator used in the numerical example in Section 3. In order to obtain an improved estimate of the input model, regularization terms can be included in the cost function. It is possible to either add terms which constrain the solution space such as the well-known Tikhonov regularization $\|\mathbf{x}\|_2$ or sparsity promoting terms such as $\|\mathbf{x}\|_1$ or to solve for a preconditioned model $\mathbf{p}$ such that $\mathbf{x} = \mathbf{Pp}$ where $\mathbf{P}$ could be a smoothing operator). While a large variety of linear solvers – e.g., conjugate-gradient solver [26] or the LSQR solver [27] – is currently available in the public domain, for example as part of the `SciPy` package, the user is generally left with the task of adding regularization and/or preconditioning terms. PyLops provides thin wrappers around some of those solvers and eases the use of regularization and/or preconditioning in inverse problems with minimal extra code. Our entire suite of *enriched* solvers is provided in the submodule `pylops.optimization` and subdivided into least-squares within `pylops.optimization.leastsquares` and sparsity-promoting solvers within `pylops.optimization.sparsity`.

4.3. Applications

Finally, the application layer is aimed at end users that wish to easily setup and solve specific problems (without digging into their implementation details — i.e., the creation and setup of linear operators and solvers). Various geophysical problems like those mentioned in the previous section are thus wrapped into a single high-level function call, which requires the user to simply provide the input dataset and a set of additional parameters.

5. Software dependencies

PyLops relies and builds on top of the two main external libraries for scientific computing in Python, namely `NumPy` [28] and `SciPy` [29], for all its linear operators and solvers.

In some circumstances, additional back-ends are also implemented to improve the performance of forward and adjoint operations. This is for example the case of the `FFT` operator, where a fast implementation of the Fast Fourier Transform (FFT) algorithm is provided by the library `pyfftw`, which is a python wrapper around the famous FFTW library [30]. In this case PyLops provides two back-end options (referred in the code as `engine`), one using NumPy's implementation of fft and ifft, and another using `pyfftw`. In the case a user uses `engine='fftw'` whilst not having `pyfftw` and `FTTW` installed, PyLops automatically falls back to the NumPy implementation. A similar approach is taken also for the `Radon2D` operator, where `numba` [31] is used in this case to speed-up for loops computations: again, a fallback NumPy engine is implemented to keep `numba` as an optional dependency.

6. Testing and operator validation

In the framework of linear operators, it is of vital importance to verify the correctness of the implementation of the forward and adjoint operations. Failure to do so may lead to sub-optimal convergence of iterative solvers when we attempt to invert a set of observations for their modeling operator. A very strong indication of the correctness of the two implementations is the so-called *dot-test* [6]. More specifically, two vectors $\mathbf{u}$ and $\mathbf{v}$ of size $[M \times 1]$ and $[N \times 1]$ are generated randomly, forward and adjoint operations performed as in Eq. (5), and the following equality tested within a certain tolerance:

$$(\mathbf{Op} * \mathbf{u})^H * \mathbf{v} = \mathbf{u}^H * (\mathbf{Op}^H * \mathbf{v}) \tag{5}$$

Alongside the dot-test, we always solve a small-scale inverse problem for every linear operator. The inverted model is compared to the original one used to model the data and it is checked that the two vectors match within a more or less strict tolerance. It is important to remember that some inverse problems, especially those with an under-determined operator ($N < M$), do not always have a unique solution and a satisfactory inverted model can only be obtained by including additional prior information in the form of additional regularization.

Tests have been implemented using `pytest` and are connected to two continuous integration systems (CI-Travis and Azure Pipelines). Automated tests cover all the linear operators, and multiple tests have been implemented to validate different combinations of both mandatory and optional input parameters. At the time of writing, PyLops has over 300 automated test with a code coverage of 86% (estimate provided by Codacy).

7. Contributing to the software

We foresee contributions from across different areas of scientific computing where inverse problems are applicable. In order to facilitate contributions we have created a check-list of four mandatory steps that are required for a new operator to be accepted to become part of the codebase of PyLops.[3] By strictly adhering to these requirements, we strive to keep a well-maintained, well-tested, and well-documented codebase, while strongly encouraging external contributions.

8. Benchmarking

Finally, we analyze the three different linear operators used in the example from the standpoints of computational performance and memory usage. For each operator, we perform a benchmark test comparing the time it takes to apply the forward operator to an input vector using the PyLops implementation of such an

[3] Refer to pylops.readthedocs.io/en/latest/adding.html for more details.

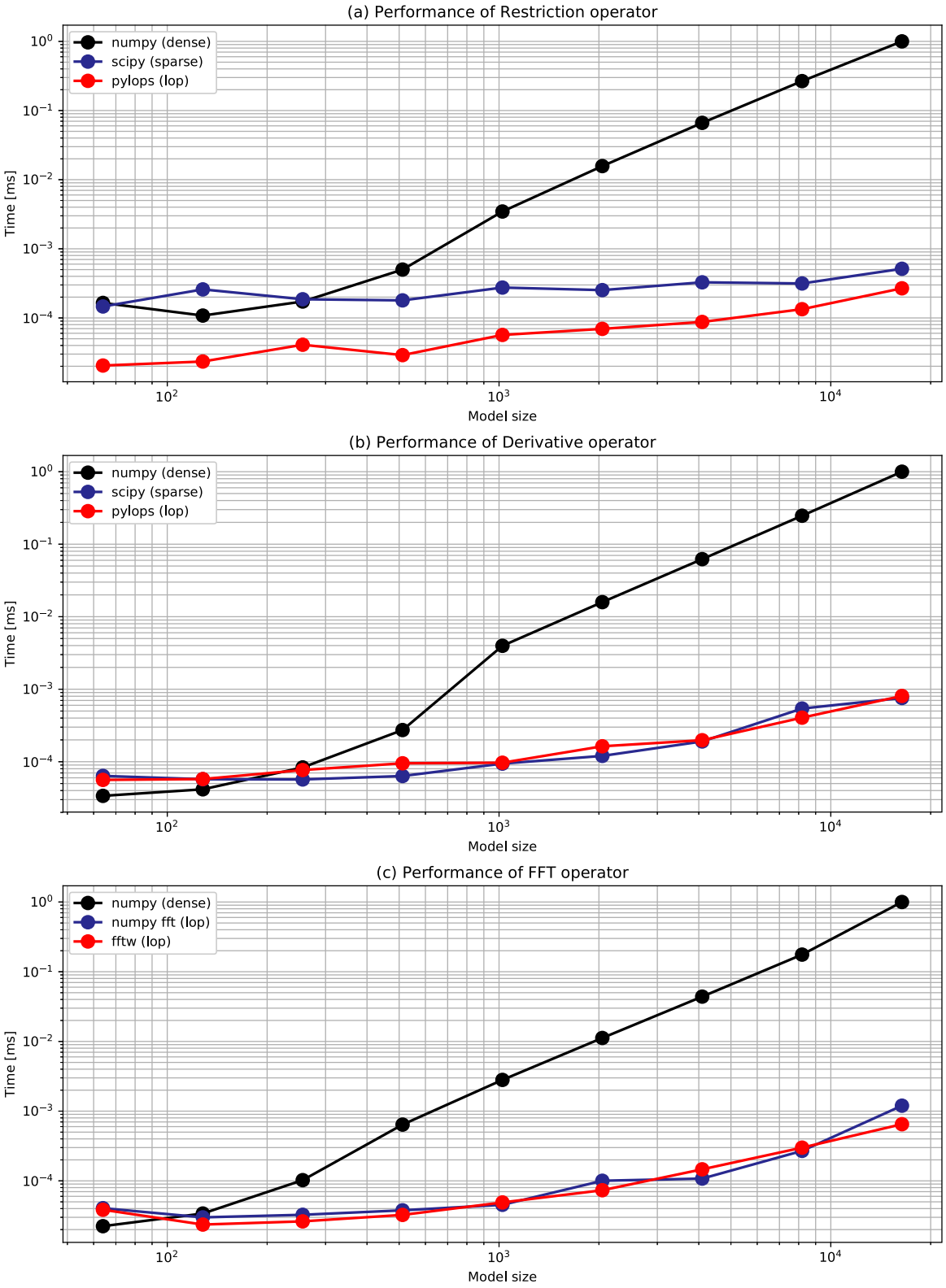

Fig. 5. Performance benchmark of (a) `Restriction` operator, (b) `FirstDerivative` operator, and (c) `FFT` operator.

operator versus the application of a dot product with a matrix that produces the same outcome. The comparison is done for operators of increasing size and the forward modeling is performed 200 times and logged via the Python `timeit.timeit` function. Comparisons are performed on a MacBook Air 1.3 GHz Intel Core i5 with a 8 GB 1600 MHz DDR3 RAM. Moreover, NumPy and SciPy are installed via the conda distribution and linked to the Intel MKL implementation of BLAS library for linear algebra. This leads to the best performance for the dot product on a CPU architecture as discussed in [32].

For the `Restriction` operator (Fig. 5a) we create both a dense matrix using `numpy.ndarray` and a sparse matrix using

`scipy.sparse.csr_matrix` as well as a PyLops operator. PyLops' implementation outperforms the naive dot product with either dense or sparse matrices. Moreover, if we consider a model vector with $M = 10^5$, and a subsampling factor of 10, the resulting data vector has size $N = 10^4$. The dense matrix used to perform such a restriction has therefore $N * M = 10^9$ elements. Using 8-bit unsigned integers, this amounts to 8 GB of memory to store the matrix, and the same for its adjoint.

The memory usage is dramatically reduced for a sparse matrix, as three values need to be stored for each index where the input signal is sampled (row index, column index, and value); this amounts to $3 * N = 3 * 10^4$ elements (120 kB if we use int32 type – 4 Bytes – for indices and values). A linear operator requires instead only storing the indices at which the input signal is sampled; in this case, that means only $N = 10^4$ values (40 kB if we use int32 type for the indices).

We now consider the two-point `FirstDerivative` operator (Fig. 5b). This operator is convolutional in essence as it can be applied by convolving the input signal by a compact filter. The PyLops implementation outperforms the explicit dot product with a dense NumPy matrix, while a similar performance is obtained in this case when a using a sparse-matrix. Though true for this isolated benchmark, we note that in real-applications multiple operators are generally chained (or stacked). Chaining explicit matrices generally increases the complexity of the resulting matrix and *densifies* it, meaning that the resulting matrix is less sparse and the dot product less efficient. This is not the case for linear operators, where the computational time of a chained operator is equivalent to the sum of computational time of each operator. Moreover, the memory usage for the `FirstDerivative` operator reduces to a single value, the step size Δx, while for a dense matrix the size quadratically increases with the size of the model.

Lastly, we benchmark the Fast Fourier Transform `FFT` operator (Fig. 5c). This is a peculiar case, as the FFT can be easily written as a fully-dense matrix and combined with other dense matrices as well as applied by means of a matrix–vector product. Using a linear operator we can however leverage available open-source implementations of the FFT algorithm such as those in `NumPy` or `FFTW` libraries. The operator storage in this case is also limited to a single number, the size of the FFT, while the required storage for the corresponding dense matrix increases again quadratically with the size of the model.

As a final remark, we wish to point out that linear operators should not always be preferred to dense matrices. It is clear from our benchmark tests that for small scale problems ($N \leq 10^2$), the most performant implementation is represented by the highly optimized dot-product available in NumPy. Nevertheless, both memory and computation benefits arise when using linear operators for problems of larger size ($N > 10^2$) and the linear operator paradigm should be the go-to solution to efficiently solve large scale inverse problems.

9. Conclusions

We present a general-purpose Python library for linear optimization, which scales from didactic numerical experiments to large-scale, real-life problems. Using the concept of *linear operator classes* and taking advantage of operator overloading in Python, a framework is created whereby linear forward and inverse problems can be solved in a fully scalable manner (from tens to millions of model parameters) without the need to store large matrices in memory. By design, PyLops maintains a compact notation that closely mimics the underlying analytical linear-algebra formulation of any chosen problem. Benchmark testing confirms that linear operators in PyLops scale well and efficiently with respect to more 'naive' implementations of the same operators by means of explicit matrices. Moreover, the software architecture is created in a modular fashion, in such a way that it is very straightforward to create and include new linear operators (or solvers). Although not part of the current version of the project, the framework is not limited to linear inverse problems: PyLops could be used for solving nonlinear inverse with optimization methods that rely on linearized forward modeling, such as the widespread adjoint-state method. Similarly, despite most of the current applications are focused on geophysical processing and imaging, the framework is generic and suited for any other discipline that deals with complex inverse problems, or requires complex linear algebra calculations. One point of note is that in its current form the *PyLops* operator objects capture some of the algebraic properties of their theoretical counterparts — namely those related to metric spaces and operations commonly invoked in optimization and inverse problems. For those interested in extending operator objects to include other algebraic properties, our software framework would easily allow for such extensions, thereby facilitating advances in numerical analysis in support of analytical algebra. Finally, two recent additions to the *PyLops* project, namely *PyLops-GPU* and *PyLops-distributed*, take advantage of high-performance computing (GPUs and out-of-memory computations via distributed computing) when dealing with linear operators and solvers.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

MR thanks Equinor for allowing the publication of this work. We also thank Joost van der Neut, Yanadet Sripanich, and Tristan van Leeuwen for insightful discussions. Jupyter notebooks are used to create Figs. 2, 3, and 5 can be found at github.com/mrava87/pylops_notebooks/tree/master/papers/softwareX_2019. The authors cannot be held liable for any inappropriate use of this software library.

References

[1] Kay SM. Fundamentals of statistical signal processing: Estimation. Prentice Hall; 1993.
[2] Gonzalez RC, Woods RE. Digital image processing. Pearson Education; 2017.
[3] Hansen PC. Discrete inverse problems: Insight and algorithms (fundamentals of algorithms). Society for Industrial and Applied Mathematics; 2010.
[4] Bertero M, Boccacci P. Introduction to inverse problems in imaging. CRC Press; 1998.
[5] Twomey S. Introduction to the mathematics of inversion in remote sensing and indirect measurements. Dover Publications; 1997.
[6] Claerbout J, Fomel S. Geophysical image estimation by example, Reading, MA.
[7] Suetens P. Fundamentals of medical imaging. Cambridge University Press; 2009.
[8] Goodfellow I, Bengio Y, Courville A. Deep learning. MIT Press Ltd; 2017.
[9] Hansen PC. Regularization tools version 4.0 for matlab 7.3. Numer Algorithms 2007;46:189–94.
[10] Jouppi NP, Young C, Patil N, co authors. In-datacenter performance analysis of a tensor processing unit. 2017, Arxiv.
[11] Al-Rfou R, Alain G, Almahairi A, Angermueller C, et al., Theano Development Team Collaboration Theano: A Python framework for fast computation of mathematical expressions. 2016, arXiv e-prints, abs/1605.02688, URL http://arxiv.org/abs/1605.02688.
[12] Abadi M, Agarwal A, Barham P, et al. Tensorflow: Large-scale machine learning on heterogeneous systems. 2015, URL https://www.tensorflow.org/, Software available from tensorflow.org.

[13] Paszke A, Gross S, Chintala S, Chanan G, et al. Automatic differentiation in pytorch. In: NIPS-W. 2017.
[14] D.Padula A, Scott SD, Symes WW. A software framework for abstract expression of coordinate-free linear algebra and optimization algorithms. ACM Trans Math Software 2009;36(2):8:1–36. http://dx.doi.org/10.1145/1499096.1499097, URL http://doi.acm.org/10.1145/1499096.1499097.
[15] van den Berg E, Friedlander MP. SPOT – a linear-operator toolbox. 2013, URL http://www.cs.ubc.ca/labs/scl/spot/.
[16] Silva CD, Herrmann FJ. A unified 2d/3d large-scale software environment for nonlinear inverse problems. ACM Trans Math Software 2019.
[17] Wagner C, Semper S. Fast linear transformations in python. Math Softw 2017.
[18] Karrasch D, contributors. LinearMaps.jl – a julia package for defining and working with linear maps, also known as linear transformations or linear operators acting on vectors. 2019, https://github.com/Jutho/LinearMaps.jl.
[19] Modzelewski H, Louboutin M. Joli.jl – julia framework for constructing matrix-free linear operators with explicite domain/range type control and applying them in basic algebraic matrix-vector operations. 2019, https://github.com/slimgroup/JOLI.jl.
[20] van Aarle W, Palenstijn WJ, Beenhouwer JD, Altantzis T, Bals S, Batenburg KJ, Sijbers J. The ASTRA toolbox: A platform for advanced algorithm development in electron tomography. Ultramicroscopy 2015;157:35–47.
[21] Clapp R. Seplib. In: 74th EAGE conference and exhibition - workshops. 2012.
[22] Fomel S, Sava P, Ioan Vlad YL, Bashkardin V. Madagascar: open-source software project for multidimensional data analysis and reproducible computational experiments. J Open Res Softw 2013;1(1). e8.
[23] Louboutin M, Lange M, Luporini F, Kukreja N, Witte PA, Herrmann FJ, Velesko P, Gorman GJ. Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. Geosci Model Dev 2019;12:1165–87, URL https://doi.org/10.5194/gmd-12-1165-2019.
[24] Beck A, Teboulle M. Shrinkage-thresholding algorithm for linear inverse problems. SIAM J Imaging Sci 2009;183–202.
[25] Lehoucq RB, Sorensen DC, Yang C. Arpack users' guide solution of large-scale eigenvalue problems with implicitly restarted arnoldi methods. Soc Ind Appl Math 1998.
[26] Hestenes MR, Stiefel E. Methods of conjugate gradients for solving linear systems. J Res Natl Bur Stand 1952;49.
[27] Paige CC, Saunders MA. LSQR: An algorithm for sparse linear equations and sparse least squares. ACM TOMS 1952;8.
[28] Oliphant T. Numpy: a guide to numpy. USA: Trelgol Publishing; 2006, URL http://www.numpy.org/.
[29] Eric Jones PP, Travis Oliphant. Scipy: Open source scientific tools for python. 2001.
[30] Frigo M, Johnson SG. FFTW: an adaptive software architecture for the fft. In: IEEE international conference on acoustics, speech and signal processing, Vol. 3. 1998, p. 1381–4.
[31] Lam SK, Pitrou A, Seibert S. Numba: A llvm-based python jit compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. LLVM '15, New York, NY, USA: ACM; 2015, p. 7:1–6. http://dx.doi.org/10.1145/2833157.2833162, URL http://doi.acm.org/10.1145/2833157.2833162.
[32] Bauke H. Boosting NumPy with MKL. 2016, URL https://www.numbercrunch.de/blog/2016/03/boosting-numpy-with-mkl/.