This Python library is inspired by the MATLAB Spot – A Linear-Operator Toolbox project.

Linear operators and inverse problems are at the core of many of the most used algorithms in signal processing, image processing, and remote sensing. When dealing with small-scale problems, the Python numerical scientific libraries numpy and scipy allow to perform most of the underlying matrix operations (e.g., computation of matrix-vector products and manipulation of matrices) in a simple and expressive way.

Many useful operators, however, do not lend themselves to an explicit matrix representation when used to solve large-scale problems. PyLops operators, on the other hand, still represent a matrix and can be treated in a similar way, but do not rely on the explicit creation of a dense (or sparse) matrix itself. Conversely, the forward and adjoint operators are represented by small pieces of codes that mimic the effect of the matrix on a vector or another matrix.

Luckily, many iterative methods (e.g. cg, lsqr) do not need to know the individual entries of a matrix to solve a linear system. Such solvers only require the computation of forward and adjoint matrix-vector products as done for any of the PyLops operators.

Here is a simple example showing how a dense first-order first derivative operator can be created, applied and inverted using numpy/scipy commands:

```python
import numpy as np
from scipy.linalg import lstsq

nx = 7
x = np.arange(nx) - (nx-1)/2

D = np.diag(0.5*np.ones(nx-1), k=1) - np.diag(0.5*np.ones(nx-1), k=-1)
D[0] = D[-1] = 0 # take away edge effects

y = np.dot(D, x)
xadj = np.dot(D.T, y)
xinv = lstsq(D, y)[0]
```

and similarly using PyLops commands:

```python
from pylops import FirstDerivative

Dlop = FirstDerivative(nx, dtype='float64')

y = Dlop*x
xadj = Dlop.H*y
xinv = Dlop / y
```

Note how this second approach does not require creating a dense matrix, reducing both the memory load and the computational cost of applying a derivative to an input vector $x$. Moreover, the code becomes even more compact and expressive than in the previous case letting the user focus on the formulation of equations of the forward problem to be solved by inversion.
A common terminology is used within the entire documentation of PyLops. Every linear operator and its application to a model will be referred to as **forward model (or operation)**

\[ y = Ax \]

while its application to a data is referred to as **adjoint modelling (or operation)**

\[ x = A^H y \]

where \( x \) is called model and \( y \) is called data. The operator \( A : \mathbb{F}^m \to \mathbb{F}^n \) effectively maps a vector of size \( m \) in the **model space** to a vector of size \( n \) in the **data space**, conversely the adjoint operator \( A^H : \mathbb{F}^n \to \mathbb{F}^m \) maps a vector of size \( n \) in the **data space** to a vector of size \( m \) in the **model space**. As linear operators mimics the effect a matrix on a vector we can also loosely refer to \( m \) as the number of **columns** and \( n \) as the number of **rows** of the operator.

Ultimately, solving an inverse problems accounts to removing the effect of \( A \) from the data \( y \) to retrieve the model \( x \).

For a more detailed description of the concepts of linear operators, adjoints and inverse problems in general, you can head over to one of Jon Claerbout’s books such as *Basic Earth Imaging*. 
PyLops is build on top of the scipy class `scipy.sparse.linalg.LinearOperator`.

This class allows in fact for the creation of objects (or interfaces) for matrix-vector and matrix-matrix products that can ultimately be used to solve any inverse problem of the form $y = Ax$.

As explained in the scipy LinearOperator official documentation, to construct a `scipy.sparse.linalg.LinearOperator`, a user is required to pass appropriate callables to the constructor of this class, or subclass it. More specifically one of the methods `_matvec` and `_matmat` must be implemented for the *forward operator* and one of the methods `_rmatvec` or `_adjoint` may be implemented to apply the *Hermitian adjoint*. The attributes/properties `shape` (pair of integers) and `dtype` (may be None) must also be provided during `__init__` of this class.

Any linear operator developed within the PyLops library follows this philosophy. As explained more in details in Implementing new operators section, a linear operator is created by subclassing the `scipy.sparse.linalg.LinearOperator` class and `_matvec` and `_rmatvec` are implemented.
PyLops was initially written and it is currently maintained by Equinor. It is a flexible and scalable python library for large-scale optimization with linear operators that can be tailored to our needs, and as contribution to the free software community.

## 3.1 Installation

The PyLops project strives to create a library that is easy to install in any environment and has a very limited number of dependencies. However, since Python2 will retire soon, we have decided to only focus on a Python3 implementation. If you are still using Python2, hurry up!

For this reason you will need **Python 3.6 or greater** to get started.

### 3.1.1 Dependencies

Our mandatory dependencies are limited to:

- numpy
- scipy

We advise using the Anaconda Python distribution to ensure that these dependencies are installed via the Conda package manager. This is not just a pure stylistic choice but comes with some hidden advantages, such as the linking to Intel MKL library (i.e., a highly optimized BLAS library created by Intel).

If you simply want to use PyLops for teaching purposes or for small-scale examples, this should not really affect you. However, if you are interested in getting better code performance, read carefully the Advanced installation page.

### 3.1.2 Optional dependencies

PyLops’s optional dependencies refer to those dependencies that we do not include in our requirements.txt and environment.yml files and thus are not strictly needed nor installed directly as part of a standard installation (see...
However, we sometimes implement additional back-ends (referred to as engine in the code) for some of our operators in order to improve their performance. To do so, we rely on third-party libraries. Those libraries are generally added to the list of our optional dependencies. If you are not after code performance, you may simply stick to the mandatory dependencies and pylops will ensure to always fallback to one of those for any linear operator.

If you are instead after code performance, take a look at the Optional dependencies section in the Advanced installation page.

### 3.1.3 Step-by-step installation for users

#### Python environment

Activate your Python environment, and simply type the following command in your terminal to install the PyPi distribution:

```bash
>> pip install pylops
```

If using Conda, you can also install our conda-forge distribution via:

```bash
>> conda install -c conda-forge pylops
```

Note that using the conda-forge distribution is recommended as all the dependencies (both mandatory and optional) will be correctly installed for you, while only mandatory dependencies are installed using the pip distribution.

Alternatively, to access the latest source from github:

```bash
>> pip install https://github.com/equinor/pylops/archive/master.zip
```

or just clone the repository

```bash
>> git clone https://github.com/equinor/pylops.git
```

or download the zip file from the repository (green button in the top right corner of the main github repo page) and install PyLops from terminal using the command:

```bash
>> make install
```

#### Docker

If you want to try PyLops but do not have Python in your local machine, you can use our Docker image instead.

After installing Docker in your computer, type the following command in your terminal (note that this will take some time the first time you type it as you will download and install the docker image):

```bash
>> docker run -it -v /path/to/local/folder:/home/jupyter/notebook -p 8888:8888
   mrava87/pylops:notebook
```

This will give you an address that you can put in your browser and will open a jupyter-notebook environment with PyLops and other basic Python libraries installed. Here /path/to/local/folder is the absolute path of a local folder on your computer where you will create a notebook (or containing notebooks that you want to continue working on). Note that anything you do to the notebook(s) will be saved in your local folder.

A larger image with Conda distribution is also available. Simply use conda_notebook instead of notebook in the previous command.
3.1.4 Step-by-step installation for developers

Fork and clone the repository by executing the following in your terminal:

```bash
$ git clone https://github.com/your_name_here/pylops.git
```

The first time you clone the repository run the following command:

```bash
$ make dev-install
```

If you prefer to build a new Conda environment just for PyLops, run the following command:

```bash
$ make dev-install_conda
```

To ensure that everything has been setup correctly, run tests:

```bash
$ make tests
```

Make sure no tests fail, this guarantees that the installation has been successfull.

If using Conda environment, always remember to activate the conda environment every time you open a new `bash` shell by typing:

```bash
$ source activate pylops
```

3.2 Advanced installation

In this section we discuss some important details regarding code performance when using PyLops.

To get the most out of PyLops operators in terms of speed you will need to follow these guidelines as much as possible or ensure that the Python libraries used by PyLops are efficiently installed (e.g., allow multithreading) in your system.

3.2.1 Dependencies

PyLops relies on the `numpy` and `scipy` libraries and being able to link these to the most performant BLAS will ensure optimal performance of PyLops when using only required dependencies.

As already mentioned in the Installation page, we strongly encourage using the Anaconda Python distribution as `numpy` and `scipy` will be automatically linked to the Intel MKL library, which is per today the most performant library for basic linear algebra operations (if you don’t believe it, take a read at this blog post).

The best way to understand which BLAS library is currently linked to your `numpy` and `scipy` libraries is to run the following commands in ipython:

```python
import numpy as np
import scipy as sp
print(np.__config__.show())
print(sp.__config__.show())
```

You should be able to understand if your `numpy` and `scipy` are linked to Intel MKL or something else.

**Note:** Unfortunately, PyLops is so far only shipped with PyPI, meaning that if you have not already installed `numpy` and `scipy` in your environment they will be installed as part of the installation process of the `pylops` library, all of those using `pip`. This comes with the disadvantage that `numpy` and `scipy` are linked to OpenBlas instead of Intel MKL, leading to a loss of performance. To prevent this, we suggest the following strategy:
create conda environment, e.g. `conda create -n envname python=3.6.4 numpy scipy`
install pylops using `pip install pylops`

Finally, it is always important to make sure that your environment variable `OMP_NUM_THREADS` is correctly set to the maximum number of threads you would like to use in your code. If that is not the case `numpy` and `scipy` will underutilize your hardware even if linked to a performant BLAS library.

For example, first set `OMP_NUM_THREADS=1` (single-threaded) in your terminal:

```bash
>> export OMP_NUM_THREADS=1
```

and run the following code in python:

```python
import os
import numpy as np
from timeit import timeit

size = 4096
A = np.random.random((size, size)),
B = np.random.random((size, size))
print('Time with %s threads: %f s' %
     (os.environ.get('OMP_NUM_THREADS'),
      timeit(lambda: np.dot(A, B), number=4)))
```

Subsequently set `OMP_NUM_THREADS=2`, or any higher number of threads available in your hardware (multi-threaded):

```bash
>> export OMP_NUM_THREADS=2
```

and run the same python code. By both looking at your processes (e.g. using `top`) and at the python print statement you should see a speed-up in the second case.

Alternatively, you could set the `OMP_NUM_THREADS` variable directly inside your script using `os.environ['OMP_NUM_THREADS']=str(2)`. Moreover, note that when using Intel MKL you can alternatively set the `MKL_NUM_THREADS` instead of `OMP_NUM_THREADS`: this could be useful if your code runs other parallel processes which you can control independently from the Intel MKL ones using `OMP_NUM_THREADS`.

**Note:** Always remember to set `OMP_NUM_THREADS` (or `MKL_NUM_THREADS`) in your environment when using PyLops

### 3.2.2 Optional dependencies

To avoid increasing the number of `required` dependencies, which may lead to conflicts with other libraries that you have in your system, we have decided to build some of the additional features of PyLops in such a way that if an `optional` dependency is not present in your python environment, a safe fallback to one of the required dependencies will be enforced.

When available in your system, we recommend using the Conda package manager and install all the mandatory and optional dependencies of PyLops at once using the command:

```bash
>> conda install -c conda-forge pylops
```

in this case all dependencies will be installed from their conda distributions.

Alternatively, from version 1.4.0 optional dependencies can also be installed as part of the pip installation via:
Available from PyPI (via wheel) are:

- `pylops` core modules
- `pylops.waveeqprocessing` modules
- `pylops.signalprocessing` modules
- Additional packages (see below)

You can install all available modules via:

```
$ pip install pylops[advanced]
```

Dependencies are however installed from their PyPI wheels.

**numba**

Although we always strive to write code for forward and adjoint operators that takes advantage of the perks of numpy and scipy (e.g., broadcasting, ufunc), in some case we may end up using for loops that may lead to poor performance. In those cases we may decide to implement alternative (optional) back-ends in numba.

In this case a user can simply switch from the native, always available implementation to the numba implementation by simply providing the following additional input parameter to the operator `engine='numba'`. This is for example the case in the `pylops.signalprocessing.Radon2D`.

If interested to use `numba` backend from conda, you will need to manually install it:

```
$ conda install numba
```

Finally, it is also advised to install the additional package `icc_rt`.

```
$ conda install -c numba icc_rt
```

or pip equivalent. Similarly to Intel MKL, you need to set the environment variable `NUMBA_NUM_THREADS` to tell numba how many threads to use. If this variable is not present in your environment, numba code will be compiled with `parallel=False`.

**fft routines**

Two different engines are provided by the `pylops.signalprocessing.FFT` operator for fft and ifft routines in the forward and adjoint modes: `engine='numpy'` (default) and `engine='fftw'`.

The first engine comes as default as numpy is part of the dependencies of PyLops and automatically installed when PyLops is installed if not already available in your Python distribution.

The second engine implements the well-known FFTW via the python wrapper `pyfftw.FFTW`. This optimized fft tends to outperform the one from numpy in many cases, however it has not been inserted in the mandatory requirements of PyLops, meaning that when installing PyLops with `pip`, `pyfftw.FFTW` will *not* be installed automatically.

Again, if interested to use FFTW backend from conda, you will need to manually install it:

```
$ conda install -c conda-forge pyfftw
```

or pip equivalent.

**skfmm**

This library is used to compute traveltime tables with the fast-marching method in the initialization of the `pylops.waveeqprocessing.Demigration` operator when choosing `mode == 'eikonal'`.

As this may not be of interest for many users, this library has not been inserted in the mandatory requirements of PyLops. If interested to use `skfmm`, you will need to manually install it:

```
$ conda install -c conda-forge scikit-fmm
```

or pip equivalent.

### 3.2. Advanced installation

- `pip install pylops[advanced]`

  Dependencies are however installed from their PyPI wheels.

- **numba**

  Although we always strive to write code for forward and adjoint operators that takes advantage of the perks of numpy and scipy (e.g., broadcasting, ufunc), in some case we may end up using for loops that may lead to poor performance. In those cases we may decide to implement alternative (optional) back-ends in numba.

  In this case a user can simply switch from the native, always available implementation to the numba implementation by simply providing the following additional input parameter to the operator `engine='numba'`. This is for example the case in the `pylops.signalprocessing.Radon2D`.

  If interested to use `numba` backend from conda, you will need to manually install it:

  ```
  $ conda install numba
  ```

  Finally, it is also advised to install the additional package `icc_rt`.

  ```
  $ conda install -c numba icc_rt
  ```

  or pip equivalent. Similarly to Intel MKL, you need to set the environment variable `NUMBA_NUM_THREADS` to tell numba how many threads to use. If this variable is not present in your environment, numba code will be compiled with `parallel=False`.

- **fft routines**

  Two different engines are provided by the `pylops.signalprocessing.FFT` operator for fft and ifft routines in the forward and adjoint modes: `engine='numpy'` (default) and `engine='fftw'`.

  The first engine comes as default as numpy is part of the dependencies of PyLops and automatically installed when PyLops is installed if not already available in your Python distribution.

  The second engine implements the well-known FFTW via the python wrapper `pyfftw.FFTW`. This optimized fft tends to outperform the one from numpy in many cases, however it has not been inserted in the mandatory requirements of PyLops, meaning that when installing PyLops with `pip`, `pyfftw.FFTW` will *not* be installed automatically.

  Again, if interested to use FFTW backend from conda, you will need to manually install it:

  ```
  $ conda install -c conda-forge pyfftw
  ```

  or pip equivalent.

- **skfmm**

  This library is used to compute traveltime tables with the fast-marching method in the initialization of the `pylops.waveeqprocessing.Demigration` operator when choosing `mode == 'eikonal'`.

  As this may not be of interest for many users, this library has not been inserted in the mandatory requirements of PyLops. If interested to use `skfmm`, you will need to manually install it:

  ```
  $ conda install -c conda-forge scikit-fmm
  ```

  or pip equivalent.
3.3 Extensions

PyLops brings to users the power of linear operators in a simple and easy to use programming interface. While very powerful on its own, this library is further extended to take advantage of more advanced computational resources, either in terms of multiple-node clusters or GPUs.

Two spin-off projects aim at extending the capabilities of PyLops to such computational environments:

- PyLops-GPU
- PyLops-Distributed

3.4 Tutorials

3.4.1 01. The LinearOperator

This first tutorials is aimed at easing the use of the PyLops library for both new users and developers.

Since PyLops heavily relies on the use of the scipy.sparse.linalg.LinearOperator class of scipy, we will start by looking at how to initialize a linear operator as well as different ways to apply the forward and adjoint operations. Finally we will investigate various special methods, also called magic methods (i.e., methods with the double underscores at the beginning and the end) that have been implemented for such a class and will allow summing, subtracting, chaining, etc. multiple operators in very easy and expressive way.

Let’s start by defining a simple operator that applies element-wise multiplication of the model with a vector \( d \) in forward mode and element-wise multiplication of the data with the same vector \( d \) in adjoint mode. This operator is present in PyLops under the name of pylops.Diagonal and its implementation is discussed in more details in the Implementing new operators page.
First of all we apply the operator in the forward mode. This can be done in four different ways:

- **\_matvec**: directly applies the method implemented for forward mode
- **matvec**: performs some checks before and after applying \_matvec
- **\*:** operator used to map the special method \_matmul\_ which checks whether the input x is a vector or matrix and applies \_matvec or \_matmul accordingly.
- **\@**: operator used to map the special method \_mul\_ which performs like the * operator

We will time these 4 different executions and see how using \_matvec (or matvec) will result in the faster computation. It is thus advised to use * (or @) in examples when expressivity has priority but prefer \_matvec (or matvec) for efficient implementations.
plt.plot(t1, 'k', label='\_matvec')
plt.plot(t2, 'r', label='matvec')
plt.plot(t3, 'g', label='@')
plt.plot(t4, 'b', label='*')
plt.axis('tight')
plt.legend();

Similarly we now consider the adjoint mode. This can be done in three different ways:

- \_rmatvec: directly applies the method implemented for adjoint mode
- rmatvec: performs some checks before and after applying \_rmatvec
- .H*: first applies the adjoint .H which creates a new scipy.sparse.linalg._CustomLinearOperator' where \_matvec and \_rmatvec are swapped and then applies the new \_matvec.

Once again, after timing these 3 different executions we can see how using \_rmatvec (or rmatvec) will result in the faster computation while .H* is very inefficient and slow. Note that if the adjoint has to be applied multiple times it is at least advised to create the adjoint operator by applying .H only once upfront. Not surprisingly, the linear solvers in scipy as well as in PyLops actually use matvec and rmatvec when dealing with linear operators.

# \_rmatvec
cmd1 = 'Dop._rmatvec(x)'

# rmatvec
cmd2 = 'Dop.rmatvec(x)'

# .H* (pre-computed H)
cmd3 = 'DopH*x'

# .H*
cmd4 = 'Dop.H*x'

# timing
t1 = 1.e3 * np.array(timeit.repeat(cmd1, setup=cmd_setup,
                                 number=500, repeat=5))
t2 = 1.e3 * np.array(timeit.repeat(cmd2, setup=cmd_setup,
                                 number=500, repeat=5))
t3 = 1.e3 * np.array(timeit.repeat(cmd3, setup=cmd_setup,
% t4 = 1.e3 * np.array(timeit.repeat(cmd4, setup=cmd_setup,
number=500, repeat=5))
plt.figure(figsize=(7, 2))
plt.plot(t1, 'k', label=' _rmatvec')
plt.plot(t2, 'r', label='rmatvec')
plt.plot(t3, 'g', label='.H* (pre-computed H)')
plt.plot(t4, 'b', label='.H*')
plt.axis('tight')
plt.legend();

Out:
<matplotlib.legend.Legend object at 0x7f745de344a8>

Just to reiterate once again, it is advised to call matvec and rmatvec unless PyLops linear operators are used for teaching purposes.

We now go through some other methods and special methods that are implemented in scipy.sparse.linalg. LinearOperator (and pylops.LinearOperator):

- Op1+Op2: maps the special method __add__ and performs summation between two operators and returns a pylops.LinearOperator
- -Op: maps the special method __neg__ and performs negation of an operators and returns a pylops.LinearOperator
- Op1-Op2: maps the special method __sub__ and performs summation between two operators and returns a pylops.LinearOperator
- Op1**N: maps the special method __pow__ and performs exponentiation of an operator and returns a pylops.LinearOperator
- Op/y (and Op.div(y)): maps the special method __truediv__ and performs inversion of an operator
- Op.eigs(): estimates the eigenvalues of the operator
- Op.cond(): estimates the condition number of the operator
- Op.conj(): create complex conjugate operator

# +
print(Dop + Dop)
To understand the effect of \texttt{conj} we need to look into a problem with an operator in the complex domain. Let’s create again our \texttt{pylops.Diagonal} operator but this time we populate it with complex numbers. We will see that the action of the operator and its complex conjugate is different even if the model is real.

```python
n = 5
d = 1j*(np.arange(n) + 1.)
x = np.ones(n)
Dop = pylops.Diagonal(d)
print('y = Dx = ', Dop*x)
print('y = conj(D)x = ', Dop.conj()*x)
```

```
y = Dx = [0.+1.j 0.+2.j 0.+3.j 0.+4.j 0.+5.j]
y = conj(D)x = [0.-1.j 0.-2.j 0.-3.j 0.-4.j 0.-5.j]
```

At this point, the concept of linear operator may sound abstract. The convinience method \texttt{pylops.LinearOperator.todense} can be used to create the equivalent dense matrix of any operator. In this case for example we expect to see a diagonal matrix with \(d \) values along the main diagonal

```python
D = Dop.todense()
plt.figure(figsize=(5, 5))
plt.imshow(np.abs(D))
```

(continues on next page)
Finally it is worth reiterating that if two linear operators are combined by means of the algebraical operations shown above, the resulting operator is still a `pylops.LinearOperator` operator. This means that we can still apply any of the methods implemented in the original scipy class definition like `*`, as well as those in our class definition like `/

```python
Dop1 = Dop - Dop.conj()
y = Dop1 * x
print('x = (Dop - conj(Dop))/y = ', Dop1 / y)
D1 = Dop1.todense()
plt.figure(figsize=(5, 5))
plt.imshow(np.abs(D1))
plt.title(r'Dense representation of $|D + D^*|$')
```

Out:

<matplotlib.colorbar.Colorbar object at 0x7f745d86df60>

(continues on next page)
plt.axis('tight')
plt.colorbar()

Dense representation of $|D + D^*|$

Out:

```
x = (Dop - conj(Dop))/y = [1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j]
<matplotlib.colorbar.Colorbar object at 0x7f745e1d48d0>
```

This first tutorial is completed. You have seen the basic operations that can be performed using `scipy.sparse.linalg.LinearOperator` and our overload of such a class `pylops.LinearOperator` and you should be able to get started combining various PyLops operators and solving your own inverse problems.

**Total running time of the script:** (0 minutes 0.723 seconds)

### 3.4.2 02. The Dot-Test

One of the most important aspect of writing a `Linear operator` is to be able to verify that the code implemented in `forward mode` and the code implemented in `adjoint mode` are effectively adjoint to each other. If this is the case, your `Linear operator` will successfully pass the so-called `dot-test`. Refer to the `Notes` section of `pylops.utils.dottest` for a more detailed description.
In this example, I will show you how to use the dot-test for a variety of operator when model and data are either real or complex numbers.

```python
# pylint: disable=C0103
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.gridspec as pltgs
import pylops
from pylops.utils import dottest
plt.close('all')

Let's start with something very simple. We will make a `pylops.MatrixMult` operator and verify that its implementation passes the dot-test. For this time, we will do this step-by-step, replicating what happens in the `pylops.utils.dottest` routine.

```python
N, M = 5, 3
Mat = np.arange(N*M).reshape(N, M)
Op = pylops.MatrixMult(Mat)

v = np.random.randn(N)
u = np.random.randn(M)

# Op * u
y = Op.matvec(u)
# Op'* v
x = Op.rmatvec(v)

yy = np.dot(y, v)  # (Op * u)' * v
xx = np.dot(u, x)  # u' * (Op' * v)

print('Dot-test %e' % np.abs((yy - xx) / ((yy + xx + 1e-15) / 2)))
```

Out:

```
Dot-test 1.366060e-16
```

And here is a visual interpretation of what a dot-test is.

```python
gs = pltgs.GridSpec(1, 9)
fig = plt.figure(figsize=(7, 3))
ax = plt.subplot(gs[0, 0:2])
ax.imshow(Op.A, cmap='rainbow')
ax.set_title(r'$(Op*$', size=20, fontweight='bold')
ax.set_xticks(np.arange(M-1)+0.5)
ax.set_yticks(np.arange(N-1)+0.5)
ax.grid(linewidth=3, color='white')
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
ax.axis('tight')

ax = plt.subplot(gs[0, 2])
ax.imshow(u[:, np.newaxis], cmap='rainbow')
ax.set_title(r'$u)^T$', size=20, fontweight='bold')
ax.set_xticks([])
ax.set_yticks(np.arange(M-1)+0.5)
ax.grid(linewidth=3, color='white')
```

(continues on next page)
ax.yaxis.set_ticklabels([])
ax.axis('tight')
ax = plt.subplot(gs[0, 3])
ax.imshow(v[:, np.newaxis], cmap='rainbow')
ax.set_title(r'$v$', size=20, fontweight='bold')
ax.set_xticks([])
ax.set_yticks(np.arange(N-1)+0.5)
ax.grid(linewidth=3, color='white')
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
ax = plt.subplot(gs[0, 4])
ax.text(0.35, 0.5, '=', horizontalalignment='center',
       verticalalignment='center', size=40, fontweight='bold')
ax.axis('off')
ax = plt.subplot(gs[0, 5])
ax.imshow(u[:, np.newaxis].T, cmap='rainbow')
ax.set_title(r'$u^T$', size=20, fontweight='bold')
ax.set_xticks(np.arange(M-1)+0.5)
ax.set_yticks([])
ax.grid(linewidth=3, color='white')
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
ax = plt.subplot(gs[0, 6:8])
ax.imshow(Op.A.T, cmap='rainbow')
ax.set_title(r'$(Op^T*$', size=20, fontweight='bold')
ax.set_xticks(np.arange(N-1)+0.5)
ax.set_yticks(np.arange(M-1)+0.5)
ax.grid(linewidth=3, color='white')
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
ax.axis('tight')
ax = plt.subplot(gs[0, 8])
ax.imshow(v[:, np.newaxis], cmap='rainbow')
ax.set_title(r'$v)$', size=20, fontweight='bold')
ax.set_xticks([])
ax.set_yticks(np.arange(N-1)+0.5)
ax.grid(linewidth=3, color='white')
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
From now on, we can simply use the `pylops.utils.dottest` implementation of the dot-test and pass the operator we would like to validate, its size in the model and data spaces and optionally the tolerance we will be accepting for the dot-test to be considered succesfull. Finally we need to specify if our data or/and model vectors contain complex numbers using the `complexflag` parameter. While the dot-test will return `True` when succesfull and `False` otherwise, we can also ask to print its outcome putting the `verb` parameters to `True`.

```
N = 10
d = np.arange(N)
Dop = pylops.Diagonal(d)
dottest(Dop, N, N, tol=1e-6, complexflag=0, verb=True)
```

```
Out:
Dot test passed, v^T(Opv) = -6.919654 - u^T(Op^Tv) = -6.919654
True
```

We move now to a more complicated operator, the `pylops.signalprocessing.FFT` operator. We use once again the `pylops.utils.dottest` to verify its implementation and since we are dealing with a transform that can be applied to both real and complex array, we try different combinations using the `complexflag` input.

```
dt = 0.005
nt = 100
nfft = 2**10
FFTop = pylops.signalprocessing.FFT(dims=(nt,), nfft=nfft,
sampling=dt, dtype=np.complex128)
dottest(FFTop, nfft, nt, complexflag=2, verb=True)
dottest(FFTop, nfft, nt, complexflag=3, verb=True)
```

```
Out:
```

3.4. Tutorials
Dot test passed, $v^T(Opu) = -17.139104 + 3.828834i$ - $u^T(Op^Tv) = -17.139104 + 3.828834i$

Dot test passed, $v^T(Opu) = 11.420800 - 3.688151i$ - $u^T(Op^Tv) = 11.420800 - 3.688151i$

Total running time of the script: (0 minutes 0.295 seconds)

### 3.4.3 03. Solvers

This tutorial will guide you through the `pylops.optimization` module and show how to use various solvers that are included in the PyLops library.

The main idea here is to provide the user of PyLops with very high-level functionalities to quickly and easily set up and solve complex systems of linear equations as well as include regularization and/or preconditioning terms (all of those constructed by means of PyLops linear operators).

To make this tutorial more interesting, we will present a real life problem and show how the choice of the solver and regularization/preconditioning terms is vital in many circumstances to successfully retrieve an estimate of the model.

The problem that we are going to consider is generally referred to as the data reconstruction problem and aims at reconstructing a regularly sampled signal of size $M$ from $N$ randomly selected samples:

$$y = Rx$$

where the restriction operator $R$ that selects the $M$ elements from $x$ at random locations is implemented using `pylops.Restriction`, and

$$y = [y_1, y_2, \ldots, y_N]^T, \quad x = [x_1, x_2, \ldots, x_M]^T,$$

with $M >> N$.

```python
# pylint: disable=C0103
import numpy as np
import matplotlib.pyplot as plt
import pylops
plt.close('all')
np.random.seed(10)
Let's first create the data in the frequency domain. The data is composed by the superposition of 3 sinusoids with different frequencies.

```
```python
fig, axs = plt.subplots(2, 1, figsize=(12, 8))
axs[0].plot(f, np.abs(X), 'k', LineWidth=2)
axs[0].set_xlim(0, 30)
axs[0].set_title('Data (frequency domain)')
axs[1].plot(t, x, 'k', LineWidth=2)
axs[1].set_title('Data (time domain)')
axs[1].axis('tight')
```
We now define the locations at which the signal will be sampled.

```python
# subsampling locations
perc_subsampling = 0.2
Nsub = int(np.round(N*perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(N))[:Nsub])

# Create restriction operator
Rop = pylops.Restriction(N, iava, dtype='float64')

y = Rop*x
ymask = Rop.mask(x)

# Visualize data
fig = plt.figure(figsize=(12, 4))
plt.plot(t, x, 'k', lw=3)
plt.plot(t, ymask, '.g', ms=15, label='available samples')
plt.legend()
plt.title('Data restriction')
```

To start let’s consider the simplest ‘solver’, i.e., *least-square inversion without regularization*. We aim here to minimize the following cost function:
\[ J = ||y - Rx||_2^2 \]

Depending on the choice of the operator \( R \), such problem can be solved using explicit matrix solvers as well as iterative solvers. In this case we will be using the latter approach (more specifically the scipy implementation of the \texttt{LSQR} solver - i.e., \texttt{scipy.sparse.linalg.lsqr}) as we do not want to explicitly create and invert a matrix. In most cases this will be the only viable approach as most of the large-scale optimization problems that we are interested to solve using PyLops do not lend naturally to the creation and inversion of explicit matrices.

This first solver can be very easily implemented using the \( / \) for PyLops operators, which will automatically call the \texttt{scipy.sparse.linalg.lsqr} with some default parameters.

```python
xinv = Rop / y
```

We can also use \texttt{pylops.optimization.leastsquares.RegularizedInversion} (without regularization term for now) and customize our solvers using \texttt{kwargs}.

```python
xinv = \
    pylops.optimization.leastsquares.RegularizedInversion(Rop, [], y, 
     **dict(damp=0, 
     iter_lim=10, 
     show=1))
```

Out:

```
LSQR       Least-squares solution of  Ax = b
The matrix A has 40 rows and 200 columns
  damp = 0.0000000000000e+00  calc_var = 0
  atol = 1.00e-08     conlim = 1.00e+08
  btol = 1.00e-08     iter_lim = 10

          Itn   x[0]   r1norm   r2norm  Compatible LS  Norm A  Cond A
        0  0.00000e+00  3.759e+00  3.759e+00    1.0e+00       2.7e-01
        1  0.00000e+00  0.000e+00  0.000e+00    0.0e+00       0.0e+00

  LSQR finished
  Ax - b is small enough, given atol, btol
  istop =  1     r1norm = 0.0e+00      anorm = 0.0e+00      arnorm = 0.0e+00
  itn   =  1     r2norm = 0.0e+00      acond = 0.0e+00      xnorm = 3.8e+00
```

Finally we can select a different starting guess from the null vector

```python
xinv_fromx0 = \
    pylops.optimization.leastsquares.RegularizedInversion(Rop, [], y, 
     x0=np.ones(N), 
     **dict(damp=0, 
     iter_lim=10, 
     show=0))
```

The cost function above can be also expanded in terms of its \textit{normal equations}

\[ x_{ne} = (R^T R)^{-1} R^T y \]
The method `pylops.optimization.leastsquares.NormalEquationsInversion` implements such system of equations explicitly and solves them using an iterative scheme suitable for square matrices (i.e., $M = N$).

While this approach may seem not very useful, we will soon see how regularization terms could be easily added to the normal equations using this method.

```python
xne = pylops.optimization.leastsquares.NormalEquationsInversion(Rop, [], y)
```

Let's now visualize the different inversion results

```python
fig = plt.figure(figsize=(12, 4))
plt.plot(t, x, 'k', lw=2, label='original')
plt.plot(t, xinv, 'b', ms=10, label='inversion')
plt.plot(t, xinv_fromx0, '--r', ms=10, label='inversion from x0')
plt.plot(t, xne, '--g', ms=10, label='normal equations')
plt.legend()
plt.title('Data reconstruction without regularization')
```

Out:

```
Text(0.5, 1.0, 'Data reconstruction without regularization')
```

### Regularization

You may have noticed that none of the inversion has been successful in recovering the original signal. This is a clear indication that the problem we are trying to solve is highly ill-posed and requires some prior knowledge from the user.

We will now see how to add prior information to the inverse process in the form of regularization (or preconditioning). This can be done in two different ways
Let’s start by regularizing the normal equations using a second derivative operator

\[
x = (R^T R + \epsilon \nabla^T \nabla)^{-1} R^T y
\]

```python
# Create regularization operator
D2op = pylops.SecondDerivative(N, dims=None, dtype='float64')

# Regularized inversion
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))
```

Note that in case we have access to a fast implementation for the chain of forward and adjoint for the regularization operator (i.e., \(\nabla^T \nabla\)), we can modify our call to `pylops.optimization.leastsquares.NormalEquationsInversion` as follows:

```python
ND2op = pylops.MatrixMult((D2op.H * D2op).tosparse()) # mimic fast D^T D

xne1 = 
    pylops.optimization.leastsquares.NormalEquationsInversion(Rop, [], y,
    NRegs=[ND2op],
    epsI=epsI,
    epsNRs=[epsR],
    returninfo=False,
    **dict(maxiter=50))
```

We can do the same while using `pylops.optimization.leastsquares.RegularizedInversion` which solves the following augmented problem

\[
\begin{bmatrix}
R \\
\epsilon \nabla
\end{bmatrix} x = \begin{bmatrix} y \\ 0 \end{bmatrix}
\]

```python
xreg = 
    pylops.optimization.leastsquares.RegularizedInversion(Rop, [D2op], y,
    epsRs=[np.sqrt(0.1)],
    returninfo=False,
    **dict(damp=np.sqrt(1e-4),
    iter_lim=50,
    show=0))
```

We can also write a preconditioned problem, whose cost function is

\[
J = ||y - RPp||_2^2
\]
where \( P \) is the preconditioned operator, \( p \) is the projected model in the preconditioned space, and \( x = Pp \) is the model in the original model space we want to solve for. Note that a preconditioned problem converges much faster to its solution than its corresponding regularized problem. This can be done using the routine `pylops.optimization.leastsquares.PreconditionedInversion`.

```python
# Create regularization operator
Sop = pylops.Smoothing1D(nsmooth=11, dims=[N], dtype='float64')

# Invert for interpolated signal
xprec = \
    pylops.optimization.leastsquares.PreconditionedInversion(Rop, Sop, y,
    returninfo=False,
    **dict(damp=np.sqrt(1e-
    →9),

Let’s finally visualize these solutions

```
Much better estimates! We have seen here how regularization and/or preconditioning can be vital to successfully solve some ill-posed inverse problems.

We have however so far only considered solvers that can include additional norm-2 regularization terms. A very active area of research is that of sparsity-promoting solvers (also sometimes referred to as compressive sensing): the regularization term added to the cost function to minimize has norm-\(p\) \((p \leq 1)\) and the problem is generally recast by considering the model to be sparse in some domain. We can follow this philosophy as our signal to invert was actually created as superposition of 3 sinusoids (i.e., three spikes in the Fourier domain). Our new cost function is:

\[J_1 = ||y - RFp||^2_2 + \epsilon||p||_1\]

where \(F\) is the FFT operator. We will thus use the `pylops.optimization.sparsityISTA` and `pylops.optimization.sparsity.FISTA` solvers to estimate our input signal.

```python
pista, niteri, costi = pylops.optimization.sparsityISTA(Rop*FFTop.H, y, niter=1000,
                                    eps=0.1, tol=1e-7, returninfo=True)
xista = FFTop.H*pista

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, axs = plt.subplots(2, 1, figsize=(12, 8))
fig.suptitle('Data reconstruction with sparsity', fontsize=14, fontweight='bold', y=0.9)
axs[0].plot(f, np.abs(X), 'k', lw=3)
axs[0].plot(f, np.abs(pista), '--r', lw=3)
axs[0].plot(f, np.abs(pfista), '--g', lw=3)
axs[0].set_xlim(0, 30)
axs[0].set_title('Frequency domain')
axs[1].plot(t[iava], y, '.k', ms=20, label='available samples')
axs[1].plot(t, x, 'k', lw=3, label='original')
axs[1].plot(t, xista, '--r', lw=3, label='ISTA')
axs[1].plot(t, xfista, '--g', lw=3, label='FISTA')
axs[1].set_title('Time domain')
axs[1].axis('tight')
axs[1].legend()
plt.tight_layout()
plt.subplots_adjust(top=0.8)

fig, ax = plt.subplots(1, 1, figsize=(12, 3))
ax.semilogy(costi, 'r', lw=2, label='ISTA')
ax.semilogy(costf, 'g', lw=2, label='FISTA')
ax.set_title('Cost functions', size=15, fontweight='bold')
ax.set_xlabel('Iteration')
ax.legend()
ax.grid(True)
plt.tight_layout()
Data reconstruction with PyLops.

Frequency domain

Time domain
As you can see, changing parametrization of the model and imposing sparsity in the Fourier domain has given an extra improvement to our ability of recovering the underlying densely sampled input signal. Moreover, FISTA converges much faster than ISTA as expected and should be preferred when using sparse solvers.

Finally we consider a slightly different cost function (note that in this case we try to solve a constrained problem):

\[
J_1 = \|p\|_1 \quad \text{subj.to} \quad \|y - RFp\|
\]

A very popular solver to solve such kind of cost function is called spgl1 and can be accessed via pylops.optimization.sparsity.SPGL1.

```python
xspgl1, pspgl1, info = 
    pylops.optimization.sparsity.SPGL1(Rop, y, FFTop, tau=3, iter_lim=200)
```

---

3.4. Tutorials

(continues on next page)
Data reconstruction with PyLops

Frequency domain

Time domain
This tutorial focuses on Bayesian inversion, a special type of inverse problem that aims at incorporating prior information in terms of model and data probabilities in the inversion process.

In this case we will be dealing with the same problem that we discussed in 03. Solvers, but instead of defining ad-hoc regularization or preconditioning terms we parametrize and model our input signal in the frequency domain in a probabilistic fashion: the central frequency, amplitude and phase of the three sinusoids have Gaussian distributions as follows:

\[
X(f) = \sum_{i=1}^{3} a_i e^{j\phi_i} \delta(f - f_i)
\]

where \( f_i \sim N(f_{0,i}, \sigma_{f,i}) \), \( a_i \sim N(a_{0,i}, \sigma_{a,i}) \), and \( \phi_i \sim N(\phi_{0,i}, \sigma_{\phi,i}) \).

Based on the above definition, we construct some prior models in the frequency domain, convert each of them to the time domain and use such an ensemble to estimate the prior mean \( \mu_x \) and model covariance \( C_x \).

We then create our data by sampling the true signal at certain locations and solve the reconstruction problem within a Bayesian framework. Since we are assuming Gaussianity in our priors, the equation to obtain the posterior mean can be derived analytically:

\[
x = x_0 + C_x R^T (R C_x R^T + C_y)^{-1} (y - R x_0)
\]
Let's start by creating our true model and prior realizations

```python
def prior_realization(f0, a0, phi0, sigmaf, sigmaa, sigmaphi, dt, nt, nfft):
    """Create realization from prior mean and std for amplitude, frequency and phase""
    f = np.fft.rfftfreq(nfft, dt)
    df = f[1] - f[0]
    ifreqs = [int(np.random.normal(f, sigma)/df)
        for f, sigma in zip(f0, sigmaf)]
    amps = [np.random.normal(a, sigma) for a, sigma in zip(a0, sigmaa)]
    phis = [np.random.normal(phi, sigma) for phi, sigma in zip(phi0, sigmaphi)]
    # input signal in frequency domain
    X = np.zeros(nfft//2+1, dtype='complex128')
    X[ifreqs] = np.array(amps).squeeze() * \n        np.exp(1j * np.deg2rad(np.array(phis))).squeeze()
    # input signal in time domain
    FFTop = pylops.signalprocessing.FFT(nt, nfft=nfft, real=True)
    x = FFTop.H*X
    return x
```

# Priors
nreals = 100
f0 = [5, 3, 8]
sigmaf = [0.5, 1., 0.6]
a0 = [1., 1., 1.]
sigmaa = [0.1, 0.5, 0.6]
phi0 = [-90., 0., 0.]
sigmaphi = [0.1, 0.2, 0.4]
sigmad = 1e-2

# Prior models
nt = 200
nfft = 2**11
dt = 0.004
t = np.arange(nt)*dt
xs = np.array([prior_realization(f0, a0, phi0, sigmaf, sigmaa, sigmaphi, dt, nt, nfft)
    for _ in range(nreals)])

# True model (taken as one possible realization)
x = prior_realization(f0, a0, phi0, [0, 0, 0], [0, 0, 0],
    [0, 0, 0], dt, nt, nfft)

We have now a set of prior models in time domain. We can easily use sample statistics to estimate the prior mean and covariance. For the covariance, we perform a second step where we average values around the main diagonal for each row and find a smooth, compact filter that we use to define a convolution linear operator that mimics the action of the covariance matrix on a vector

```python
x0 = np.average(xs, axis=0)
Cm = ((xs - x0).T @ (xs - x0))/nreals

N = 30  # lenght of decorrelation
diags = np.array([Cm[i, i-N:i+N+1] for i in range(N, nt-N)])
diag_ave = np.average(diags, axis=0)
```

(continues on next page)
PyLops

(continued from previous page)

diag_ave *= np.hamming(2*N+1) # add a taper at the end to avoid edge effects
fig, ax = plt.subplots(1, 1, figsize=(12, 4))
ax.plot(t, xs.T, 'r', lw=1)
ax.plot(t, x0, 'g', lw=4)
ax.plot(t, x, 'k', lw=4)
ax.set_title('Prior realizations and mean')
ax.set_xlim(0, 0.8)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
im = ax1.imshow(Cm, interpolation='nearest', cmap='seismic',
extent=(t[0], t[-1], t[-1], t[0]))
ax1.set_title(r"$\mathbf{C}_m^{prior}$")
ax1.axis('tight')
ax2.plot(np.arange(-N, N + 1) * dt, diags.T, '--r', lw=1)
ax2.plot(np.arange(-N, N + 1) * dt, diag_ave, 'k', lw=4)
ax2.set_title("Averaged covariance 'filter'")

•

3.4. Tutorials

37


Let's define now the sampling operator as well as create our covariance matrices in terms of linear operators. This may not be strictly necessary here but shows how even Bayesian-type of inversion can very easily scale to large model and data spaces.

```python
# Sampling operator
perc_subsampling = 0.2
ntsub = int(np.round(nt*perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(nt))[:ntsub])
iava[-1] = nt-1 # assume we have the last sample to avoid instability
Rop = pylops.Restriction(nt, iava, dtype='float64')

# Covariance operators
Cm_op = pylops.signalprocessing.Convolve1D(nt, diag_ave, offset=N)
Cd_op = sigmad**2 * pylops.Identity(ntsub)
```

We model now our data and add noise that respects our prior definition

```python
n = np.random.normal(0, sigmad, nt)
y = Rop * x
yn = Rop * (x + n)
ymask = Rop.mask(x)
ynmask = Rop.mask(x + n)
```

First we apply the Bayesian inversion equation
So far we have been able to estimate our posterior mean. What about its uncertainties (i.e., posterior covariance)?

In real-life applications it is very difficult (if not impossible) to directly compute the posterior covariance matrix. It is much more useful to create a set of models that sample the posterior probability. We can do that by solving our problem several times using different prior realizations as starting guesses:
```python
xpost = [x0 + Cm_op * Rop.H * (lsqr(Rop * Cm_op * Rop.H + Cd_op, yn - Rop * x0, iter_→lim=400)[0])
    for x0 in xs[:30]]
xpost = np.array(xpost)

x0post = np.average(xpost, axis=0)
Cm_post = ((xpost - x0post).T @ (xpost - x0post)) / nreals

# Visualize
fig, ax = plt.subplots(1, 1, figsize=(12, 5))
ax.plot(t, x, 'k', lw=6, label='true')
ax.plot(t, xpost.T, '--r', lw=1)
ax.plot(t, x0post, 'r', lw=3, label='bayesian inverse')
ax.plot(t, ymask, '.k', ms=25, label='available samples')
ax.plot(t, ynmask, '.r', ms=25, label='available noisy samples')
ax.legend()
ax.set_title('Signal')
ax.set_xlim(0, 0.8)

fig, ax = plt.subplots(1, 1, figsize=(5, 4))
im = ax.imshow(Cm_post, interpolation='nearest', cmap='seismic',
              extent=(t[0], t[-1], t[-1], t[0]))
ax.set_title(r'$\mathbf{C}_m^{\text{posterior}}$')
ax.axis('tight')
```

---

Chapter 3. History
Out:

\[(0.0, 0.796, 0.796, 0.0)\]

Note that here we have been able to compute a sample posterior covariance from its estimated samples. By displaying it we can see how both the overall variances and the correlation between different parameters have become narrower compared to their prior counterparts.

**Total running time of the script:** (0 minutes 1.567 seconds)

### 3.4.5 Image deblurring

*Deblurring* is the process of removing blurring effects from images, caused for example by defocus aberration or motion blur.

In forward mode, such blurring effect is typically modelled as a 2-dimensional convolution between the so-called *point spread function* and a target sharp input image, where the sharp input image (which has to be recovered) is unknown and the point-spread function can be either known or unknown.

In this tutorial, an example of 2d blurring and deblurring will be shown using the `pylops.signalprocessing.Convolve2D` operator assuming knowledge of the point-spread function.

```python
import numpy as np
import matplotlib.pyplot as plt
import pylops

Let’s start by importing a 2d image and defining the blurring operator
```
im = np.load('../testdata/python.npy')[::5, ::5, 0]

Nz, Nx = im.shape

# Blurring guassian operator
nh = [15, 25]
hz = np.exp(-0.1*np.linspace(-(nh[0]//2), nh[0]//2, nh[0])**2)
hx = np.exp(-0.03*np.linspace(-(nh[1]//2), nh[1]//2, nh[1])**2)
hz /= np.trapz(hz)  # normalize the integral to 1
hx /= np.trapz(hx)  # normalize the integral to 1
h = hz[:, np.newaxis] * hx[np.newaxis, :]

fig, ax = plt.subplots(1, 1, figsize=(5, 3))
him = ax.imshow(h)
ax.set_title('Blurring operator')
fig.colorbar(him, ax=ax)
ax.axis('tight')

Cop = pylops.signalprocessing.Convolve2D(Nz * Nx, h=h,
                                          offset=(nh[0] // 2, nh[1] // 2),
                                          dims=(Nz, Nx), dtype='float32')

We first apply the blurring operator to the sharp image. We then try to recover the sharp input image by inverting the convolution operator from the blurred image. Note that when we perform inversion without any regularization, the deblurred image will show some ringing due to the instabilities of the inverse process. Using a L1 solver with a DWT preconditioner or TV regularization allows to recover sharper contrasts.

imblur = Cop * im.flatten()

imdeblur = pylops.optimization.leastsquares.NormalEquationsInversion(Cop, None, imblur, maxiter=50)

Wop = pylops.signalprocessing.DWT2D((Nz, Nx), wavelet='haar', level=3)
Dop = [pylops.FirstDerivative(Nz * Nx, dims=(Nz, Nx), dir=0, edge=False),
    pylops.FirstDerivative(Nz * Nx, dims=(Nz, Nx), dir=1, edge=False)]
DWop = Dop + [Wop, ]

imdeblurfista = \n    pylops.optimization.sparsity.FISTA(Cop * Wop.H, imblur, eps=1e-1,
    niter=100)[0]
imdeblurfista = Wop.H * imdeblurfista

imdeblurtv = \n    pylops.optimization.sparsity.SplitBregman(Cop, Dop, imblur.flatten(),
    niter_outer=10, niter_inner=5,
    mu=1.5, epsRL1s=[2e0, 2e0],
    tol=1e-4, tau=1., show=False,
    ** dict(iter_lim=5, damp=1e-4))[0]
imdeblurtv1 = \n    pylops.optimization.sparsity.SplitBregman(Cop, DWop,
    imblur.flatten(),
    niter_outer=10, niter_inner=5,
    mu=1.5, epsRL1s=[1e0, 1e0, 1e0],
    tol=1e-4, tau=1., show=False,
    ** dict(iter_lim=5, damp=1e-4))[0]

Finally we visualize the original, blurred, and recovered images.

fig = plt.figure(fignsize=(12, 6))
fig.suptitle('Deblurring', fontsize=14, fontweight='bold', y=0.95)
ax1 = plt.subplot2grid((2, 5), (0, 0))
ax2 = plt.subplot2grid((2, 5), (0, 1))
ax3 = plt.subplot2grid((2, 5), (0, 2))
ax4 = plt.subplot2grid((2, 5), (0, 3), colspan=2)
ax5 = plt.subplot2grid((2, 5), (1, 0))
ax6 = plt.subplot2grid((2, 5), (1, 1))
ax7 = plt.subplot2grid((2, 5), (1, 2))
ax8 = plt.subplot2grid((2, 5), (1, 3), colspan=2)
ax9 = plt.subplot2grid((2, 5), (1, 4))
ax1.imshow(im, cmap='viridis', vmin=0, vmax=250)
ax1.axis('tight')
ax1.set_title('Original')
ax2.imshow(imblur, cmap='viridis', vmin=0, vmax=250)
ax2.axis('tight')
ax2.set_title('Blurred')
ax3.imshow(imdeblur, cmap='viridis', vmin=0, vmax=250)
ax3.axis('tight')
ax3.set_title('Deblurred')
ax4.imshow(imdeblurfista, cmap='viridis', vmin=0, vmax=250)
ax4.axis('tight')
ax4.set_title('FISTA deblurred')
ax5.imshow(imdeblurtv, cmap='viridis', vmin=0, vmax=250)
ax5.axis('tight')
ax5.set_title('TV deblurred')
ax6.imshow(imdeblurtv1, cmap='viridis', vmin=0, vmax=250)
ax6.axis('tight')
ax6.set_title('TV deblurred')
ax7.imshow(imdeblurtv1, cmap='viridis', vmin=0, vmax=250)
ax7.axis('tight')
ax7.set_title('TV deblurred')
ax8.imshow(imdeblurtv1, cmap='viridis', vmin=0, vmax=250)
ax8.axis('tight')
ax8.set_title('TV deblurred')
ax9.imshow(imdeblurtv1, cmap='viridis', vmin=0, vmax=250)
ax9.axis('tight')
ax9.set_title('TV deblurred')
ax5.axis('tight')
ax5.set_title('TV deblurred')
ax6.imshow(imdeblurtv1, cmap='viridis', vmin=0, vmax=250)
ax6.axis('tight')
ax6.set_title('TV+Haar deblurred')
ax7.plot(im[Nz//2], 'k')
ax7.plot(imblur[Nz//2], '--r')
ax7.plot(imdeblur[Nz//2], '--b')
ax7.plot(imdeblurfista[Nz//2], '--g')
ax7.plot(imdeblurtv[Nz//2], '--m')
ax7.plot(imdeblurtv1[Nz//2], '--y')
ax7.axis('tight')
ax7.set_title('Horizontal section')
ax8.plot(im[:, Nx//2], 'k', label='Original')
ax8.plot(imblur[:, Nx//2], '--r', label='Blurred')
ax8.plot(imdeblur[:, Nx//2], '--b', label='Deblurred')
ax8.plot(imdeblurfista[:, Nx//2], '--g', label='FISTA deblurred')
ax8.plot(imdeblurtv[:, Nx//2], '--m', label='TV deblurred')
ax8.plot(imdeblurtv1[:, Nx//2], '--y', label='TV+Haar deblurred')
ax8.axis('tight')
ax8.set_title('Vertical section')
ax8.legend(loc=5, fontsize='small')
plt.tight_layout()
plt.subplots_adjust(top=0.8)
In the mathematical field of numerical analysis, interpolation is the problem of constructing new data points within the range of a discrete set of known data points. In signal and image processing, the data may be recorded at irregular locations and it is often required to regularize the data into a regular grid.

In this tutorial, an example of 2d interpolation of an image is carried out using a combination of PyLops operators (`pylops.Restriction` and `pylops.Laplacian`) and the `pylops.optimization` module.

Mathematically speaking, if we want to interpolate a signal using the theory of inverse problems, we can define the following forward problem:

\[ y = Rx \]
where the restriction operator $R$ selects $M$ elements from the regularly sampled signal $x$ at random locations. The input and output signals are:

$$y = [y_1, y_2, ..., y_N]^T, \quad x = [x_1, x_2, ..., x_M]^T,$$

with $M \gg N$.

```python
import numpy as np
import matplotlib.pyplot as plt
import pylops

plt.close('all')
np.random.seed(0)

To start we import a 2d image and define our restriction operator to irregularly and randomly sample the image for 30\% of the entire grid

```python
im = np.load('../testdata/python.npy')[:, :, 0]
Nz, Nx = im.shape
N = Nz * Nx

# Subsample signal
perc_subsampling = 0.2
Nsub2d = int(np.round(N*perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(N))[:Nsub2d])

# Create operators and data
Rop = pylops.Restriction(N, iava, dtype='float64')
D2op = pylops.Laplacian((Nz, Nx), weights=(1, 1), dtype='float64')
x = im.flatten()
y = Rop * x
y1 = Rop.mask(x)
```

We will now use two different routines from our optimization toolbox to estimate our original image in the regular grid.

```python
xcg_reg_lop = pylops.optimization.leastsquares.NormalEquationsInversion(Rop, [D2op], y,
                         epsRs=[np.sqrt(0.1)],
                         returninfo=False,
                         **dict(maxiter=200))

# Invert for interpolated signal, lsqrt
xlsqr_reg_lop, istop, itn, r1norm, r2norm = pylops.optimization.leastsquares.RegularizedInversion(Rop, [D2op], y,
                                         epsRs=[np.sqrt(0.1)],
                                         returninfo=True,
                                         **dict(damp=0,
                                         iter_lim=200,
                                         show=0))

# Reshape estimated images
im_sampled = y1.reshape((Nz, Nx))
im_rec_lap_cg = xcg_reg_lop.reshape((Nz, Nx))
im_rec_lap_lsqr = xlsqr_reg_lop.reshape((Nz, Nx))
```
Finally we visualize the original image, the reconstructed images and their error:

```python
def figure_reconstruction(data, method, title):
    fig, axs = plt.subplots(1, 4, figsize=(12, 4))
    fig.suptitle('Data reconstruction - ' + method, fontsize=14, fontweight='bold', y=0.95)
    axs[0].imshow(data, cmap='viridis', vmin=0, vmax=250)
    axs[0].axis('tight')
    axs[0].set_title('Original')
    axs[1].imshow(data_sampled, cmap='viridis', vmin=0, vmax=250)
    axs[1].axis('tight')
    axs[1].set_title('Sampled')
    axs[2].imshow(data_reconstructed, cmap='viridis', vmin=0, vmax=250)
    axs[2].axis('tight')
    axs[2].set_title('Reconstructed')
    axs[3].imshow(data - data_reconstructed, cmap='gray', vmin=-80, vmax=80)
    axs[3].axis('tight')
    axs[3].set_title('Error')
    plt.tight_layout()
    plt.subplots_adjust(top=0.8)
```
Data reconstruction - no regularization

Data reconstruction - regularization

Total running time of the script: 0 minutes 12.434 seconds
3.4.7 07. Post-stack inversion

Estimating subsurface properties from band-limited seismic data represents an important task for geophysical subsurface characterization.

In this tutorial, the `pylops.avo.poststack.PoststackLinearModelling` operator is used for modelling of both 1d and 2d synthetic post-stack seismic data from a profile or 2d model of the subsurface acoustic impedance.

\[
d(t, \theta = 0) = \frac{1}{2} w(t) * \frac{d \ln(AI(t))}{dt}
\]

where \(AI(t)\) is the acoustic impedance profile and \(w(t)\) is the time domain seismic wavelet. In compact form:

\[
d = WDai
\]

where \(W\) is a convolution operator, \(D\) is a first derivative operator, and \(ai\) is the input model. Subsequently the acoustic impedance model is estimated via the `pylops.avo.poststack.PoststackInversion` module. A two-steps inversion strategy is finally presented to deal with the case of noisy data.

```python
# model
nt0 = 301
dt0 = 0.004
t0 = np.arange(nt0)*dt0
vp = 1200 + np.arange(nt0) + \
    filtfilt(np.ones(5)/5., 1, np.random.normal(0, 80, nt0))
rho = 1000 + vp + \
    filtfilt(np.ones(5)/5., 1, np.random.normal(0, 30, nt0))
vp[131:] += 500
rho[131:] += 100
m = np.log(vp*rho)

# smooth model
nsMOOTH = 100
mback = filtfilt(np.ones(nsMOOTH)/float(nsMOOTH), 1, m)

# wavelet
ntwav = 41
wav, twav, wavc = ricker(t0[:ntwav//2+1], 20)

# dense operator
PPop_dense = \
    pylops.avo.poststack.PoststackLinearModelling(wav / 2, nt0=nt0, explicit=True)

# lop operator
```

(continues on next page)
We can now estimate the acoustic profile from band-limited data using either the dense operator or linear operator.

```python
# solve dense
minv_dense = \
    pylops.avo.poststack.PoststackInversion(d, wav / 2, m0=mback, explicit=True,
                                          simultaneous=False)[0]

# solve lop
minv = \
    pylops.avo.poststack.PoststackInversion(d_dense, wav / 2, m0=mback,
                                          explicit=False,
                                          simultaneous=False,
                                          **dict(iter_lim=2000))[0]

# solve noisy
mn = \
    pylops.avo.poststack.PoststackInversion(dn_dense, wav / 2, m0=mback,
                                          explicit=True,
                                          epsR=1e0, **dict(damp=1e-1))[0]

fig, axs = plt.subplots(1, 2, figsize=(6, 7), sharey=True)
axs[0].plot(d_dense, t0, 'k', lw=4, label='Dense')
axs[0].plot(d, t0, '--r', lw=2, label='Lop')
axs[0].plot(dn_dense, t0, '-.g', lw=2, label='Noisy')
axs[0].set_title('Data')
axs[0].invert_yaxis()
axs[0].axis('tight')
axs[0].legend(loc=1)

axs[1].plot(m, t0, 'k', lw=4, label='True')
axs[1].plot(mback, t0, '--b', lw=4, label='Back')
axs[1].plot(minv_dense, t0, '--m', lw=2, label='Inv Dense')
axs[1].plot(minv, t0, '--r', lw=2, label='Inv Lop')
axs[1].plot(mn, t0, '--g', lw=2, label='Inv Noisy')
axs[1].set_title('Model')
axs[1].axis('tight')
axs[1].legend(loc=1)
```

3.4. Tutorials
We see how inverting a dense matrix is in this case faster than solving for the linear operator (a good estimate of the model is in fact obtained only after 2000 iterations of lsqr). Nevertheless, having a linear operator is useful when we deal with larger dimensions (2d or 3d) and we want to couple our modelling operator with different types of spatial regularizations or preconditioning.

Before we move onto a 2d example, let’s consider the case of non-stationary wavelet and see how we can easily use
the same routines in this case

```python
# wavelet
ntwav = 41
f0s = np.flip(np.arange(nt0) * 0.05 + 3)
wavs = np.array([ricker(t0[:ntwav], f0)[0] for f0 in f0s])
wavc = np.argmax(wavs[0])

plt.figure(figsize=(5, 4))
plt.imshow(wavs.T, cmap='gray', extent=(t0[0], t0[-1], t0[ntwav], -t0[ntwav]))
plt.xlabel('t')
plt.title('Wavelets')
plt.axis('tight')

# operator
PPop = 
    pylops.avo.poststack.PoststackLinearModelling(wavs / 2, nt0=nt0, explicit=True)

# data
d = PPop * m

# solve
minv = 
    pylops.avo.poststack.PoststackInversion(d, wavs / 2, m0=mback, explicit=True, **dict(cond=1e-10))[0]

fig, axs = plt.subplots(1, 2, figsize=(6, 7), sharey=True)
axs[0].plot(d, t0, 'k', lw=4)
axs[0].set_title('Data')
axs[0].invert_yaxis()
axs[0].axis('tight')
axs[1].plot(m, t0, 'k', lw=4, label='True')
axs[1].plot(mback, t0, '--b', lw=4, label='Back')
axs[1].plot(minv, t0, '--r', lw=2, label='Inv')
axs[1].set_title('Model')
axs[1].axis('tight')
axs[1].legend(loc=1)
```
We move now to a 2d example. First of all the model is loaded and data generated.

```python
# model
inputfile = '../testdata/avo/poststack_model.npz'
```

Out:

```
<matplotlib.legend.Legend object at 0x7f745d139f98>
```

We move now to a 2d example. First of all the model is loaded and data generated.
model = np.load(inputfile)
m = np.log(model['model'][::, ::3])
x, z = model['x'][::3]/1000., model['z']/1000.
x, nz = len(x), len(z)

# smooth model
nsmoothz, nsmootherx = 60, 50
mback = filtfilt(np.ones(nsmoothz)/float(nsmoothz), 1, m, axis=0)
mback = filtfilt(np.ones(nsmoothx)/float(nsmoothx), 1, mback, axis=1)

# dense operator
PPop_dense = pylops.avo.poststack.PoststackLinearModelling(wav / 2, nt0=nz, spatdims=nx, explicit=True)

# lop operator
PPop = pylops.avo.poststack.PoststackLinearModelling(wav / 2, nt0=nz, spatdims=nx)

# data
d = (PPop_dense * m.flatten()).reshape(nz, nx)
n = np.random.normal(0, 1e-1, d.shape)
dn = d + n

Finally we perform 4 different inversions:

- trace-by-trace inversion with explicit solver and dense operator with noise-free data
- trace-by-trace inversion with explicit solver and dense operator with noisy data
- multi-trace regularized inversion with iterative solver and linear operator using the result of trace-by-trace inversion as starting guess

\[
J = ||\Delta d - W\Delta ai||_2 + \epsilon^2 ||\nabla ai||_2
\]

where \( \Delta d = d - \text{WAI}_0 \) is the residual data
- multi-trace blocky inversion with iterative solver and linear operator

# dense inversion with noise-free data
minv_dense = \n    pylops.avo.poststack.PoststackInversion(d, wav / 2, m0=mback, explicit=True, simultaneous=False)[0]

# dense inversion with noisy data
minv_dense_noisy = \n    pylops.avo.poststack.PoststackInversion(dn, wav / 2, m0=mback, explicit=True, epsI=4e-2, simultaneous=False)[0]

# spatially regularized lop inversion with noisy data
minv_lop_reg = \n    pylops.avo.poststack.PoststackInversion(dn, wav / 2, m0=minv_dense_noisy, explicit=False, epsR=5e1, **dict(damp=np.sqrt(1e-4), iter_lim=80)))[0]

(continues on next page)
# blockiness promoting inversion with noisy data

```
minv_lop_blocky = \
    pylops.avostack.PoststackInversion(dn, wav / 2, m0=mback,
    explicit=False,
    epsR=[0.4], epsRL1=[0.1],
    **dict(mu=0.1,
        niter_outer=5,
        niter_inner=10,
        iter_lim=5, damp=1e-3))[0]
```

```python
fig, axs = plt.subplots(2, 4, figsize=(15, 9))
axs[0][0].imshow(d, cmap='gray',
    extent=(x[0], x[-1], z[-1], z[0]),
    vmin=-0.4, vmax=0.4)
axs[0][0].set_title('Data')
axs[0][0].axis('tight')
axs[0][1].imshow(dn, cmap='gray',
    extent=(x[0], x[-1], z[-1], z[0]),
    vmin=-0.4, vmax=0.4)
axs[0][1].set_title('Noisy Data')
axs[0][1].axis('tight')
axs[0][2].imshow(m, cmap='gist_rainbow',
    extent=(x[0], x[-1], z[-1], z[0]),
    vmin=m.min(), vmax=m.max())
axs[0][2].set_title('Model')
axs[0][2].axis('tight')
axs[0][3].imshow(mback, cmap='gist_rainbow',
    extent=(x[0], x[-1], z[-1], z[0]),
    vmin=m.min(), vmax=m.max())
axs[0][3].set_title('Smooth Model')
axs[0][3].axis('tight')
axs[1][0].imshow(minv_dense, cmap='gist_rainbow',
    extent=(x[0], x[-1], z[-1], z[0]),
    vmin=m.min(), vmax=m.max())
axs[1][0].set_title('Noise-free Inversion')
axs[1][0].axis('tight')
axs[1][1].imshow(minv_dense_noisy, cmap='gist_rainbow',
    extent=(x[0], x[-1], z[-1], z[0]),
    vmin=m.min(), vmax=m.max())
axs[1][1].set_title('Trace-by-trace Noisy Inversion')
axs[1][1].axis('tight')
axs[1][2].imshow(minv_lop_reg, cmap='gist_rainbow',
    extent=(x[0], x[-1], z[-1], z[0]),
    vmin=m.min(), vmax=m.max())
axs[1][2].set_title('Regularized Noisy Inversion - lop')
axs[1][2].axis('tight')
axs[1][3].imshow(minv_lop_blocky, cmap='gist_rainbow',
    extent=(x[0], x[-1], z[-1], z[0]),
    vmin=m.min(), vmax=m.max())
axs[1][3].set_title('Blocky Noisy Inversion - lop')
axs[1][3].axis('tight')
```

```
fig, ax = plt.subplots(1, 1, figsize=(3, 7))
ax.plot(m[:, nx//2], z, 'k', lw=4, label='True')
ax.plot(mback[:, nx//2], z, '--r', lw=4, label='Back')
ax.plot(minv.Dense[:, nx//2], z, '--b', lw=2, label='Inv Dense')
ax.plot(minv.Dense_noisy[:, nx//2], z, '--m', lw=2, label='Inv Dense noisy')
```

(continues on next page)
ax.plot(minv_lop_reg[:, nx//2], z, '--g', lw=2, label='Inv Lop regularized')
ax.plot(minv_lop_blocky[:, nx//2], z, '--y', lw=2, label='Inv Lop blocky')
ax.set_title('Model')
ax.invert_yaxis()
ax.axis('tight')
ax.legend()
plt.tight_layout()
That’s almost it. If you wonder how this can be applied to real data, head over to the following notebook where the open-source segyio library is used alongside pylops to create an end-to-end open-source seismic inversion workflow with SEG-Y input data.

**Total running time of the script:** ( 0 minutes 13.377 seconds)
Pre-stack inversion represents one step beyond post-stack inversion in that not only the profile of acoustic impedance can be inferred from seismic data, rather a set of elastic parameters is estimated from pre-stack data (i.e., angle gathers) using the information contained in the so-called AVO (amplitude versus offset) response. Such elastic parameters represent vital information for more sophisticated geophysical subsurface characterization than it would be possible to achieve working with post-stack seismic data.

In this tutorial, the `pylops.avo.prestack.PrestackLinearModelling` operator is used for modelling of both 1d and 2d synthetic pre-stack seismic data using 1d profiles or 2d models of different subsurface elastic parameters (P-wave velocity, S-wave velocity, and density) as input.

\[
d(t, \theta) = w(t) * \sum_{i=1}^{N} G_i(t, \theta) \frac{d \ln(m_i(t))}{dt}
\]

where \( m(t) = [V_P(t), V_S(t), \rho(t)] \) is a vector containing three elastic parameters at time \( t \), \( G_i(t, \theta) \) are the coefficients of the AVO parametrization used to model pre-stack data and \( w(t) \) is the time domain seismic wavelet. In compact form:

\[
d = WGDm
\]

where \( W \) is a convolution operator, \( G \) is the AVO modelling operator, \( D \) is a block-diagonal derivative operator, and \( m \) is the input model. Subsequently the elastic parameters are estimated via the `pylops.avo.prestack.PrestackInversion` module. Once again, a two-steps inversion strategy can also be used to deal with the case of noisy data.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import filtfilt

import pylops
from pylops.utils.wavelets import ricker

plt.close('all')
np.random.seed(0)

Let’s start with a 1d example. A synthetic profile of acoustic impedance is created and data is modelled using both the dense and linear operator version of `pylops.avo.prestack.PrestackLinearModelling` operator.

```
m = np.stack((np.log(vp), np.log(vs), np.log(rho)), axis=1)

# background model
nsMOOTH = 50
mback = filtfilt(np.ones(nsMOOTH)/float(nsMOOTH), 1, m, axis=0)

# angles
ntheta = 21
thetamin, thetamax = 0, 40
theta = np.linspace(thetamin, thetamax, ntheta)

# wavelet
ntwav = 41
wav = ricker(t0[:ntwav//2+1], 15)[0]

# lop
PPop = 
    pylops.avo.prestack.PrestackLinearModelling(wav, theta,
        vsvp=vsvp, nt0=nt0,
        linearization='akirich')

# dense
PPop_dense = 
    pylops.avo.prestack.PrestackLinearModelling(wav, theta,
        vsvp=vsvp, nt0=nt0,
        linearization='akirich',
        explicit=True)

# data lop
dPP = PPop*m.flatten()
dPP = dPP.reshape(nt0, ntheta)

# data dense
dPP_dense = PPop_dense*m.T.flatten()
dPP_dense = dPP_dense.reshape(ntheta, nt0).T

# noisy data
dPPn_dense = dPP_dense + np.random.normal(0, 1e-2, dPP_dense.shape)

We can now invert our data and retrieve elastic profiles for both noise-free and noisy data using pylops.avo.prestack.PrestackInversion.

# dense
minv_dense, dPP_dense_res = 
    pylops.avo.prestack.PrestackInversion(dPP_dense, theta, wav, m0=mback,
        linearization='akirich',
        explicit=True, returnres=True,
        **dict(cond=1e-10))

# lop
minv, dPP_res = 
    pylops.avo.prestack.PrestackInversion(dPP, theta, wav, m0=mback,
        linearization='akirich',
        explicit=False, returnres=True,
        **dict(damp=1e-10, iter_lim=2000))
The data, inverted models and residuals are now displayed.

```python
# Data and model
fig, (axd, axdn, axvp, axvs, axrho) = plt.subplots(1, 5, figsize=(8, 5), sharey=True)
axd.imshow(dPP_dense, cmap='gray',
           extent=(theta[0], theta[-1], t0[-1], t0[0]),
           vmin=-np.abs(dPP_dense).max(), vmax=np.abs(dPP_dense).max())
axd.set_title('Data')
axd.axis('tight')
axdn.imshow(dPPn_dense, cmap='gray',
            extent=(theta[0], theta[-1], t0[-1], t0[0]),
            vmin=-np.abs(dPP_dense).max(), vmax=np.abs(dPP_dense).max())
axdn.set_title('Noisy Data')
axdn.axis('tight')
axvp.plot(vp, t0, 'k', lw=4, label='True')
axvp.plot(np.exp(mback[:, 0]), t0, '--r', lw=4, label='Back')
axvp.plot(np.exp(minv_dense[:, 0]), t0, '--b', lw=2, label='Inv Dense')
axvp.plot(np.exp(minv[:, 0]), t0, '--m', lw=2, label='Inv Lop')
axvp.plot(np.exp(minv_dense_noise[:, 0]), t0, '--c', lw=2, label='Noisy Dense')
axvp.plot(np.exp(minv_noise[:, 0]), t0, '--g', lw=2, label='Noisy Lop')
axvp.set_title(r'$V_P$')
axvs.plot(vs, t0, 'k', lw=4, label='True')
axvs.plot(np.exp(mback[:, 1]), t0, '--r', lw=4, label='Back')
axvs.plot(np.exp(minv_dense[:, 1]), t0, '--b', lw=2, label='Inv Dense')
axvs.plot(np.exp(minv[:, 1]), t0, '--m', lw=2, label='Inv Lop')
axvs.plot(np.exp(minv_dense_noise[:, 1]), t0, '--c', lw=2, label='Noisy Dense')
axvs.plot(np.exp(minv_noise[:, 1]), t0, '--g', lw=2, label='Noisy Lop')
axvs.set_title(r'$V_S$')
axrho.plot(rho, t0, 'k', lw=4, label='True')
axrho.plot(np.exp(mback[:, 2]), t0, '--r', lw=4, label='Back')
axrho.plot(np.exp(minv_dense[:, 2]), t0, '--b', lw=2, label='Inv Dense')
axrho.plot(np.exp(minv[:, 2]), t0, '--m', lw=2, label='Inv Lop')
axrho.plot(np.exp(minv_dense_noise[:, 2]), t0, '--c', lw=2, label='Noisy Dense')
axrho.plot(np.exp(minv_noise[:, 2]), t0, '--g', lw=2, label='Noisy Lop')
axrho.set_title(r'$\rho$')
axrho.legend(loc='center left', bbox_to_anchor=(1, 0.5))
axd.axis('tight')
plt.tight_layout()
```

# Residuals

```python
# Residuals
```

(continues on next page)
fig, axs = plt.subplots(1, 4, figsize=(8, 5), sharey=True)
fig.suptitle('Residuals', fontsize=14, fontweight='bold', y=0.95)
im = axs[0].imshow(dPP_dense_res, cmap='gray',
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=-0.1, vmax=0.1)
axs[0].set_title('Dense')
axs[0].set_xlabel(r'$\theta$')
axs[0].set_ylabel('t[s]')
axs[0].axis('tight')
axs[1].imshow(dPP_res, cmap='gray',
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=-0.1, vmax=0.1)
axs[1].set_title('Lop')
axs[1].set_xlabel(r'$\theta$')
axs[1].axis('tight')
axs[2].imshow(dPPn_dense_res, cmap='gray',
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=-0.1, vmax=0.1)
axs[2].set_title('Noisy Dense')
axs[2].set_xlabel(r'$\theta$')
axs[2].axis('tight')
axs[3].imshow(dPPn_res, cmap='gray',
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=-0.1, vmax=0.1)
axs[3].set_title('Noisy Lop')
axs[3].set_xlabel(r'$\theta$')
axs[3].axis('tight')
plt.tight_layout()
plt.subplots_adjust(top=0.85)
We move now to a 2d example. First of all the model is loaded and data generated.

```python
# model
inputfile = '../testdata/avo/poststack_model.npz'

model = np.load(inputfile)
x, z = model['x'][:6]/1000., model['z'][:300]/1000.
nx, nz = len(x), len(z)
m = 1000*model['model'][:300, ::6]

mvp = m.copy()
mvs = m/2
mrho = m/3+300
m = np.log(np.stack((mvp, mvs, mrho), axis=1))

# smooth model
nsMOOTHz, nsMOOTHx = 30, 25
mback = filtfilt(np.ones(nsMOOTHz)/float(nsMOOTHz), 1, m, axis=0)
mbACK = filtfilt(np.ones(nsMOOTHx)/float(nsMOOTHx), 1, mback, axis=2)

# dense operator
PPop_dense = 
    pylops.avo.prestack.PrestackLinearModelling(wav, theta, vsvp=vsvp,
```
Finally we perform the same 4 different inversions as in the post-stack tutorial (see 07. Post-stack inversion for more details).

```python
# dense inversion with noise-free data
minv_dense = \n    pylops.avo.prestack.PrestackInversion(dPP, theta, wav, m0=mback,
                                        explicit=True,
                                        simultaneous=False)

# dense inversion with noisy data
minv_dense_noisy = \n    pylops.avo.prestack.PrestackInversion(dPPn, theta, wav, m0=mback,
                                          explicit=True, epsI=4e-2,
                                          simultaneous=False)

# spatially regularized lop inversion with noisy data
minv_lop_reg = \n    pylops.avo.prestack.PrestackInversion(dPPn, theta, wav,
                                           m0=minv_dense_noisy,
                                           explicit=False, epsR=1e1,
                                           **dict(damp=np.sqrt(1e-4),
                                                  iter_lim=20))

# blockiness promoting inversion with noisy data
minv_blocky = \n    pylops.avo.prestack.PrestackInversion(dPPn, theta, wav,
                                         m0=mback,
                                         explicit=False,
                                         epsR=0.4, epsRL1=0.1,
                                         **dict(mu=0.1,
                                                niter_outer=3,
                                                niter_inner=3,
                                                iter_lim=5, damp=1e-3))
```

Let's now visualize the inverted elastic parameters for the different scenarios

```python
def plotmodel(axs, m, x, z, vmin, vmax,
              params=('VP', 'VS', 'Rho'),
              cmap='gist_rainbow', title=None):
    """Quick visualization of model
    """
    for ip, param in enumerate(params):
```

(continued on next page)
```python
    # data
    fig = plt.figure(figsize=(8, 9))
    ax1 = plt.subplot2grid((2, 3), (0, 0), colspan=3)
    ax2 = plt.subplot2grid((2, 3), (1, 0))
    ax3 = plt.subplot2grid((2, 3), (1, 1), sharey=ax2)
    ax4 = plt.subplot2grid((2, 3), (1, 2), sharey=ax2)
    ax1.imshow(dPP[:, 0], cmap='gray',
                extent=(x[0], x[-1], z[-1], z[0]),
                vmin=-0.4, vmax=0.4)
    ax1.vlines([x[nx//5], x[nx//2], x[4*nx//5]], ymin=z[0], ymax=z[-1],
               colors='w', linestyles='--')
    ax1.set_xlabel('x [km]'
    ax1.set_ylabel('z [km]')
    ax1.set_title(r'Stack ($\theta$=0)')
    ax1.axis('tight')
    ax2.imshow(dPP[:, :, nx//5], cmap='gray',
                extent=(theta[0], theta[-1], z[-1], z[0]),
                vmin=-0.4, vmax=0.4)
    ax2.set_xlabel(r'$\theta$')
    ax2.set_ylabel('z [km]')
    ax2.set_title(r'Gather (x= %.2f)' % x[nx//5])
    ax2.axis('tight')
    ax3.imshow(dPP[:, :, nx//2], cmap='gray',
                extent=(theta[0], theta[-1], z[-1], z[0]),
                vmin=-0.4, vmax=0.4)
    ax3.set_xlabel(r'$\theta$')
    ax3.set_title(r'Gather (x= %.2f)' % x[nx//2])
    ax3.axis('tight')
    ax4.imshow(dPP[:, :, 4*nx//5], cmap='gray',
                extent=(theta[0], theta[-1], z[-1], z[0]),
                vmin=-0.4, vmax=0.4)
    ax4.set_xlabel(r'$\theta$')
    ax4.set_title(r'Gather (x= %.2f)' % x[4*nx//5])
    ax4.axis('tight')
    plt.setp(ax3.get_yticklabels(), visible=False)
    plt.setp(ax4.get_yticklabels(), visible=False)

    # noisy data
    fig = plt.figure(figsize=(8, 9))
    ax1 = plt.subplot2grid((2, 3), (0, 0), colspan=3)
    ax2 = plt.subplot2grid((2, 3), (1, 0))
    ax3 = plt.subplot2grid((2, 3), (1, 1), sharey=ax2)
    ax4 = plt.subplot2grid((2, 3), (1, 2), sharey=ax2)
    ax1.imshow(dPP[:, 0], cmap='gray',
                extent=(x[0], x[-1], z[-1], z[0]),
                vmin=-0.4, vmax=0.4)
    ax1.vlines([x[nx//5], x[nx//2], x[4*nx//5]], ymin=z[0], ymax=z[-1],
               colors='w', linestyles='--')
    ax1.set_xlabel('x [km]')
```

(continues on next page)
ax1.set_ylabel('z [km]')
ax1.set_title(r'Noisy Stack ($\theta$=0)')
ax1.axis('tight')
ax2.imshow(dPPn[:, :, nx//5], cmap='gray',
            extent=(theta[0], theta[-1], z[-1], z[0]),
            vmin=-0.4, vmax=0.4)
ax2.set_xlabel(r'$\theta$')
ax2.set_ylabel('z [km]')
ax2.set_title(r'Gather (x=%.2f) % x[nx//5])
ax2.axis('tight')
ax3.imshow(dPPn[:, :, nx//2], cmap='gray',
            extent=(theta[0], theta[-1], z[-1], z[0]),
            vmin=-0.4, vmax=0.4)
ax3.set_xlabel(r'$\theta$')
ax3.set_ylabel('z [km]')
ax3.set_title(r'Gather (x=%.2f) % x[nx//2])
ax3.axis('tight')
ax4.imshow(dPPn[:, :, 4*nx//5], cmap='gray',
            extent=(theta[0], theta[-1], z[-1], z[0]),
            vmin=-0.4, vmax=0.4)
ax4.set_xlabel(r'$\theta$')
ax4.set_ylabel('z [km]')
ax4.set_title(r'Gather (x=%.2f) % x[4*nx//5])
ax4.axis('tight')
plt.setp(ax3.get_yticklabels(), visible=False)
plt.setp(ax4.get_yticklabels(), visible=False)

# inverted models
fig, axs = plt.subplots(6, 3, figsize=(8, 19))
fig.suptitle('Model', fontsize=12, fontweight='bold', y=0.95)
plotmodel(axs[0], m, x, z, m.min(), m.max(), title='True')
plotmodel(axs[1], mback, x, z, m.min(), m.max(), title='Back')
plotmodel(axs[2], minv_dense, x, z, m.min(), m.max(), title='Dense')
plotmodel(axs[3], minv_dense_noisy, x, z, m.min(), m.max(), title='Dense noisy')
plotmodel(axs[4], minv_lop_reg, x, z, m.min(), m.max(), title='Lop regularized')
plotmodel(axs[5], minv_blocky, x, z, m.min(), m.max(), title='Lop blocky')
plt.tight_layout()
plt.subplots_adjust(top=0.92)

fig, axs = plt.subplots(1, 3, figsize=(8, 7))
for ip, param in enumerate(['VP', 'VS', 'Rho']):
    axs[ip].plot(m[:, ip, nx//2], z, 'k', lw=4, label='True')
    axs[ip].plot(mback[:, ip, nx//2], z, '--r', lw=4, label='Back')
    axs[ip].plot(minv_dense[:, ip, nx//2], z, '--b', lw=2, label='Inv Dense')
    axs[ip].plot(minv_dense_noisy[:, ip, nx//2], z, '--m', lw=2,
                  label='Inv Dense noisy')
    axs[ip].plot(minv_lop_reg[:, ip, nx//2], z, '--g', lw=2,
                  label='Inv Lop regularized')
    axs[ip].plot(minv_blocky[:, ip, nx//2], z, '--y', lw=2,
                  label='Inv Lop blocky')
    axs[ip].set_title(param)
    axs[ip].invert_yaxis()
axs[2].legend(loc=8, fontsize='small')
Noisy Stack ($\theta=0$)

Gather ($x=1.42$)  Gather ($x=2.41$)  Gather ($x=3.37$)
While the background model $m_0$ has been provided in all the examples so far, it is worth showing that the module `pylopsavo.prestack.PrestackInversion` can also produce so-called relative elastic parameters (i.e., variations from an average medium property) when the background model $m_0$ is not available.

```python
(continues on next page)
```
```python
pylops.avoprestack.PrestackInversion(dPP, theta, wav, m0=None,
    explicit=True,
    simultaneous=False)

fig, axs = plt.subplots(1, 3, figsize=(8, 3))
plotmodel(axs, dminv, x, z, -dminv.max(), dminv.max(),
    cmap='seismic', title='relative')

fig, axs = plt.subplots(1, 3, figsize=(8, 7))
for ip, param in enumerate(['VP', 'VS', 'Rho']):
    axs[ip].plot(dminv[:, ip, nx//2], z, 'k', lw=2)
    axs[ip].set_title(param)
    axs[ip].invert_yaxis()
```

•
3.4.9 09. Multi-Dimensional Deconvolution

This example shows how to set-up and run the `pylops.waveeqprocessing.MDD` inversion using synthetic data.

```python
import warnings
import numpy as np
```

Total running time of the script: ( 0 minutes 32.952 seconds)
Let’s start by creating a set of hyperbolic events to be used as our MDC kernel

```python
# Input parameters
par = {'ox':-150, 'dx':10, 'nx':31,
       'oy':-250, 'dy':10, 'ny':51,
       'ot':0, 'dt':0.004, 'nt':300,
       'f0': 20, 'nfmax': 200}
t0_m = [0.2]
vrms_m = [700.]
amp_m = [1.]
t0_G = [0.2, 0.5, 0.7]
vrms_G = [800., 1200., 1500.]
amp_G = [1., 0.6, 0.5]

# Taper
tap = taper3d(par['nt'], [par['ny'], par['nx']],
              (5, 5), tapertype='hanning')

# Create axis
t, t2, x, y = makeaxis(par)

# Create wavelet
wav = ricker(t[:41], f0=par['f0'])[0]

# Generate model
m, mwav = hyperbolic2d(x, t, t0_m, vrms_m, amp_m, wav)

# Generate operator
G, Gwav = np.zeros((par['ny'], par['nx'], par['nt'])), 
          np.zeros((par['ny'], par['nx'], par['nt'])))
for iy, y0 in enumerate(y):
    G[iy], Gwav[iy] = hyperbolic2d(x-y0, t, t0_G, vrms_G, amp_G, wav)
G, Gwav = G*tap, Gwav*tap

# Add negative part to data and model
m = np.concatenate((np.zeros((par['nx'], par['nt']-1)), m), axis=-1)
mwav = np.concatenate((np.zeros((par['nx'], par['nt']-1)), mwav), axis=-1)
Gwav2 = np.concatenate((np.zeros((par['ny'], par['nx'], par['nt']-1)), Gwav),
                        axis=-1)

# Define MDC linear operator
Gwav_fft = np.fft.rfft(Gwav2, 2*par['nt']-1, axis=-1)
Gwav_fft = Gwav_fft[..., :par['nfmax']]

MDCop = pylops.waveeqprocessing.MDC(Gwav_fft, nt=2 * par['nt']-1, nv=1,
                                     ntt=par['nt']-1, nvy=1)
```

(continues on next page)
Let's display what we have so far: operator, input model, and data

```python
fig, axs = plt.subplots(1, 2, figsize=(8, 6))
axs[0].imshow(Gwav2[int(par['ny'] / 2)].T, aspect='auto',
              interpolation='nearest', cmap='gray',
              vmin=-np.abs(Gwav2.max()), vmax=np.abs(Gwav2.max()),
              extent=(x.min(), x.max(), t2.max(), t2.min()))
axs[0].set_title('G - inline view', fontsize=15)
axs[0].set_xlabel(r'$x_R$')
axs[1].set_ylabel(r'$t$')
axs[1].imshow(Gwav2[:, int(par['nx'] / 2)].T, aspect='auto',
              interpolation='nearest', cmap='gray',
              vmin=-np.abs(Gwav2.max()), vmax=np.abs(Gwav2.max()),
              extent=(y.min(), y.max(), t2.max(), t2.min()))
axs[1].set_title('G - inline view', fontsize=15)
axs[1].set_xlabel(r'$x_S$')
axs[1].set_ylabel(r'$t$')
fig.tight_layout()

fig, axs = plt.subplots(1, 2, figsize=(8, 6))
axs[0].imshow(mwav.T, aspect='auto', interpolation='nearest', cmap='gray',
              vmin=-np.abs(mwav.max()), vmax=np.abs(mwav.max()),
              extent=(x.min(), x.max(), t2.max(), t2.min()))
axs[0].set_title(r'$m$', fontsize=15)
axs[0].set_xlabel(r'$x_R$')
axs[1].set_ylabel(r'$t$')
axs[1].imshow(d.T, aspect='auto', interpolation='nearest', cmap='gray',
              vmin=-np.abs(d.max()), vmax=np.abs(d.max()),
              extent=(x.min(), x.max(), t2.max(), t2.min()))
axs[1].set_title(r'$d$', fontsize=15)
axs[1].set_xlabel(r'$x_S$')
axs[1].set_ylabel(r'$t$')
fig.tight_layout()
```
We are now ready to feed our operator to `pylops.waveeqprocessing.MDD` and invert back for our input model

```python
minv, madj, psfinv, psfadj = 
    pylops.waveeqprocessing.MDD(Gwav, d[:, par['nt'] - 1:],
    dt=par['dt'], dr=par['dx'],
    nfmax=par['nfmax'], wav=wav,
    twosided=True, add_negative=True,
    adjoint=True, psf=True,
    dtype='complex64', dottest=False,
    **dict(damp=1e-4, iter_lim=20, show=0))

fig = plt.figure(figsize=(8, 6))
ax1 = plt.subplot2grid((1, 5), (0, 0), colspan=2)
ax2 = plt.subplot2grid((1, 5), (0, 2), colspan=2)
ax3 = plt.subplot2grid((1, 5), (0, 4))
ax1.imshow(madj.T, aspect='auto', interpolation='nearest', cmap='gray',
(continues on next page)
```
vmin=-np.abs(madj.max()), vmax=np.abs(madj.max()),
exponent=(x.min(), x.max(), t2.max(), t2.min()))
ax1.set_title('Adjoint m', fontsize=15)
ax1.set_xlabel(r'$x_V$')
axs[1].set_ylabel(r'$t$')
ax2.imshow(minv.T, aspect='auto', interpolation='nearest', cmap='gray',
vmin=-np.abs(minv.max()), vmax=np.abs(minv.max()),
exponent=(x.min(), x.max(), t2.max(), t2.min()))
ax2.set_title('Inverted m', fontsize=15)
ax2.set_xlabel(r'$x_V$')
axs[1].set_ylabel(r'$t$')
ax3.plot(madj[int(par['nx']/2)]/np.abs(madj[int(par['nx']/2)]).max(),
t2, 'r', lw=5)
ax3.plot(minv[int(par['nx']/2)]/np.abs(minv[int(par['nx']/2)]).max(),
t2, 'k', lw=3)
ax3.set_ylim([t2[-1], t2[0]])
fig.tight_layout()

fig, axs = plt.subplots(1, 2, figsize=(8, 6))
axs[0].imshow(psfinv[int(par['nx']/2)].T,
    aspect='auto', interpolation='nearest',
    vmin=-np.abs(psfinv.max()), vmax=np.abs(psfinv.max()),
    cmap='gray', extent=(x.min(), x.max(), t2.max(), t2.min()))
axs[0].set_title('Inverted psf - inline view', fontsize=15)
axs[0].set_xlabel(r'$x_V$')
axs[1].set_ylabel(r'$t$')
axs[1].imshow(psfinv[:, int(par['nx']/2)].T,
    aspect='auto', interpolation='nearest',
    vmin=-np.abs(psfinv.max()), vmax=np.abs(psfinv.max()),
    cmap='gray', extent=(x.min(), y.max(), t2.max(), t2.min()))
axs[1].set_title('Inverted psf - xline view', fontsize=15)
axs[1].set_xlabel(r'$x_V$')
axs[1].set_ylabel(r'$t$')
fig.tight_layout()
We repeat the same procedure but this time we will add a preconditioning by means of `causality_precond` parameter, which enforces the inverted model to be zero in the negative part of the time axis (as expected by theory). This preconditioning will have the effect of speeding up the convergence of the iterative solver and thus reduce the computation time of the deconvolution.

```python
minvprec = pylops.waveeqprocessing.MDD(Gwav, d[:, par['nt'] - 1:],
                                         dt=par['dt'], dr=par['dx'],
                                        nfmax=par['nfmax'], wav=wav,
                                        twosided=True, add_negative=True,
                                        adjoint=False, psf=False,
                                        causality_precond=True,
                                        dtype='complex64',
                                        dottest=False,
                                        **dict(damp=1e-4, iter_lim=50, show=0))
```

# sphinx_gallery_thumbnail_number = 5

(continues on next page)
fig = plt.figure(figsize=(8, 6))
ax1 = plt.subplot2grid((1, 5), (0, 0), colspan=2)
ax2 = plt.subplot2grid((1, 5), (0, 2), colspan=2)
ax3 = plt.subplot2grid((1, 5), (0, 4))
ax1.imshow(madj.T, aspect='auto', interpolation='nearest', cmap='gray',
    vmin=-np.abs(madj.max()), vmax=np.abs(madj.max()),
    extent=(x.min(), x.max(), t2.max(), t2.min()))
ax1.set_title('Adjoint m', fontsize=15)
ax1.set_xlabel(r'$x_V$')
ax2.imshow(minvprec.T, aspect='auto', interpolation='nearest', cmap='gray',
    vmin=-np.abs(minvprec.max()), vmax=np.abs(minvprec.max()),
    extent=(x.min(), x.max(), t2.max(), t2.min()))
ax2.set_title('Inverted m', fontsize=15)
ax2.set_xlabel(r'$x_V$')
ax3.plot(madj[int(par['nx']//2)]/np.abs(madj[int(par['nx']//2)]).max(),
    t2, 'r', lw=5)
ax3.plot(minvprec[int(par['nx']//2)]/np.abs(minv[int(par['nx']//2)]).max(),
    t2, 'k', lw=3)
ax3.set_ylim([t2[-1], t2[0]])
fig.tight_layout()
This example shows how to set-up and run the `pylops.waveeqprocessing.Marchenko` inversion using synthetic data.

```python
import warnings
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve
from pylops.waveeqprocessing import Marchenko
```

Total running time of the script: ( 0 minutes 8.312 seconds)
Let’s start by defining some input parameters and loading the test data

```python
# Input parameters
inputfile = '../testdata/marchenko/input.npz'
vel = 2400.0  # velocity
toff = 0.045  # direct arrival time shift
nsmooth = 10  # time window smoothing
nfmax = 1000  # max frequency for MDC (#samples)
niter = 10  # iterations

inputdata = np.load(inputfile)

# Receivers
r = inputdata['r']
r0 = r.shape[1]
Dr = r[0, 1]-r[0, 0]

# Sources
s = inputdata['s']
s0 = s.shape[1]
Ds = s[0, 1]-s[0, 0]

# Virtual points
vs = inputdata['vs']

# Density model
rho = inputdata['rho']
z, x = inputdata['z'], inputdata['x']

# Reflection data (R[s, r, t]) and subsurface fields
R = np.swapaxes(inputdata['R'][:, :, :-100], 0, 1)  # just because of how the data was saved
Gsub = inputdata['Gsub'][:, :-100]
G0sub = inputdata['G0sub'][:, :-100]
wav = inputdata['wav']
wav_c = np.argmax(wav)
t = inputdata['t'][:, :-100]
Dt, Dt, Nt = t[0], t[1]-t[0], len(t)
Gsub = np.apply_along_axis(convolve, 0, Gsub, wav, mode='full')
Gsub = Gsub[:, :-Nt]
G0sub = np.apply_along_axis(convolve, 0, G0sub, wav, mode='full')
G0sub = G0sub[:, :-Nt]

plt.figure(figsize=(10, 5))
plt.imshow(rho, cmap='gray', extent=(x[0], x[-1], z[-1], z[0]))
plt.scatter(s[0, 5::10], s[1, 5::10], marker='*', s=150, c='r', edgecolors='k')
plt.scatter(r[0, ::10], r[1, ::10], marker='v', s=150, c='b', edgecolors='k')
plt.scatter(vs[0], vs[1], marker='.', s=250, c='m', edgecolors='k')
plt.axis('tight')
```

(continues on next page)
plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.title('Model and Geometry')
plt.xlim(x[0], x[-1])

fig, axs = plt.subplots(1, 3, sharey=True, figsize=(12, 7))
axs[0].imshow(R[0].T, cmap='gray', vmin=-1e-2, vmax=1e-2,
             extent=(r[0, 0], r[0, -1], t[-1], t[0]))
axs[0].set_title('R shot=0')
axs[0].set_xlabel(r'$x_R$')
axs[0].set_ylabel(r'$t$')
axs[0].axis('tight')
axs[0].set_ylim(1.5, 0)
axs[1].imshow(R[ns//2].T, cmap='gray', vmin=-1e-2, vmax=1e-2,
             extent=(r[0, 0], r[0, -1], t[-1], t[0]))
axs[1].set_title('R shot=%d' % (ns//2))
axs[1].set_xlabel(r'$x_R$')
axs[1].set_ylabel(r'$t$')
axs[1].axis('tight')
axs[1].set_ylim(1.5, 0)
axs[2].imshow(R[-1].T, cmap='gray', vmin=-1e-2, vmax=1e-2,
             extent=(r[0, 0], r[0, -1], t[-1], t[0]))
axs[2].set_title('R shot=%d' % ns)
axs[2].set_xlabel(r'$x_R$')
axs[2].set_ylabel(r'$t$')
axs[2].axis('tight')
axs[2].set_ylim(1.5, 0)

fig.tight_layout()

fig, axs = plt.subplots(1, 2, sharey=True, figsize=(8, 6))
axs[0].imshow(Gsub, cmap='gray', vmin=-1e6, vmax=1e6,
             extent=(r[0, 0], r[0, -1], t[-1], t[0]))
axs[0].set_title('G')
axs[0].set_xlabel(r'$x_R$')
axs[0].set_ylabel(r'$t$')
axs[0].axis('tight')
axs[0].set_ylim(1.5, 0)
axs[1].imshow(G0sub, cmap='gray', vmin=-1e6, vmax=1e6,
             extent=(r[0, 0], r[0, -1], t[-1], t[0]))
axs[1].set_title('G0')
axs[1].set_xlabel(r'$x_R$')
axs[1].set_ylabel(r'$t$')
axs[1].axis('tight')
axs[1].set_ylim(1.5, 0)

fig.tight_layout()
Let's now create an object of the `pylops.waveeqprocessing.Marchenko` class and apply redatuming for a single subsurface point $v_s$.

```python
# direct arrival window
trav = np.sqrt((vs[0]-r[0])**2+(vs[1]-r[1])**2)/vel

MarchenkoWM = Marchenko(R, dt=dt, dr=dr, nfmax=nfmax, wav=wav,
                         toff=toff, nsn=nsn)

f1_inv_minus, f1_inv_plus, p0_minus, g_inv_minus, g_inv_plus = 
    MarchenkoWM.apply_onepoint(trav, G0=G0sub.T, rtm=True, greens=True,
                               dottest=True, **dict(iter_lim=niter, show=True))

g_inv_tot = g_inv_minus + g_inv_plus
```

Out:
Dot test passed, \( v^T(Opu) = 491.131956 \) \(- u^T(Op^Tv) = 491.131956 \)

Dot test passed, \( v^T(Opu) = 464.113195 \) \(- u^T(Op^Tv) = 464.113195 \)

**LSQR**

Least-squares solution of \( Ax = b \)
The matrix \( A \) has 282598 rows and 282598 columns
\[
damp = 0.00000000000000e+00 \quad \text{calc_var} = 0
\]
\[
atol = 1.00e-08 \quad \text{conlim} = 1.00e+08
\]
\[
btol = 1.00e-08 \quad \text{iter_lim} = 10
\]

<table>
<thead>
<tr>
<th>Itn</th>
<th>x[0]</th>
<th>r1norm</th>
<th>r2norm</th>
<th>Compatible LS Norm A</th>
<th>Cond A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00000e+00</td>
<td>3.134e+07</td>
<td>3.134e+07</td>
<td>1.0e+00</td>
<td>3.3e-08</td>
</tr>
<tr>
<td>1</td>
<td>0.00000e+00</td>
<td>1.374e+07</td>
<td>1.374e+07</td>
<td>4.4e-01</td>
<td>9.3e-01</td>
</tr>
<tr>
<td>2</td>
<td>0.00000e+00</td>
<td>7.770e+06</td>
<td>7.770e+06</td>
<td>2.5e-01</td>
<td>3.9e-01</td>
</tr>
<tr>
<td>3</td>
<td>0.00000e+00</td>
<td>5.750e+06</td>
<td>5.750e+06</td>
<td>1.8e-01</td>
<td>3.3e-01</td>
</tr>
<tr>
<td>4</td>
<td>0.00000e+00</td>
<td>3.930e+06</td>
<td>3.930e+06</td>
<td>1.3e-01</td>
<td>3.4e-01</td>
</tr>
<tr>
<td>5</td>
<td>0.00000e+00</td>
<td>3.042e+06</td>
<td>3.042e+06</td>
<td>9.7e-02</td>
<td>2.6e-01</td>
</tr>
<tr>
<td>6</td>
<td>0.00000e+00</td>
<td>2.423e+06</td>
<td>2.423e+06</td>
<td>7.7e-02</td>
<td>2.2e-01</td>
</tr>
<tr>
<td>7</td>
<td>0.00000e+00</td>
<td>1.675e+06</td>
<td>1.675e+06</td>
<td>5.3e-02</td>
<td>2.5e-01</td>
</tr>
<tr>
<td>8</td>
<td>0.00000e+00</td>
<td>1.248e+06</td>
<td>1.248e+06</td>
<td>4.0e-02</td>
<td>2.0e-01</td>
</tr>
<tr>
<td>9</td>
<td>0.00000e+00</td>
<td>1.004e+06</td>
<td>1.004e+06</td>
<td>3.2e-02</td>
<td>1.5e-01</td>
</tr>
<tr>
<td>10</td>
<td>0.00000e+00</td>
<td>7.762e+05</td>
<td>7.762e+05</td>
<td>2.5e-02</td>
<td>1.8e-01</td>
</tr>
</tbody>
</table>

LSQR finished
The iteration limit has been reached

\[
\begin{align*}
\text{istop} & = 7 & \text{r1norm} & = 7.8e+05 & \text{anorm} & = 4.4e+00 & \text{anorm} & = 6.1e+05 \\
\text{itn} & = 10 & \text{r2norm} & = 7.8e+05 & \text{acond} & = 1.6e+01 & \text{xnorm} & = 3.6e+07
\end{align*}
\]

We can now compare the result of Marchenko redatuming via LSQR with standard redatuming

```python
fig, axs = plt.subplots(1, 3, sharey=True, figsize=(12, 7))
axs[0].imshow(p0_minus.T, cmap='gray', vmin=-5e5, vmax=5e5, extent=(r[0, 0], r[0, -1], t[-1], -t[-1]))
axs[0].set_title(r'$p_0^-$')
axs[0].set_xlabel(r'$x_R$')
axs[0].set_ylabel(r'$t$')
axs[0].axis('tight')
axs[0].set_ylim(1.2, 0)
axs[1].imshow(g_inv_minus.T, cmap='gray', vmin=-5e5, vmax=5e5, extent=(r[0, 0], r[0, -1], t[-1], -t[-1]))
axs[1].set_title(r'$g^-$')
axs[1].set_xlabel(r'$x_R$')
axs[1].set_ylabel(r'$t$')
axs[1].axis('tight')
axs[1].set_ylim(1.2, 0)
axs[2].imshow(g_inv_plus.T, cmap='gray', vmin=-5e5, vmax=5e5, extent=(r[0, 0], r[0, -1], t[-1], -t[-1]))
axs[2].set_title(r'$g^+$')
axs[2].set_xlabel(r'$x_R$')
axs[2].set_ylabel(r'$t$')
axs[2].axis('tight')
axs[2].set_ylim(1.2, 0)
fig.tight_layout()
```

Figure 3.3.8: Comparison of Marchenko redatuming via LSQR with standard redatuming.
ax3 = plt.subplot2grid((1, 5), (0, 4))
ax1.imshow(Gsub, cmap='gray', vmin=-5e5, vmax=5e5,
          extent=(r[0, 0], r[0, -1], t[-1], t[0]))
ax1.set_title(r'$G_{true}$')
axs[0].set_xlabel(r'$x_R$')
axs[0].set_ylabel(r'$t$')
ax1.axis('tight')
ax1.set_ylim(1.2, 0)
ax2.imshow(g_inv_tot.T, cmap='gray', vmin=-5e5, vmax=5e5,
          extent=(r[0, 0], r[0, -1], t[-1], -t[-1]))
ax2.set_title(r'$G_{est}$')
axs[1].set_xlabel(r'$x_R$')
axs[1].set_ylabel(r'$t$')
ax2.axis('tight')
ax2.set_ylim(1.2, 0)
ax3.plot(Gsub[:, nr//2]/Gsub.max(), t, 'r', lw=5)
ax3.plot(g_inv_tot[nr//2, nt-1:]/g_inv_tot.max(), t, 'k', lw=3)
ax3.set_ylim(1.2, 0)
fig.tight_layout()
Note that Marchenko redatuming can also be applied simultaneously to multiple subsurface points. Use `pylops.waveeqprocessing.Marchenko.apply_multiplepoints` instead of `pylops.waveeqprocessing.Marchenko.apply_onepoint`.

**Total running time of the script:** (0 minutes 7.101 seconds)
3.4.11 11. Radon filtering

In this example we will be taking advantage of the `pylops.signalprocessing.Radon2D` operator to perform filtering of unwanted events from a seismic data. For those of you not familiar with seismic data, let’s imagine that we have a data composed of a certain number of flat events and a parabolic event, we are after a transform that allows us to separate such an event from the others and filter it out. Those of you with a geophysics background may immediately realize this is the case of seismic angle (or offset) gathers after migration and those events with parabolic moveout are generally residual multiples that we would like to suppress prior to performing further analysis of our data.

The Radon transform is actually a very good transform to perform such a separation. We can thus devise a simple workflow that takes our data as input, applies a Radon transform, filters some of the events out and goes back to the original domain.

```python
import numpy as np
import matplotlib.pyplot as plt
import pylops
from pylops.utils.wavelets import ricker
plt.close('all')
np.random.seed(0)
```

Let’s first create a data composed on 3 linear events and a parabolic event.

```python
par = {'ox':0, 'dx':2, 'nx':121,
       'ot':0, 'dt':0.004, 'nt':100,
       'f0': 30}

# linear events
v = 1500
T = 0.1
theta = [0, 0, 0]
amp = [1., -2, 0.5]

# parabolic event
tp0 = [0.13]
px = [0]
pxx = [5e-7]
ampp = [0.7]

# create axis
taxis, taxis2, xaxis, yaxis = pylops.utils.seismicevents.makeaxis(par)

# create wavelet
wav = ricker(taxis[:41], f0=par['f0'])[0]

# generate model
y = pylops.utils.seismicevents.linear2d(xaxis, taxis, v, t0,
                                      theta, amp, wav)[1] +
    pylops.utils.seismicevents.parabolic2d(xaxis, taxis, tp0,
                                        px, pxx, ampp, wav)[1]
```

We can now create the `pylops.signalprocessing.Radon2D` operator. We also apply its adjoint to the data to obtain a representation of those 3 linear events overlapping to a parabolic event in the Radon domain. Similarly, we feed the operator to a sparse solver like `pylops.optimization.sparsity.FISTA` to obtain a sparse representation of the data in the Radon domain. At this point we try to filter out the unwanted event. We can see how this is much easier for the sparse transform as each event has a much more compact representation in the Radon domain than for the adjoint transform.
# PyLops

```python
# radon operator
npx = 61
pxmax = 5e-4
px = np.linspace(-pxmax, pxmax, npx)

Rop = pylops.signalprocessing.Radon2D(taxis, xaxis, px, kind='linear',
    interp='nearest', centeredh=False,
    dtype='float64')

# adjoint Radon transform
xadj = Rop.H * y.flatten()
xadj = xadj.reshape(npx, par['nt'])

# sparse Radon transform
xinv, niter, cost =
    pylops.optimization.sparsity.FISTA(Rop, y.flatten(), 15,
        eps=1e1, returninfo=True)
xinv = xinv.reshape(npx, par['nt'])

# filtering
xfilt = np.zeros_like(xadj)
yfilt = Rop * xfilt.flatten()
yfilt = yfilt.reshape(par['nx'], par['nt'])

# filtering on sparse transform
xinvfilt = np.zeros_like(xinv)
yinvfilt = Rop * xinvfilt.flatten()
yinvfilt = yinvfilt.reshape(par['nx'], par['nt'])

Finally we visualize our results.

```python
fig, axs = plt.subplots(1, 5, sharey=True, figsize=(12, 5))

axs[0].imshow(y.T, cmap='gray',
    vmin=-np.abs(y).max(), vmax=np.abs(y).max(),
    extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[0].set_title('Data')
axs[0].axis('tight')

axs[1].imshow(xadj.T, cmap='gray',
    vmin=-np.abs(xadj).max(), vmax=np.abs(xadj).max(),
    extent=(px[0], px[-1], taxis[-1], taxis[0]))
axs[1].axvline(px[npx//2-3], color='r', linestyle='--')
axs[1].axvline(px[npx//2+3], color='r', linestyle='--')
axs[1].set_title('Radon')
axs[1].axis('tight')

axs[2].imshow(yfilt.T, cmap='gray',
    vmin=-np.abs(yfilt).max(), vmax=np.abs(yfilt).max(),
    extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[2].set_title('Filtered data')
axs[2].axis('tight')

axs[3].imshow(xinv.T, cmap='gray',
    vmin=-np.abs(xinv).max(), vmax=np.abs(xinv).max(),
    extent=(px[0], px[-1], taxis[-1], taxis[0]))
axs[3].axvline(px[npx//2-3], color='r', linestyle='--')
```

(continues on next page)
As expected, the Radon domain is a suitable domain for this type of filtering and the sparse transform improves the ability to filter out parabolic events with small curvature.

On the other hand, it is important to note that we have not been able to correctly preserve the amplitudes of each event. This is because the sparse Radon transform can only identify a sparsest response that explain the data within a certain threshold. For this reason an more suitable approach for preserving amplitudes could be to apply a parabolic Raodn transform with the aim of reconstructing only the unwanted event and apply an adaptive subtraction between the input data and the reconstructed unwanted event.

**Total running time of the script:** (0 minutes 27.073 seconds)
3.4.12 12. Seismic regularization

The problem of seismic data regularization (or interpolation) is a very simple one to write, yet ill-posed and very hard to solve.

The forward modelling operator is a simple `pylops.Restriction` operator which is applied along the spatial direction(s).

\[ y = Rx \]

Here \( y = [y_{R_1}^T, y_{R_2}^T, ..., y_{R_N}^T]^T \) where each vector \( y_{R_i} \) contains all time samples recorded in the seismic data at the specific receiver \( R_i \). Similarly, \( x = [x_{r_1}^T, x_{r_2}^T, ..., x_{r_M}^T] \), contains all traces at the regularly and finely sampled receiver locations \( r_i \).

By inverting such an equation we can create a regularized data with densely and regularly spatial direction(s).

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve

import pylops
from pylops.utils.wavelets import ricker
from pylops.utils.seismicevents import makeaxis, linear2d

np.random.seed(0)
plt.close('all')

Let's start by creating a very simple 2d data composed of 3 linear events input parameters

```python
par = {'ox':0, 'dx':2, 'nx':70,
       'ot':0, 'dt':0.004, 'nt':80,
       'f0': 20}

v = 1500
t0_m = [0.1, 0.2, 0.28]
theta_m = [0, 30, -80]
phi_m = [0]
amp_m = [1., -2, 0.5]

# axis
taxis, t2, xaxis, y = makeaxis(par)

# wavelet
wav = ricker(taxis[:41], f0=par['f0'])[0]

# model
__, x = linear2d(xaxis, taxis, v, t0_m, theta_m, amp_m, wav)
```

We can now define the spatial locations along which the data has been sampled. In this specific example we will assume that we have access only to 40% of the ‘original’ locations.

```python
perc_subsampling = 0.6
nxsub = int(np.round(par['nx']*perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(par['nx']))[:nxsub])

# restriction operator
Rop = pylops.Restriction(par['nx']*par['nt'], iava)
```
# data
y = Rop*x.ravel()
y = y.reshape(nxsub, par['nt'])

# mask
ymask = Rop.mask(x.flatten())

# inverse
xinv = Rop / y.ravel()
xinv = xinv.reshape(par['nx'], par['nt'])

fig, axs = plt.subplots(1, 2, sharey=True, figsize=(5, 4))
axs[0].imshow(x.T, cmap='gray', vmin=-2, vmax=2, extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[0].set_title('Model')
axs[0].axis('tight')
axs[1].imshow(ymask.T, cmap='gray', vmin=-2, vmax=2, extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[1].set_title('Masked model')
axs[1].axis('tight')

Out:

(0.0, 138.0, 0.316, 0.0)

As we can see, inverting the restriction operator is not possible without adding any prior information into the inverse
problem. In the following we will consider two possible routes:

- regularized inversion with second derivative along the spatial axis

\[ J = \| y - Rx \|_2 + \epsilon^2 \| \nabla x \|_2 \]

- sparsity-promoting inversion with \texttt{pylops.FFT2} operator used as sparsifying transform

\[ J = \| y - RF^H x \|_2 + \epsilon \| F^H x \|_1 \]

```python
# smooth inversion
D2op = pylops.SecondDerivative(par['nx']*par['nt'],
                               dims=(par['nx'], par['nt']),
                               dir=0, dtype='float64')
xsmooth, _, _ = \
    pylops.waveeqprocessing.SeismicInterpolation(y, par['nx'], iava,
                                                kind='spatial',
                                                **dict(epsRs=[np.sqrt(0.1)],
                                                   damp=np.sqrt(1e-4),
                                                   iter_lim=50, show=0))

# sparse inversion with \texttt{FFT2}
nfft = 2**8
FFTop = pylops.signalprocessing.FFT2D(dims=[par['nx'], par['nt']],
                                       nffts=[nfft, nfft],
                                       sampling=[par['dx'], par['dt']])
X = FFTop*x.flatten()
X = np.reshape(X, (nfft, nfft))
xl1, Xl1, cost = \
    pylops.waveeqprocessing.SeismicInterpolation(y, par['nx'], iava, kind='fk',
                                                nffts=(nfft, nfft),
                                                sampling=(par['dx'], par['dt']),
                                                **dict(niter=50, eps=1e-1,
                                                       returninfo=True))
```

```python
fig, axs = plt.subplots(1, 4, sharey=True, figsize=(13, 4))
axs[0].imshow(x.T, cmap='gray', vmin=-2, vmax=2,
              extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[0].set_title('Model')
axs[0].axis('tight')
axs[1].imshow(ymask.T, cmap='gray', vmin=-2, vmax=2,
              extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[1].set_title('Masked model')
axs[1].axis('tight')
axs[2].imshow(xsmooth.T, cmap='gray', vmin=-2, vmax=2,
              extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[2].set_title('Smoothed model')
axs[2].axis('tight')
axs[3].imshow(xl1.T, cmap='gray', vmin=-2, vmax=2,
              extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[3].set_title('L1 model')
axs[3].axis('tight')
```

(continues on next page)
```python
axs[0].imshow(np.fft.fftshift(np.abs(X[:, :nfft//2-1]), axes=0).T,
             extent=(np.fft.fftshift(FFTop.f1)[0],
                     np.fft.fftshift(FFTop.f1)[-1],
                     FFTop.f2[nfft // 2 - 1], FFTop.f2[0]))
axs[0].set_title('Model in f-k domain')
axs[0].axis('tight')
axs[0].set_xlim(-0.1, 0.1)
axs[0].set_ylim(50, 0)
axs[1].imshow(np.fft.fftshift(np.abs(Xl1[:, :nfft // 2 - 1]), axes=0).T,
              extent=(np.fft.fftshift(FFTop.f1)[0],
                      np.fft.fftshift(FFTop.f1)[-1],
                      FFTop.f2[nfft // 2 - 1], FFTop.f2[0]))
axs[1].set_title('Reconstructed model in f-k domain')
axs[1].axis('tight')
axs[1].set_xlim(-0.1, 0.1)
axs[1].set_ylim(50, 0)
axs[2].plot(cost, 'k', lw=3)
axs[2].set_title('FISTA convergence')
```
We see how adding prior information to the inversion can help improving the estimate of the regularized seismic data. Nevertheless, in both cases the reconstructed data is not perfect. A better sparsifying transform could in fact be chosen here to be the linear `pylops.signalprocessing.Radon2D` transform in spite of the `pylops.FFT2` transform.

```python
npix = 40
pxmax = 1e-3
px = np.linspace(-pxmax, pxmax, npix)
Rop = pylops.signalprocessing.Radon2D(taxis, xaxis, px, engine='numba')

RRop = Rop*Radop

# adjoint
Xadj_fromx = Radop.H*x.flatten()
Xadj_fromx = Xadj_fromx.reshape(npx, par['nt'])

Xadj = RRop.H*y.flatten()
Xadj = Xadj.reshape(npx, par['nt'])

# L1 inverse
xll, Xll, cost =
    pylops.waveeqprocessing.SeismicInterpolation(y, par['nx'], iava, kind='radon-linear', spataxis=xaxis, taxis=taxis, paxis=px, centeredh=True, **dict(niter=50, eps=1e-1, returninfo=True))

fig, axs = plt.subplots(2, 3, sharey=True, figsize=(12, 7))
axs[0][0].imshow(x.T, cmap='gray', vmin=-2, vmax=2, extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[0][0].set_title('Data', fontsize=12)
axs[0][0].axis('tight')
axs[0][1].imshow(ymask.T, cmap='gray', vmin=-2, vmax=2, extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[0][1].set_title('Masked data', fontsize=12)
axs[0][1].axis('tight')
as
```
extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
axs[0][2].set_title('Reconstructed data', fontsize=12)
axs[0][2].axis('tight')
axs[1][0].imshow(Xadj_fromx.T, cmap='gray', vmin=-70, vmax=70,
    extent=(px[0], px[-1], taxis[-1], taxis[0]))
axs[1][0].set_title('Adj. Radon on data', fontsize=12)
axs[1][0].axis('tight')
axs[1][1].imshow(Xadj.T, cmap='gray', vmin=-50, vmax=50,
    extent=(px[0], px[-1], taxis[-1], taxis[0]))
axs[1][1].set_title('Adj. Radon on subsampled data', fontsize=12)
axs[1][1].axis('tight')
axs[1][2].imshow(Xl1.T, cmap='gray', vmin=-0.2, vmax=0.2,
    extent=(px[0], px[-1], taxis[-1], taxis[0]))
axs[1][2].set_title('Inverse Radon on subsampled data', fontsize=12)
axs[1][2].axis('tight')
Finally, let’s take now a more realistic dataset. We will use once again the linear `pylops.signalprocessing.Radon2D` transform but we will take advantage of the `pylops.signalprocessing.Sliding2D` operator to perform such a transform locally instead of globally to the entire dataset.

```python
inputfile = '../testdata/marchenko/input.npz'
```

Out:

```
(-0.001, 0.001, 0.316, 0.0)
```
inputdata = np.load(inputfile)

x = inputdata['R'][50, :, ::2]

x = x/np.abs(x).max()
taxis, xaxis = inputdata['t'][::2], inputdata['r'][0]

par = {
    'nx', 'nt' = x.shape
    'dx' = inputdata['r'][0, 1] - inputdata['r'][0, 0]
    'dt' = inputdata['t'][1] - inputdata['t'][0]
}

wav = inputdata['wav'][::2]
wav_c = np.argmax(wav)
x = np.apply_along_axis(convolve, 1, x, wav, mode='full')
x = x[:, wav_c:

# gain

gain = np.tile((taxis**2)[:, np.newaxis], (1, par['nx'])).T
x = x*gain

# subsampling locations

perc_subsampling = 0.5
Nsub = int(np.round(par['nx'])*perc_subsampling)
iava = np.sort(np.random.permutation(np.arange(par['nx']))[:Nsub])

# restriction operator

Rop = pylops.Restriction(par['nx']*par['nt'], iava, 
                           dims=(par['nx'], par['nt']), 
                           dir=0, dtype='float64')

y = Rop*x.flatten()
xadj = Rop.H*y.flatten()

y = y.reshape(Nsub, par['nt'])
xadj = xadj.reshape(par['nx'], par['nt'])

# apply mask

ymask = Rop.mask(x.flatten())

# sliding windows with radon transform

dx = par['dx']
nwins = 4
nwin = 27
nover = 3
npx = 31
pxmax = 5e-4
px = np.linspace(-pxmax, pxmax, npx)
dimsd = x.shape

dims = (nwins*npx, dimsd[1])

Op = 
    pylops.signalprocessing.Radon2D(taxis, np.linspace(-par['dx']*nwin/2, 
                                                    par['dx']*nwin/2, 
                                                    nwin),
                                        px, centeredh=True, kind='linear',
                                        engine='numba')
Slidop = pylops.signalprocessing.Sliding2D(Op, dims, dimsd, nwin, nover,
    tapertype='cosine', design=True)

# adjoint
RSop = Rop*Slidop

Xadj_fromx = Slidop.H*x.flatten()
Xadj_fromx = Xadj_fromx.reshape(npx*nwins, par['nt'])

Xadj = RSop.H*y.flatten()
Xadj = Xadj.reshape(npx*nwins, par['nt'])

# inverse
xll, Xl1, _ = \
    pylops.waveeqprocessing.SeismicInterpolation(y, par['nx'], iava,
        kind='sliding',
        spataxis=xaxis,
        taxis=taxis, paxis=px,
        nwins=nwins, nwin=nwin,
        nover=nover,
        **dict(niter=50, eps=1e-2))

fig, axs = plt.subplots(2, 3, sharey=True, figsize=(12, 14))
asx[0][0].imshow(x.T, cmap='gray', vmin=-0.1, vmax=0.1,
    extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
asx[0][0].set_title('Data')
asx[0][0].axis('tight')
asx[0][1].imshow(ymask.T, cmap='gray', vmin=-0.1, vmax=0.1,
    extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
asx[0][1].set_title('Masked data')
asx[0][1].axis('tight')
asx[0][2].imshow(xll.T, cmap='gray', vmin=-0.1, vmax=0.1,
    extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]))
asx[0][2].set_title('Reconstructed data')
asx[0][2].axis('tight')
asx[1][0].imshow(Xadj_fromx.T, cmap='gray', vmin=-1, vmax=1,
    extent=(px[0], px[-1], taxis[-1], taxis[0]))
asx[1][0].set_title('Adjoint Radon on data')
asx[1][0].axis('tight')
asx[1][1].imshow(Xadj.T, cmap='gray', vmin=-0.6, vmax=0.6,
    extent=(px[0], px[-1], taxis[-1], taxis[0]))
asx[1][1].set_title('Adjoint Radon on subsampled data')
asx[1][1].axis('tight')
asx[1][2].imshow(Xl1.T, cmap='gray', vmin=-0.03, vmax=0.03,
    extent=(px[0], px[-1], taxis[-1], taxis[0]))
asx[1][2].set_title('Inverse Radon on subsampled data')
asx[1][2].axis('tight')
Out:

```
(-0.0005, 0.0005, 1.995, 0.0)
```

As expected the linear `pylops.signalprocessing.Radon2D` is able to locally explain events in the input data and leads to a satisfactory recovery. Note that increasing the number of iterations and sliding windows can further refine the result, especially the accuracy of weak events, as shown in this companion notebook.

**Total running time of the script:** (0 minutes 5.161 seconds)

### 3.4.13 13. Deghosting

Single-component seismic data can be decomposed in their up- and down-going constituents in a model driven fashion. This task can be achieved by defining an f-k propagator (or ghost model) and solving an inverse problem as described in `pylops.waveeqprocessing.Deghosting`.

```python
# sphinx_gallery_thumbnail_number = 3
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse.linalg import lsqr
import pylops

np.random.seed(0)
plt.close('all')
Let's start by loading the input dataset and geometry

```
inputfile = '../testdata/updown/input.npz'
inputdata = np.load(inputfile)

vel_sep = 2400.0  # velocity at separation level
clip = 1e-1  # plotting clip

# Receivers
r = inputdata['r']
r = r.shape[1]
dr = r[0, 1] - r[0, 0]

# Sources
s = inputdata['s']

# Model
rho = inputdata['rho']

# Axes
nt, dt = len(t), t[1]-t[0]
x, z = inputdata['x'], inputdata['z']
dx, dz = x[1] - x[0], z[1] - z[0]

# Data
p = inputdata['p'].T
p /= p.max()

fig = plt.figure(figsize=(9, 4))
```

(continues on next page)
To be able to deghost the input dataset, we need to remove its direct arrival. In this example we will create a mask based on the analytical traveltime of the direct arrival.

```python
direct = np.sqrt(np.sum((s[:, np.newaxis]-r)**2, axis=0))/vel_sep
```

# Window
off = 0.035
direct_off = direct + off
win = np.zeros((nt, nr))
iwin = np.round(direct_off/dt).astype(np.int)
for i in range(nr):
    win[iwin[i]:, i] = 1

fig, axs = plt.subplots(1, 2, sharey=True, figsize=(8, 7))
axs[0].imshow(p.T, cmap='gray', vmin=-clip*np.abs(p).max(),
              vmax=clip*np.abs(p).max(),
              extent=(r[0, 0], r[0, -1], t[-1], t[0]))
axs[0].plot(r[0], direct_off, 'r', lw=2)
axs[0].set_title(r'$P$')
axs[0].axis('tight')
axs[1].imshow(win * p.T, cmap='gray', vmin=-clip*np.abs(p).max(),
              vmax=clip*np.abs(p).max(),
              extent=(r[0, 0], r[0, -1], t[-1], t[0]))
axs[1].set_title(r'Windowed $P$')
axs[1].axis('tight')
axs[1].set_ylim(1, 0)
Out:

\[(1.0, 0.0)\]

We can now perform deghosting

```python
pup, pdown = 
    pylops.waveeqprocessing.Deghosting(p.T, nt, nr, dt, dr, vel_sep,
    r[1, 0] + dz, win=win,
```

(continues on next page)
fig, axs = plt.subplots(1, 3, sharey=True, figsize=(12, 7))
axs[0].imshow(p.T, cmap='gray', vmin=-clip * np.abs(p).max(),
              vmax=clip * np.abs(p).max(),
              extent=(r[0, 0], r[0, -1], t[-1], t[0]))
axs[0].set_title(r'$P$')
axs[0].axis('tight')
axs[1].imshow(pup, cmap='gray', vmin=-clip * np.abs(p).max(),
              vmax=clip * np.abs(p).max(),
              extent=(r[0, 0], r[0, -1], t[-1], t[0]))
axs[1].set_title(r'$P^-$')
axs[1].axis('tight')
axs[2].imshow(pdown, cmap='gray', vmin=-clip * np.abs(p).max(),
              vmax=clip * np.abs(p).max(),
              extent=(r[0, 0], r[0, -1], t[-1], t[0]))
axs[2].set_title(r'$P^+$')
axs[2].axis('tight')
axs[2].set_ylim(1, 0)

plt.figure(figsize=(14, 3))
plt.plot(t, p[nr // 2], 'k', lw=2, label=r'$p$')
plt.plot(t, pup[:, nr // 2], 'r', lw=2, label=r'$p^-$')
plt.xlim(0, t[200])
plt.ylim(-0.2, 0.2)
plt.legend()

plt.figure(figsize=(14, 3))
plt.plot(t, pdown[:, nr // 2], 'b', lw=2, label=r'$p^+$')
plt.plot(t, pup[:, nr // 2], 'r', lw=2, label=r'$p^-$')
plt.xlim(0, t[200])
plt.ylim(-0.2, 0.2)
plt.legend()
3.4.14 14. Seismic wavefield decomposition

Multi-component seismic data can be decomposed in their up- and down-going constituents in a purely data driven fashion. This task can be accurately achieved by linearly combining the input pressure and particle velocity data in the frequency-wavenumber described in details in pylops.waveeqprocessingUpDownComposition2D and pylops.waveeqprocessing.WavefieldDecomposition.
In this tutorial we will consider a simple synthetic data composed of six events (three up-going and three down-going). We will first combine them to create pressure and particle velocity data and then show how we can retrieve their directional constituents both by directly combining the input data as well as by setting an inverse problem. The latter approach results vital in case of spatial aliasing, as applying simple scaled summation in the frequency-wavenumber would result in sub-optimal decomposition due to the superposition of different frequency-wavenumber pairs at some (aliased) locations.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import filtfilt
import pylops
from pylops.utils.wavelets import ricker
from pylops.utils.seismicevents import makeaxis, hyperbolic2d

np.random.seed(0)
plt.close('all')

Let's first the input up- and down-going wavefields

```python
par = {'ox':-220, 'dx':5, 'nx':89,
      'ot':0, 'dt':0.004, 'nt':200,
      'f0': 40}
t0_plus = np.array([0.2, 0.5, 0.7])
t0_minus = t0_plus + 0.04
vrms = np.array([1400., 1500., 2000.])
amp = np.array([1., -0.6, 0.5])
vel_sep = 1000.0  # velocity at separation level
rho_sep = 1000.0  # density at separation level

# Create axis
t, t2, x, y = makeaxis(par)

# Create wavelet
wav = ricker(t[:41], f0=par['f0'])[0]

# Create data
_, p_minus = hyperbolic2d(x, t, t0_minus, vrms, amp, wav)
_, p_plus = hyperbolic2d(x, t, t0_plus, vrms, amp, wav)

We can now combine them to create pressure and particle velocity data

```python
critical = 1.1
ntaper = 51
nfft = 2**10

# 2d fft operator
FFTop = pylops.signalprocessing.FFT2D(dims=[par['nx'], par['nt']],
                                       nffts=[nfft, nfft],
                                       sampling=[par['dx'], par['dt']])

# obliquity factor
[Kx, F] = np.meshgrid(FFTop.f1, FTop.f2, indexing='ij')
k = F/vel_sep
Kz = np.sqrt((k**2-Kx**2).astype(np.complex))
Kz[np.isnan(Kz)] = 0
OBL = rho_sep*(np.abs(F)/Kz)
```

(continues on next page)
OBL[Kz == 0] = 0

mask = np.abs(Kx) < critical*np.abs(F)/vel_sep
OBL *= mask
OBL = filtfilt(np.ones(ntaper)/float(ntaper), 1, OBL, axis=0)
OBL = filtfilt(np.ones(ntaper)/float(ntaper), 1, OBL, axis=1)

# composition operator
UPop = pylops.waveeqprocessing.UpDownComposition2D(par['nt'], par['nx'],
par['dt'], par['dx'],
rho_sep, vel_sep,
nffts=(nfft, nfft),
critical=critical*100.,
ntaper=ntaper,
dtype='complex128')

# wavefield modelling
p, vz = d[:par['nx']], d[par['nx']:]
VZ_obl = OBL * VZ
vz_obl = FFTop.H*VZ_obl.flatten()

fig, axs = plt.subplots(1, 4, figsize=(10, 5))
axs[0].imshow(p.T, aspect='auto', vmin=-1, vmax=1,
interpolation='nearest', cmap='gray',
extent=(x.min(), x.max(), t.max(), t.min()))
axs[0].set_title(r'$p$', fontsize=15)
axs[0].set_xlabel('x')
axs[0].set_ylabel('t')
axs[1].imshow(vz_obl.T, aspect='auto', vmin=-1, vmax=1,
interpolation='nearest', cmap='gray',
extent=(x.min(), x.max(), t.max(), t.min()))
axs[1].set_title(r'$v_z^{obl}$', fontsize=15)
axs[1].set_xlabel('x')
axs[1].set_ylabel('t')
axs[2].imshow(p_plus.T, aspect='auto', vmin=-1, vmax=1,
interpolation='nearest', cmap='gray',
extent=(x.min(), x.max(), t.max(), t.min()))
axs[2].set_title(r'$p^+$', fontsize=15)
axs[2].set_xlabel('x')
axs[2].set_ylabel('t')
axs[3].imshow(p_minus.T, aspect='auto', vmin=-1, vmax=1,
interpolation='nearest', cmap='gray',
extent=(x.min(), x.max(), t.max(), t.min()))
axs[3].set_title(r'$p^-$', fontsize=15)
axs[3].set_xlabel('x')
axs[3].set_ylabel('t')
plt.tight_layout()
Wavefield separation is first performed using the analytical expression for combining pressure and particle velocity data in the wavenumber-frequency domain:

```python
pup_sep, pdown_sep = 
    pylops.waveeqprocessing.WavefieldDecomposition(p, vz, par['nt'], par['nx'],
        par['dt'], par['dx'],
        rho_sep, vel_sep,
        nffts=(nfft, nfft),
        kind='analytical',
        critical=critical*100,
        ntaper=ntaper,
        dtype='complex128')
```

```python
fig = plt.figure(figsize=(12, 5))
axs0 = plt.subplot2grid((2, 5), (0, 0), rowspan=2)
axs1 = plt.subplot2grid((2, 5), (0, 1), rowspan=2)
axs2 = plt.subplot2grid((2, 5), (0, 2), colspan=3)
axs3 = plt.subplot2grid((2, 5), (1, 2), colspan=3)
```

(continues on next page)
We repeat the same exercise but this time we invert the composition operator `pylops.waveeqprocessing.UpDownComposition2D`:

```python
pup_inv, pdown_inv = \\n    pylops.waveeqprocessing.WavefieldDecomposition(p, vz, par['nt'], par['nx'],
        par['dt'], par['dx'],
```
fig = plt.figure(figsize=(12, 5))
asx0 = plt.subplot2grid((2, 5), (0, 0), rowspan=2)
asx1 = plt.subplot2grid((2, 5), (0, 1), rowspan=2)
asx2 = plt.subplot2grid((2, 5), (0, 2), colspan=3)
asx3 = plt.subplot2grid((2, 5), (1, 2), colspan=3)
asx0.imshow(pup_inv.T, cmap='gray', vmin=-1, vmax=1, extent=(x.min(), x.max(), t.max(), t.min()))
asx0.set_title(r'$p^-$ inverse')
asx0.axis('tight')
asx1.imshow(pdown_inv.T, cmap='gray', vmin=-1, vmax=1, extent=(x.min(), x.max(), t.max(), t.min()))
asx1.set_title(r'$p^+$ inverse')
asx1.axis('tight')
asx2.plot(t, p[par['nx']//2], 'r', lw=2, label=r'$p$')
asx2.plot(t, vz_obl[par['nx']//2], '--b', lw=2, label=r'$v_z^{obl}$')
asx2.set_ylim(-1, 1)
asx2.set_title('Data at x=%.2f' % x[par['nx']//2])
asx2.set_xlabel('t [s]')
asx2.legend()
asx3.plot(t, pup_inv[par['nx']//2], 'r', lw=2, label=r'$p^-$ inv')
asx3.plot(t, pdown_inv[par['nx']//2], '--b', lw=2, label=r'$p^+$ inv')
asx3.set_title('Separated wavefields at x=%.2f' % x[par['nx']//2])
asx3.set_xlabel('t [s]')
asx3.set_ylim(-1, 1)
asx3.legend()
plt.tight_layout()
The up- and down-going constituents have been successfully separated in both cases. Finally, we use the `pylops.waveeqprocessing.UpDownComposition2D` operator to reconstruct the particle velocity wavefield from its up- and down-going pressure constituents.

```python
PtoVop = 
    pylops.waveeqprocessing.PressureToVelocity(par['nt'], par['nx'], 
    par['dt'], par['dx'], 
    rho_sep, vel_sep, 
    nffts=(nfft, nfft), 
    critical=critical * 100., 
    ntaper=ntaper, 
    topressure=False)

vdown_rec = (PtoVop * pdown_inv.ravel()).reshape(par['nx'], par['nt'])
vup_rec = (PtoVop * pup_inv.ravel()).reshape(par['nx'], par['nt'])
vz_rec = np.real(vdown_rec - vup_rec)

fig, axs = plt.subplots(1, 3, figsize=(13, 6))
axs[0].imshow(vz.T, cmap='gray', vmin=-1e-6, vmax=1e-6, 
    extent=(x.min(), x.max(), t.max(), t.min()))
axs[0].set_title(r'$v_z$')
axs[0].axis('tight')
axs[1].imshow(vz_rec.T, cmap='gray', vmin=-1e-6, vmax=1e-6, 
    extent=(x.min(), x.max(), t.max(), t.min()))
axs[1].set_title(r'$p^+_{\text{inv}}$')
axs[1].axis('tight')
axs[2].plot(p, 'r', label='p')
axs[2].plot(vz_rec, 'b', label='v_z')
axs[2].set_xlim(0, 0.2)
axs[2].set_ylim(-1, 1)
axs[2].set_title(r'$v_z$')
axs[2].legend()
```

(continues on next page)
To see more examples, including applying wavefield separation and regularization simultaneously, as well as 3D examples, head over to the following notebooks: notebook1 and notebook2

**Total running time of the script:** (0 minutes 7.427 seconds)
3.4.15 15. Least-squares migration

Seismic migration is the process by which seismic data are manipulated to create an image of the subsurface reflectivity.

While traditionally solved as the adjoint of the demigration operator, it is becoming more and more common to solve the underlying inverse problem in the quest for more accurate and detailed subsurface images.

Independently of the choice of the modelling operator (i.e., ray-based or full wavefield-based), the demigration/migration process can be expressed as a linear operator of such a kind:

\[ d(x_r, x_s, t) = w(t) \ast \int_{V} G(x, x_s, t)G(x_r, x, t)m(x)dx \]

where \( m(x) \) is the reflectivity at every location in the subsurface, \( G(x, x_s, t) \) and \( G(x_r, x, t) \) are the Green's functions from source-to-subsurface-to-receiver and finally \( w(t) \) is the wavelet. Ultimately, while the Green's functions can be computed in many different ways, solving this system of equations for the reflectivity model is what we generally refer to as Least-squares migration (LSM).

In this tutorial we will consider the most simple scenario where we use an eikonal solver to compute the Green's functions and show how we can use the `pylops.waveeqprocessing.LSM` operator to perform LSM.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse.linalg import lsqr
import pylops
plt.close('all')
np.random.seed(0)

To start we create a simple model with 2 interfaces

```
plt.figure(figsize=(10, 5))
im = plt.imshow(vel.T, cmap='gray', extent = (x[0], x[-1], z[-1], z[0]))
plt.scatter(recs[0], recs[1], marker='v', s=150, c='b', edgecolors='k')
plt.scatter(sources[0], sources[1], marker='*', s=150, c='r', edgecolors='k')
plt.colorbar(im)
plt.axis('tight')
plt.xlabel('x [m]'), plt.ylabel('y [m]')
plt.title('Velocity')
plt.xlim(x[0], x[-1])

plt.figure(figsize=(10, 5))
im = plt.imshow(refl.T, cmap='gray', extent = (x[0], x[-1], z[-1], z[0]))
plt.scatter(recs[0], recs[1], marker='v', s=150, c='b', edgecolors='k')
plt.scatter(sources[0], sources[1], marker='*', s=150, c='r', edgecolors='k')
plt.colorbar(im)
plt.axis('tight')
plt.xlabel('x [m]'), plt.ylabel('y [m]')
plt.title('Reflectivity')
plt.xlim(x[0], x[-1])
We can now create our LSM object and invert for the reflectivity using two different solvers: `scipy.sparse.linalg.lsqr` (LS solution) and `pylops.optimization.sparsity.FISTA` (LS solution with sparse model).

```python
nt = 651
dt = 0.004
t = np.arange(nt)*dt
wav, wavt, wavc = pylops.utils.wavelets.ricker(t[:41], f0=20)

lsm = pylops.waveeqprocessing.LSM(z, x, t, sources, recs, v0, wav, wavc, mode='analytic')

d = lsm.Demop * refl.ravel()
d = d.reshape(ns, nr, nt)

madj = lsm.Demop.H * d.ravel()
madj = madj.reshape(nx, nz)
```

(continues on next page)
minv = lsm.solve(d.ravel(), solver=lsqr, **dict(iter_lim=100))
minv = minv.reshape(nx, nz)

minv_sparse = lsm.solve(d.ravel(),
    solver=pylops.optimization.sparsity.FISTA,
    **dict(eps=1e2, niter=100))
minv_sparse = minv_sparse.reshape(nx, nz)

# demigration
dadj = lsm.Demop * madj.ravel()
dadj = dadj.reshape(ns, nr, nt)
dinv = lsm.Demop * minv.ravel()
dinv = dinv.reshape(ns, nr, nt)
dinv_sparse = lsm.Demop * minv_sparse.ravel()
dinv_sparse = dinv_sparse.reshape(ns, nr, nt)

# sphinx_gallery_thumbnail_number = 2
fig, axs = plt.subplots(2, 2, figsize=(10, 8))
axs[0][0].imshow(refl.T, cmap='gray', vmin=-1, vmax=1)
axs[0][0].axis('tight')
axs[0][0].set_title(r'$m$')
axs[0][1].imshow(madj.T, cmap='gray', vmin=-madj.max(), vmax=madj.max())
axs[0][1].set_title(r'$m_{adj}$')
axs[0][1].axis('tight')
axs[1][0].imshow(minv.T, cmap='gray', vmin=-1, vmax=1)
axs[1][0].axis('tight')
axs[1][0].set_title(r'$m_{inv}$')
axs[1][1].imshow(minv_sparse.T, cmap='gray', vmin=-1, vmax=1)
axs[1][1].axis('tight')
axs[1][1].set_title(r'$m_{FISTA}$')

fig, axs = plt.subplots(1, 4, figsize=(10, 4))
axs[0].imshow(d[0, :, :300].T, cmap='gray', vmin=-d.max(), vmax=d.max())
axs[0].set_title(r'$d$')
axs[0].axis('tight')
axs[1].imshow(dadj[0, :, :300].T, cmap='gray', vmin=-dadj.max(), vmax=dadj.max())
axs[1].set_title(r'$d_{adj}$')
axs[1].axis('tight')
axs[2].imshow(dinv[0, :, :300].T, cmap='gray', vmin=-d.max(), vmax=d.max())
axs[2].set_title(r'$d_{inv}$')
axs[2].axis('tight')
axs[3].imshow(dinv_sparse[0, :, :300].T, cmap='gray', vmin=-d.max(), vmax=d.max())
axs[3].set_title(r'$d_{fista}$')
axs[3].axis('tight')

(continues on next page)
axs[1].set_title(r'$d_{\text{adj}}$')
axs[1].axis('tight')
axs[2].imshow(dinv[ns//2, :, :300].T, cmap='gray',
            vmin=-d.max(), vmax=d.max())
axs[2].set_title(r'$d_{\text{inv}}$')
axs[2].axis('tight')
axs[3].imshow(dinv_sparse[ns//2, :, :300].T, cmap='gray',
            vmin=-d.max(), vmax=d.max())
axs[3].set_title(r'$d_{\text{fista}}$')
axs[3].axis('tight')
Out:
This was just a short teaser, for a more advanced set of examples of 2D and 3D traveltine-based LSM head over to
this notebook.

**Total running time of the script:** ( 0 minutes 4.510 seconds)

### 3.4.16 16. CT Scan Imaging

This tutorial considers a very well-known inverse problem from the field of medical imaging.

We will be using the `pylops.signalprocessing.Radon2D` operator to model a *sinogram*, which is a graphic
representation of the raw data obtained from a CT scan. The sinogram is further inverted using both a L2 solver and a
TV-regularized solver like Split-Bregman.

```python
# sphinx_gallery_thumbnail_number = 2
import numpy as np
import matplotlib.pyplot as plt
import pylops
from numba import jit
plt.close('all')
np.random.seed(10)

Let’s start by loading the Shepp-Logan phantom model. We can then construct the sinogram by providing a custom-
made function to the `pylops.signalprocessing.Radon2D` that samples parametric curves of such a type:

\[ t(r, \theta; x) = \tan(90 - \theta) \cdot x + r / \sin(\theta) \]

where \( \theta \) is the angle between the x-axis \( x \) and the perpendicular to the summation line and \( r \) is the distance from the
origin of the summation line.

```python
@jit(nopython=True)
def radoncurve(x, r, theta):
    return (r - ny//2)/(np.sin(np.deg2rad(theta))+1e-15) + np.tan(np.deg2rad(90 - theta))*x + ny//2

x = np.load('../testdata/optimization/shepp_logan_phantom.npy')
x = x / x.max()
ny, nx = x.shape
ntheta = 150
theta = np.linspace(0., 180., ntheta, endpoint=False)

RLop = pylops.signalprocessing.Radon2D(np.arange(ny), np.arange(nx),
theta, kind=radoncurve,
centered=True, interp=False,
engine='numba', dtype='float64')

y = RLoop.H * x.T.ravel()
y = y.reshape(ntheta, ny).T
```

We can now first perform the adjoint, which in the medical imaging literature is also referred to as back-projection.

This is the first step of a common reconstruction technique, named filtered back-projection, which simply applies a
correction filter in the frequency domain to the adjoint model.
Finally we take advantage of our different solvers and try to invert the modelling operator both in a least-squares sense and using TV-reg.

```python
Dop = [
    pylops.FirstDerivative(ny * nx, dims=(nx, ny), dir=0, edge=True, 
    kind='backward', dtype=np.float),
    pylops.FirstDerivative(ny * nx, dims=(nx, ny), dir=1, edge=True, 
    kind='backward', dtype=np.float)]
D2op = pylops.Laplacian(dims=(nx, ny), edge=True, dtype=np.float)

# L2
xinv_sm = \\n    pylops.optimization.leastsquares.RegularizedInversion(RLop.H, 
    [D2op],
(continues on next page)
xinv_sm = np.real(xinv_sm.reshape(nx, ny)).T

# TV
mu = 1.5
lamda = [1., 1.]
niter = 3
niterinner = 4

xinv, niter = pylops.optimization.sparsity.SplitBregman(RLop.H, Dop, y.T.flatten(),
            niter, niterinner,
            mu=mu, epsRL1s=lamda, tol=1e-4, tau=1., show=False,
            **dict(iter_lim=20, damp=1e-2))
xinv = np.real(xinv.reshape(nx, ny)).T

fig, axs = plt.subplots(1, 3, figsize=(10, 4))
axs[0].imshow(x, vmin=0, vmax=1, cmap='gray')
axs[0].set_title('Model')
axs[0].axis('tight')
axs[1].imshow(xinv_sm, vmin=0, vmax=1, cmap='gray')
axs[1].set_title('L2 Inversion')
axs[1].axis('tight')
axs[2].imshow(xinv, vmin=0, vmax=1, cmap='gray')
axs[2].set_title('TV-Reg Inversion')
axs[2].axis('tight')
fig.tight_layout()
3.5 Frequently Asked Questions

1. Can I visualize my operator?

Yes, you can. Every operator has a method called todense that will return the dense matrix equivalent of the operator. Note, however, that in order to do so we need to allocate a numpy array of the size of your operator and apply the operator N times, where N is the number of columns of the operator. The allocation can be very heavy on your memory and the computation may take long time, so use it with care only for small toy examples to understand what your operator looks like. This method should however not be abused, as the reason of working with linear operators is indeed that you don’t really need to access the explicit matrix representation of an operator.

3.6 PyLops API

The Application Programming Interface (API) of PyLops can be loosely seen as composed of a stack of three main layers:

- **Linear operators**: building blocks for the setting up of inverse problems
- **Solvers**: interfaces to a variety of solvers, providing an easy way to augment an inverse problem with additional regularization and/or preconditioning term
- **Applications**: high-level interfaces allowing users to easily setup and solve specific problems (while hiding the non-needed details - i.e., creation and setup of linear operators and solvers).

3.6.1 Linear operators

**Templates**

| **LinearOperator**([Op, explicit]) | Common interface for performing matrix-vector products. |
| **FunctionOperator**(f, *args, **kwargs) | Function Operator. |

**pylops.LinearOperator**

```python
class pylops.LinearOperator(Op=None, explicit=False)
    Common interface for performing matrix-vector products.
```

This class is an overload of the `scipy.sparse.linalg.LinearOperator` class. It adds functionalities by overloading standard operators such as `__truediv__` as well as creating convenience methods such as `eigs`, `cond`, and `conj`.

**Note**: End users of PyLops should not use this class directly but simply use operators that are already implemented. This class is meant for developers and it has to be used as the parent class of any new operator developed within PyLops. Find more details regarding implementation of new operators at *Implementing new operators*.

**Parameters**
PyLops

**Op** [scipy.sparse.linalg.LinearOperator or scipy.sparse.linalg._ProductLinearOperator or scipy.sparse.linalg._SumLinearOperator] Operator

`explicit` [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(Op, explicit)</code></td>
<td>Initialize this LinearOperator.</td>
</tr>
<tr>
<td><code>adjoint()</code></td>
<td>Hermitian adjoint.</td>
</tr>
<tr>
<td><code>apply_columns(cols)</code></td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td><code>cond([uselobpcg])</code></td>
<td>Condition number of linear operator.</td>
</tr>
<tr>
<td><code>conj()</code></td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td><code>div(y[, niter])</code></td>
<td>Solve the linear problem $y = Ax$.</td>
</tr>
<tr>
<td><code>dot(x)</code></td>
<td>Matrix-matrix or matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>eigs([neigs, symmetric, niter, uselobpcg])</code></td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td><code>matmat(X)</code></td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>matvec(x)</code></td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>rmatmat(X)</code></td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>rmatvec(x)</code></td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>todense()</code></td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td><code>tosparse()</code></td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td><code>transpose()</code></td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

**div**(y, niter=100)

Solve the linear problem $y = Ax$.

Overloading of operator / to improve expressivity of Pylops when solving inverse problems.

**Parameters**

- **y** [np.ndarray] Data
- **niter** [int, optional] Number of iterations (to be used only when explicit=False)

**Returns**

- **xest** [np.ndarray] Estimated model

**todense()**

Return dense matrix.

The operator in converted into its dense matrix equivalent. In order to do so, the operator is applied to an identity matrix whose number of rows and columns is equivalent to the number of columns of the operator. Note that this operation may be costly for very large operators and it is only suggest it to use as a way to inspect the structure of the matricial equivalent of the operator.

**Returns**

- **matrix** [numpy.ndarray] Dense matrix.

**tosparse()**

Return sparse matrix.

The operator in converted into its sparse (CSR) matrix equivalent. In order to do so, the operator is applied to series of unit vectors with length equal to the number of columns in the original operator.

**Returns**
**eigs** *(neigs=None, symmetric=False, niter=None, uselobpcg=False, **kwargs_eig)*

Most significant eigenvalues of linear operator.

Return an estimate of the most significant eigenvalues of the linear operator. If the operator has rectangular shape (shape[0]!=shape[1]), eigenvalues are first computed for the square operator $A^H A$ and the square-root values are returned.

**Parameters**

- **neigs** [int] Number of eigenvalues to compute (if None, return all). Note that for explicit=False, only $N - 1$ eigenvalues can be computed where $N$ is the size of the operator in the model space.
- **symmetric** [bool, optional] Operator is symmetric (True) or not (False). User should set this parameter to True only when it is guaranteed that the operator is real-symmetric or complex-hermitian matrices.
- **niter** [int, optional] Number of iterations for eigenvalue estimation.
- **uselobpcg** [bool, optional] Use scipy.sparse.linalg.lobpcg.
- **kwargs_eig** Arbitrary keyword arguments for scipy.sparse.linalg.eigs, scipy.sparse.linalg.eigsh, or scipy.sparse.linalg.lobpcg.

**Returns**

- **eigenvalues** [numpy.ndarray] Operator eigenvalues.

**Raises**

- **ValueError** If uselobpcg=True for a non-symmetric square matrix with complex type.

**Notes**

Depending on the size of the operator, whether it is explicit or not and the number of eigenvalues requested, different algorithms are used by this routine.

More precisely, when only a limited number of eigenvalues is requested the scipy.sparse.linalg.eigsh method is used in case of symmetric=True and the scipy.sparse.linalg.eigs method is used symmetric=False. However, when the matrix is represented explicitly within the linear operator (explicit=True) and all the eigenvalues are requested the scipy.linalg.eigvals is used instead.

Finally, when only a limited number of eigenvalues is required, it is also possible to explicitly choose to use the scipy.sparse.linalg.lobpcg method via the uselobpcg input parameter flag.

Most of these algorithms are a port of ARPACK [1], a Fortran package which provides routines for quickly finding eigenvalues/eigenvectors of a matrix. As ARPACK requires only left-multiplication by the matrix in question, eigenvalues/eigenvectors can also be estimated for linear operators when the dense matrix is not available.

**cond** *(uselobpcg=False, **kwargs_eig)*

Condition number of linear operator.

Return an estimate of the condition number of the linear operator as the ratio of the largest and lowest estimated eigenvalues.

**Parameters**

- **uselobpcg** [bool, optional] Use scipy.sparse.linalg.lobpcg to compute eigenvalues.
**kwargs_eig
Arbitrary keyword arguments for `scipy.sparse.linalg.eigs`, `scipy.sparse.linalg.eigsh`, or `scipy.sparse.linalg.lobpcg`

** Returns **


** Notes **

The condition number of a matrix (or linear operator) can be estimated as the ratio of the largest and lowest estimated eigenvalues:

\[ k = \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}} \]

The condition number provides an indication of the rate at which the solution of the inversion of the linear operator \( A \) will change with respect to a change in the data \( y \).

Thus, if the condition number is large, even a small error in \( y \) may cause a large error in \( x \). On the other hand, if the condition number is small then the error in \( x \) is not much bigger than the error in \( y \). A problem with a low condition number is said to be well-conditioned, while a problem with a high condition number is said to be ill-conditioned.

** conj() **

Complex conjugate operator

** Returns **

conjop [pylops.LinearOperator] Complex conjugate operator

** apply_columns(cols) **

Apply subset of columns of operator

This method can be used to wrap a LinearOperator and mimic the action of a subset of columns of the operator on a reduced model in forward mode, and retrieve only the result of a subset of rows in adjoint mode.

Note that unless the operator has explicit=True, this is not optimal as the entire forward and adjoint passes of the original operator will have to be performed. It can however be useful for the implementation of solvers such as Orthogonal Matching Pursuit (OMP) that iteratively build a solution by evaluate only a subset of the columns of the operator.

** Parameters **

cols [list] Columns to be selected

** Returns **

colop [pylops.LinearOperator] Apply column operator

** Examples using pylops.LinearOperator **

- sphx_glr_gallery_plot_sliding.py
- sphx_glr_gallery_plot_avo.py
- sphx_glr_gallery_plot_bilinear.py
- sphx_glr_gallery_plot_causalintegration.py
- sphx_glr_gallery_plot_convolve.py
• sphx_glr_gallery_plot_derivative.py
• sphx_glr_gallery_plot_diagonal.py
• sphx_glr_gallery_plot_flip.py
• sphx_glr_gallery_plot_fft.py
• sphx_glr_gallery_plot_identity.py
• sphx_glr_gallery_plot_linearregr.py
• sphx_glr_gallery_plot_ista.py
• sphx_glr_gallery_plot_matrixmult.py
• sphx_glr_gallery_plot_mdc.py
• sphx_glr_gallery_plot_stacking.py
• sphx_glr_gallery_plot_pad.py
• sphx_glr_gallery_plot_phaseshift.py
• sphx_glr_gallery_plot_regr.py
• sphx_glr_gallery_plot_prestack.py
• sphx_glr_gallery_plot_restriction.py
• sphx_glr_gallery_plot_roll.py
• sphx_glr_gallery_plot_seislet.py
• sphx_glr_gallery_plot_sum.py
• sphx_glr_gallery_plot_symmetrize.py
• sphx_glr_gallery_plot_tvreg.py
• sphx_glr_gallery_plot_transpose.py
• sphx_glr_gallery_plot_wavelet.py
• sphx_glr_gallery_plot_wave.py
• sphx_glr_gallery_plot_zero.py

01. The LinearOperator
02. The Dot-Test
03. Solvers
04. Bayesian Inversion
05. Image deblurring
06. 2D Interpolation
07. Post-stack inversion
08. Pre-stack (AVO) inversion
09. Multi-Dimensional Deconvolution
12. Seismic regularization
14. Seismic wavefield decomposition
15. Least-squares migration
pylops.FunctionOperator

class pylops.FunctionOperator(f, *args, **kwargs)

Function Operator.

Simple wrapper to functions for forward \( f \) and adjoint \( f^c \) multiplication.

Functions \( f \) and \( f^c \) are such that \( f : \mathbb{F}^m \to \mathbb{F}^n \) and \( f^c : \mathbb{F}^n \to \mathbb{F}^m \) where \( \mathbb{F} \) is the appropriate underlying type (e.g., \( \mathbb{R} \) for real or \( \mathbb{C} \) for complex).

FunctionOperator can be called in the following ways: FunctionOperator\((f, n)\), FunctionOperator\((f, n, m)\), FunctionOperator\((f, f^c, n)\), and FunctionOperator\((f, f^c, n, m)\).

The first two methods can only be used for forward modelling and will return \( \text{NotImplementedError} \) if the adjoint is called. The first and third method assume the matrix (or matrices) to be square. All methods can be called with the \textit{dtype} keyword argument.

Parameters

- \( f \) [callable] Function for forward multiplication.
- \( f^c \) [callable, optional] Function for adjoint multiplication.
- \( n \) [int, optional] Number of rows (length of data vector).
- \( m \) [int, optional] Number of columns (length of model vector).
- \textit{dtype} [str, optional] Type of elements in input array.

Examples

```python
>>> from pylops.basicoperators import FunctionOperator
>>> def forward(v):
...     return np.array([2*v[0], 3*v[1]])
...     
>>> A = FunctionOperator(forward, 2)
>>> A
<2x2 FunctionOperator with dtype=float64>
>>> A.matvec(np.ones(2))
array([2., 3.])
>>> A @ np.ones(2)
array([2., 3.])
```

Attributes

- \textit{shape} [tuple] Operator shape \([n \times m]\)
- \textit{explicit} [bool] Operator contains a matrix that can be solved explicitly \((\text{True})\) or not \((\text{False})\)

Methods

- \_init\_(\(f, *\text{args}, **\text{kwargs}\)) Initialize this \textit{LinearOperator}.
- \textit{adjoint}() Hermitian adjoint.

Continued on next page
Table 3 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>apply_columns(cols)</code></td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td><code>cond([uselobpcg])</code></td>
<td>Condition number of linear operator.</td>
</tr>
<tr>
<td><code>conj()</code></td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td><code>div(y, niter)</code></td>
<td>Solve the linear problem $y = Ax$.</td>
</tr>
<tr>
<td><code>dot(x)</code></td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>eigs([neigs, symmetric, niter, uselobpcg])</code></td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td><code>matmat(X)</code></td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>matvec(x)</code></td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>rmatmat(X)</code></td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>rmatvec(x)</code></td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>todense()</code></td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td><code>tospase()</code></td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td><code>transpose()</code></td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

Basic operators

<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MatrixMult(A, dims, dtype)</code></td>
<td>Matrix multiplication.</td>
</tr>
<tr>
<td><code>Identity(N[, M, dtype, inplace])</code></td>
<td>Identity operator.</td>
</tr>
<tr>
<td><code>Zero(N[, M, dtype])</code></td>
<td>Zero operator.</td>
</tr>
<tr>
<td><code>Diagonal(diag[, dims, dir, dtype])</code></td>
<td>Diagonal operator.</td>
</tr>
<tr>
<td><code>Transpose(dims, axes[, dtype])</code></td>
<td>Transpose operator.</td>
</tr>
<tr>
<td><code>Flip(N[, dims, dir, dtype])</code></td>
<td>Flip along an axis.</td>
</tr>
<tr>
<td><code>Roll(N[, dims, dir, shift, dtype])</code></td>
<td>Roll along an axis.</td>
</tr>
<tr>
<td><code>Pad(dims, pad[, dtype])</code></td>
<td>Pad operator.</td>
</tr>
<tr>
<td><code>Sum(dims, dir[, dtype])</code></td>
<td>Sum operator.</td>
</tr>
<tr>
<td><code>Symmetrize(N[, dims, dir, dtype])</code></td>
<td>Symmetrize along an axis.</td>
</tr>
<tr>
<td><code>Restriction(M, iava[, dims, dir, dtype, inplace])</code></td>
<td>Restriction (or sampling) operator.</td>
</tr>
<tr>
<td><code>Regression(taxis, order[, dtype])</code></td>
<td>Polynomial regression.</td>
</tr>
<tr>
<td><code>LinearRegression(taxis[, dtype])</code></td>
<td>Linear regression.</td>
</tr>
<tr>
<td><code>CausalIntegration(N[, dims, dir, sampling, ...])</code></td>
<td>Causal integration.</td>
</tr>
<tr>
<td><code>Spread(dims, dimsd[, table, dtype, fh, ...])</code></td>
<td>Spread operator.</td>
</tr>
<tr>
<td><code>VStack(ops[, dtype])</code></td>
<td>Vertical stacking.</td>
</tr>
<tr>
<td><code>HStack(ops[, dtype])</code></td>
<td>Horizontal stacking.</td>
</tr>
<tr>
<td><code>Block(ops[, dtype])</code></td>
<td>Block operator.</td>
</tr>
<tr>
<td><code>BlockDiag(ops[, dtype])</code></td>
<td>Block-diagonal operator.</td>
</tr>
<tr>
<td><code>Kronecker(Op1, Op2[, dtype])</code></td>
<td>Kronecker operator.</td>
</tr>
</tbody>
</table>

**pylops.MatrixMult**

```python
class pylops.MatrixMult(A, dims=None, dtype='float64')
```

Matrix multiplication.

Simple wrapper to `numpy.dot` and `numpy.vdot` for an input matrix `A`.

**Parameters**

- `dims` [tuple, optional] Number of samples for each other dimension of model (model/data will be reshaped and `A` applied multiple times to each column of the model/data).
- `dtype` [str, optional] Type of elements in input array.
Attributes

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)
- **complex** [bool] Matrix has complex numbers (True) or not (False)

Methods

```python
__init__(A[, dims, dtype]) Initialize this LinearOperator.
adjoint() Hermitian adjoint.
apply_columns(cols) Apply subset of columns of operator
cond([uselobpcg]) Condition number of linear operator.
conj() Complex conjugate operator
div(y[, niter]) Solve the linear problem \( y = Ax \).
dot(x) Matrix-matrix or matrix-vector multiplication.
eigs([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
inv() Return the inverse of \( A \).
matmat(X) Matrix-matrix multiplication.
matvec(x) Matrix-vector multiplication.
rmatmat(X) Adjoint matrix-matrix multiplication.
rmatvec(x) Adjoint matrix-vector multiplication.
todense() Return dense matrix.
tosparse() Return sparse matrix.
transpose() Transpose this linear operator.
```

**inv()**

Return the inverse of \( A \).

**Returns**

- \( A_{\text{inv}} \) [numpy.ndarray] Inverse matrix.

Examples using `pylops.MatrixMult`

- sphx_glr_gallery_plot_ista.py
- sphx_glr_gallery_plot_matrixmult.py
- sphx_glr_gallery_plot_stacking.py
- 02. The Dot-Test
- 03. Solvers
- 07. Post-stack inversion
- 08. Pre-stack (AVO) inversion

**pylops.Identity**

```python
class pylops.Identity (N, M=None, dtype='float64', inplace=True)
```

Identity operator.
Simply move model to data in forward model and vice versa in adjoint mode if $M = N$. If $M > N$ removes last $M - N$ elements from model in forward and pads with 0 in adjoint. If $N > M$ removes last $N - M$ elements from data in adjoint and pads with 0 in forward.

**Parameters**

- $N$ [int] Number of samples in data (and model, if $M$ is not provided).
- $M$ [int, optional] Number of samples in model.
- dtype [str, optional] Type of elements in input array.
- inplace [bool, optional] Work inplace (True) or make a new copy (False). By default, data is a reference to the model (in forward) and model is a reference to the data (in adjoint).

**Notes**

For $M = N$, an *Identity* operator simply moves the model $x$ to the data $y$ in forward mode and vice versa in adjoint mode:

$$y_i = x_i \quad \forall i = 1, 2, ..., N$$

or in matrix form:

$$y = Ix = x$$

and

$$x = Iy = y$$

For $M > N$, the *Identity* operator takes the first $M$ elements of the model $x$ into the data $y$ in forward mode

$$y_i = x_i \quad \forall i = 1, 2, ..., N$$

and all the elements of the data $y$ into the first $M$ elements of model in adjoint mode (other elements are 0):

$$x_i = y_i \quad \forall i = 1, 2, ..., M$$

$$x_i = 0 \quad \forall i = M + 1, ..., N$$

**Attributes**

- shape [tuple] Operator shape
- explicit [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>([N[, M, dtype, inplace]])</td>
<td>Initialize this LinearOperator.</td>
</tr>
<tr>
<td>adjoint()</td>
<td>Hermitian adjoint.</td>
</tr>
<tr>
<td>apply_columns(cols)</td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td>cond([uselobpcg])</td>
<td>Condition number of linear operator.</td>
</tr>
<tr>
<td>conj()</td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td>div(y[, niter])</td>
<td>Solve the linear problem $y = Ax$.</td>
</tr>
<tr>
<td>dot(x)</td>
<td>Matrix-matrix or matrix-vector multiplication.</td>
</tr>
<tr>
<td>eigs([neigs, symmetric, niter, uselobpcg])</td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 6 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matmat(X)</td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td>matvec(x)</td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td>rmatmat(X)</td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td>rmatvec(x)</td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td>todense()</td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

**Examples using pylops.Identity**

- sphx_glr_gallery_plot_identity.py
- sphx_glr_gallery_plot_stacking.py
- sphx_glr_gallery_plot_tvreg.py
- 04. Bayesian Inversion

**pylops.Zero**

```python
class pylops.Zero(N=M=None, dtype='float64')
```

Zero operator.

Transform model into array of zeros of size $N$ in forward and transform data into array of zeros of size $N$ in adjoint.

**Parameters**

- `N` [int] Number of samples in data (and model in M is not provided).
- `M` [int, optional] Number of samples in model.
- `dtype` [str, optional] Type of elements in input array.

**Notes**

An Zero operator simply creates a null data vector $y$ in forward mode:

$$0x = 0_N$$

and a null model vector $x$ in forward mode:

$$0y = 0_M$$

**Attributes**

- `shape` [tuple] Operator shape
- `explicit` [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

```python
__init__(N[, M, dtype]) Initialize this LinearOperator.
```
Table 7 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjoint()</td>
<td>Hermitian adjoint.</td>
</tr>
<tr>
<td>apply_columns(cols)</td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td>cond([uselobpcg])</td>
<td>Condition number of linear operator.</td>
</tr>
<tr>
<td>conj()</td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td>div(y[, niter])</td>
<td>Solve the linear problem $y = Ax$.</td>
</tr>
<tr>
<td>dot(x)</td>
<td>Matrix-matrix or matrix-vector multiplication.</td>
</tr>
<tr>
<td>eigs([neigs, symmetric, niter, uselobpcg])</td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td>matmat(X)</td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td>rmatmat(X)</td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td>rmatvec(x)</td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td>todense()</td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

Examples using pylops.Zero

- sphx_glr_gallery_plot_zero.py

pylops.Diagonal

class pylops.Diagonal(diag, dims=None, dir=0, dtype='float64')

Diagonal operator.

Applies element-wise multiplication of the input vector with the vector `diag` in forward and with its complex conjugate in adjoint mode.

This operator can also broadcast; in this case the input vector is reshaped into its dimensions `dims` and the element-wise multiplication with `diag` is performed on the direction `dir`. Note that the vector `diag` will need to have size equal to `dims[dir].`

Parameters

- `diag` [numpy.ndarray] Vector to be used for element-wise multiplication.
- `dims` [list, optional] Number of samples for each dimension (None if only one dimension is available)
- `dir` [int, optional] Direction along which multiplication is applied.
- `dtype` [str, optional] Type of elements in input array.

Notes

Element-wise multiplication between the model $x$ and/or data $y$ vectors and the array $d$ can be expressed as

$$y_i = d_i x_i \quad \forall i = 1, 2, ..., N$$

This is equivalent to a matrix-vector multiplication with a matrix containing the vector $d$ along its main diagonal.

For real-valued `diag`, the Diagonal operator is self-adjoint as the adjoint of a diagonal matrix is the diagonal matrix itself. For complex-valued `diag`, the adjoint is equivalent to the element-wise multiplication with the complex conjugate elements of `diag`.

Attributes
shape [tuple] Operator shape

explicit [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

```
__init__ (diag[, dims, dir, dtype]) Initialize this LinearOperator.
adjoint () Hermitian adjoint.
apply_columns(cols) Apply subset of columns of operator.
cond([uselobpcg]) Condition number of linear operator.
conj () Complex conjugate operator.
div(y[, niter]) Solve the linear problem $y = Ax$.
dot(x) Matrix-matrix or matrix-vector multiplication.
eigs([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
matmat(X) Matrix-matrix multiplication.
matrix() Return diagonal matrix as dense numpy.ndarray.
matvec(x) Matrix-vector multiplication.
rmatmat(X) Adjoint matrix-matrix multiplication.
rmatvec(x) Adjoint matrix-vector multiplication.
todense() Return dense matrix.
tosparse() Return sparse matrix.
transpose () Transpose this linear operator.
```

density()

Return diagonal matrix as dense numpy.ndarray

Returns

densemat [numpy.ndarray] Dense matrix.

Examples using pylops.Diagonal

- sphx_glr_gallery_plot_diagonal.py
- 01. The LinearOperator
- 02. The Dot-Test

pylops.Transpose

```
class pylops.Transpose (dims, axes, dtype='float64')

Transpose operator.

Transpose axes of a multi-dimensional array. This operator works with flattened input model (or data), which are however multi-dimensional in nature and will be reshaped and treated as such in both forward and adjoint modes.

Parameters

dims [tuple, optional] Number of samples for each dimension
axes [tuple, optional] Direction along which transposition is applied
dtype [str, optional] Type of elements in input array
```
PyLops

Raises

**ValueError** If axes contains repeated dimensions (or a dimension is missing)

Notes

The Transpose operator reshapes the input model into a multi-dimensional array of size dims and transposes (or swaps) its axes as defined in axes.

Similarly, in adjoint mode the data is reshaped into a multi-dimensional array whose size is a permuted version of dims defined by axes. The array is then rearranged into the original model dimensions dims.

Attributes

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

- **__init__**(dims, axes[, dtype]) Initialize this LinearOperator.
- **adjoint**() Hermitian adjoint.
- **apply_columns**(cols) Apply subset of columns of operator
- **cond**([uselobpcg]) Condition number of linear operator.
- **conj**() Complex conjugate operator
- **div**(y[, niter]) Solve the linear problem \( y = Ax \).
- **dot**(x) Matrix-matrix or matrix-vector multiplication.
- **eigs**([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
- **matmat**(X) Matrix-matrix multiplication.
- **matvec**(x) Matrix-vector multiplication.
- **rmatmat**(X) Adjoint matrix-matrix multiplication.
- **rmatvec**(x) Adjoint matrix-vector multiplication.
- **todense**() Return dense matrix.
- **tosparse**() Return sparse matrix.
- **transpose**() Transpose this linear operator.

Examples using pylops.Transpose

- sphx_glr_gallery_plot_transpose.py

pylops.Flip

**class** pylops.Flip**(N, dims=None, dir=0, dtype='float64')**

Flip along an axis.

Flip a multi-dimensional array along a specified direction dir.

**Parameters**

- **N** [int] Number of samples in model.
- **dims** [list, optional] Number of samples for each dimension (None if only one dimension is available)
- **dir** [int, optional] Direction along which flipping is applied.
**dtype** [str, optional] Type of elements in input array.

**Notes**

The Flip operator flips the input model (and data) along any chosen direction. For simplicity, given a one dimensional array, in forward mode this is equivalent to:

\[ y[i] = x[N - i] \quad \forall i = 0, 1, 2, ..., N - 1 \]

where \( N \) is the length of the input model. As this operator is self-adjoint, \( x \) and \( y \) in the equation above are simply swapped in adjoint mode.

**Attributes**

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

- **__init__**(N[, dims, dir, dtype]) Initialize this LinearOperator.
- **adjoint**() Hermitian adjoint.
- **apply_columns**(cols) Apply subset of columns of operator
- **cond**(uselobpcg) Condition number of linear operator.
- **conj**() Complex conjugate operator
- **div**(y[, niter]) Solve the linear problem \( y = Ax \).
- **dot**(x) Matrix-matrix or matrix-vector multiplication.
- **eigs**(neigs, symmetric, niter, uselobpcg) Most significant eigenvalues of linear operator.
- **matmat**(X) Matrix-matrix multiplication.
- **matvec**(x) Matrix-vector multiplication.
- **rmatmat**(X) Adjoint matrix-matrix multiplication.
- **rmatvec**(x) Adjoint matrix-vector multiplication.
- **todense**() Return dense matrix.
- **tosparse**() Return sparse matrix.
- **transpose**() Transpose this linear operator.

**Examples using pylops.Flip**

- sphx_glr_gallery_plot_flip.py

**pylops.Roll**

**class pylops.Roll**(N, dims=None, dir=0, shift=1, dtype='float64')

Roll along an axis.

Roll a multi-dimensional array along a specified direction \( \text{dir} \) for a chosen number of samples \( \text{shift} \).

**Parameters**

- **N** [int] Number of samples in model.
- **dims** [list, optional] Number of samples for each dimension (None if only one dimension is available)
dir  [int, optional] Direction along which rolling is applied.
shift [int, optional] Number of samples by which elements are shifted
dtype [str, optional] Type of elements in input array.

Notes

The Roll operator is a thin wrapper around numpy.roll and shifts elements in a multi-dimensional array along a specified direction for a chosen number of samples.

Attributes

shape [tuple] Operator shape
explicit [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

__init__([N[, dims, dir, shift, dtype]]) Initialize this LinearOperator.
adjoint() Hermitian adjoint.
apply_columns(cols) Apply subset of columns of operator
cond([uselobpcg]) Condition number of linear operator.
conj() Complex conjugate operator
div(y[, niter]) Solve the linear problem $y = Ax$.
dot(x) Matrix-matrix or matrix-vector multiplication.
eigs([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
matmat(X) Matrix-matrix multiplication.
matvec(x) Matrix-vector multiplication.
rmatmat(X) Adjoint matrix-matrix multiplication.
rmatvec(x) Adjoint matrix-vector multiplication.
todense() Return dense matrix.
tosparse() Return sparse matrix.
transpose() Transpose this linear operator.

Examples using pylops.Roll

- sphx_glr_gallery_plot_roll.py

pylops.Pad

class pylops.Pad(dims, pad, dtype='float64')

Pad operator.

Zero-pad model in forward model and extract non-zero subsequence in adjoint. Padding can be performed in one or multiple directions to any multi-dimensional input arrays.

Parameters

dims [int or tuple] Number of samples for each dimension

pad [tuple] Number of samples to pad. If dims is a scalar, pad is a single tuple (pad_in, pad_end). If dims is a tuple, pad is a tuple of tuples where each inner tuple contains the number of samples to pad in each dimension

3.6. PyLops API
**dtype** [str, optional] Type of elements in input array.

**Raises**

**ValueError** If any element of pad is negative.

**Notes**

Given an array of size $N$, the Pad operator simply adds $pad_{in}$ at the start and $pad_{end}$ at the end in forward mode:

$$y_i = x_{i-pad_{in}} \quad \forall i = pad_{in}, pad_{in} + 1, ..., pad_{in} + N - 1$$

and $y_i = 0 \quad \forall i = 0, ..., pad_{in} - 1, pad_{in} + N - 1, ..., N + pad_{in} + pad_{end}$

In adjoint mode, values from $pad_{in}$ to $N - pad_{end}$ are extracted from the data:

$$x_i = y_{pad_{in}+i} \quad \forall i = 0, N - 1$$

**Attributes**

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Initialize this LinearOperator.</td>
</tr>
<tr>
<td>adjoint()</td>
<td>Hermitian adjoint.</td>
</tr>
<tr>
<td>apply_columns(cols)</td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td>cond([uselobpcg])</td>
<td>Condition number of linear operator.</td>
</tr>
<tr>
<td>conj()</td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td>div(y[, niter])</td>
<td>Solve the linear problem $y = Ax$.</td>
</tr>
<tr>
<td>dot(x)</td>
<td>Matrix-matrix or matrix-vector multiplication.</td>
</tr>
<tr>
<td>eigs([neigs, symmetric, niter, uselobpcg])</td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td>matmat(X)</td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td>matvec(x)</td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td>rmatmat(X)</td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td>rmatvec(x)</td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td>todense()</td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

**Examples using pylops.Pad**

- sphx_glr_gallery_plot_pad.py

**pylops.Sum**

```python
class pylops.Sum(dims, dir, dtype='float64')
```

Sum operator.
Sum along an axis of a multi-dimensional array (at least 2 dimensions are required) in forward model, and spread along the same axis in adjoint mode.

**Parameters**

- `dims` [tuple] Number of samples for each dimension
- `dir` [int] Direction along which summation is performed.
- `dtype` [str, optional] Type of elements in input array.

**Notes**

Given a two dimensional array, the `Sum` operator re-arranges the input model into a multi-dimensional array of size `dims` and sums values along direction `dir`:

\[ y_j = \sum_i x_{i,j} \]

In adjoint mode, the data is spread along the same direction:

\[ x_{i,j} = y_j \quad \forall i = 0, N-1 \]

**Attributes**

- `shape` [tuple] Operator shape
- `explicit` [bool] Operator contains a matrix that can be solved explicitly (`True`) or not (`False`)

**Methods**

- `__init__(dims, dir[, dtype])` Initialize this LinearOperator.
- `adjoint()` Hermitian adjoint.
- `apply_columns(cols)` Apply subset of columns of operator
- `cond([uselobpcg])` Condition number of linear operator.
- `conj()` Complex conjugate operator
- `div(y[, niter])` Solve the linear problem `y = Ax`.
- `dot(x)` Matrix-matrix or matrix-vector multiplication.
- `eigs([neigs, symmetric, niter, uselobpcg])` Most significant eigenvalues of linear operator.
- `matmat(X)` Matrix-matrix multiplication.
- `matvec(x)` Matrix-vector multiplication.
- `rmatmat(X)` Adjoint matrix-matrix multiplication.
- `rmatvec(x)` Adjoint matrix-vector multiplication.
- `todense()` Return dense matrix.
- `tosparse()` Return sparse matrix.
- `transpose()` Transpose this linear operator.

**Examples using pylops.Sum**

- `sphx_glr_gallery_plot_sum.py`
**pylops.Symmetrize**

```python
class pylops.Symmetrize(N, dims=None, dir=0, dtype='float64')
    Symmetrize along an axis.
    Symmetrize a multi-dimensional array along a specified direction `dir`.

    Parameters
    N [int] Number of samples in model. Symmetric data has 2N − 1 samples
    dims [list, optional] Number of samples for each dimension (None if only one dimension is available)
    dir [int, optional] Direction along which symmetrization is applied
    dtype [str, optional] Type of elements in input array

    Notes
    The Symmetrize operator constructs a symmetric array given an input model in forward mode, by pre-pending the input model in reversed order.
    For simplicity, given a one dimensional array, the forward operation can be expressed as:
    
    $$ y[i] = \begin{cases} 
    x[i-N], & i \geq N \\
    x[N-i], & \text{otherwise} 
    \end{cases} $$
    
    for $i = 0, 1, 2, ..., 2N - 2$, where $N$ is the length of the input model.
    In adjoint mode, the Symmetrize operator assigns the sums of the elements in position $N - i$ and $N + i$ to position $i$ as follows:
    
    apart from the central sample where $x[0] = y[N]$.

    Attributes
    shape [tuple] Operator shape
    explicit [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

    Methods
    __init__(N[, dims, dir, dtype]) Initialize this LinearOperator.
    adjoint() Hermitian adjoint.
    apply_columns(cols) Apply subset of columns of operator
    cond([uselobpcg]) Condition number of linear operator.
    conj() Complex conjugate operator
    div(y[, niter]) Solve the linear problem $y = Ax$.
    dot(x) Matrix-matrix or matrix-vector multiplication.
    eigs([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
    matmat(X) Matrix-matrix multiplication.
    matvec(x) Matrix-vector multiplication.
    rmatmat(X) Adjoint matrix-matrix multiplication.
    rmatvec(x) Adjoint matrix-vector multiplication.
```

Continued on next page
Table 14 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmatvec(x)</td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td>todense()</td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

**Examples using `pylops.Symmetrize`**

- `sphx_glr_gallery_plot_symmetrize.py`
- `sphx_glr_gallery_plot_wavest.py`

**`pylops.Restriction`**

```python
class pylops.Restriction(M, iava, dims=None, dir=0, dtype='float64', inplace=True)
```

Restriction (or sampling) operator.

Extract subset of values from input vector at locations `iava` in forward mode and place those values at locations `iava` in an otherwise zero vector in adjoint mode.

**Parameters**

- `M [int]` Number of samples in model.
- `iava [list or numpy.ndarray]` Integer indices of available samples for data selection.
- `dims [list]` Number of samples for each dimension (None if only one dimension is available)
- `dir [int, optional]` Direction along which restriction is applied.
- `dtype [str, optional]` Type of elements in input array.
- `inplace [bool, optional]` Work inplace (True) or make a new copy (False). By default, data is a reference to the model (in forward) and model is a reference to the data (in adjoint).

**See also:**

- `pylops.signalprocessing.Interp` Interpolation operator

**Notes**

Extraction (or sampling) of a subset of $N$ values at locations `iava` from an input (or model) vector $x$ of size $M$ can be expressed as:

$$y_i = x_{l_i} \quad \forall i = 1, 2, ..., N$$

where $l = [l_1, l_2, ..., l_N]$ is a vector containing the indeces of the original array at which samples are taken.

Conversely, in adjoint mode the available values in the data vector $y$ are placed at locations $l = [l_1, l_2, ..., l_M]$ in the model vector:

$$x_{l_i} = y_i \quad \forall i = 1, 2, ..., N$$

and $x_j = 0$ if $j \neq l_i$ (i.e., at all other locations in input vector).

**Attributes**

- `shape [tuple]` Operator shape
- `explicit [bool]` Operator contains a matrix that can be solved explicitly (True) or not (False)
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(M, iava[, dims, dir, dtype, inplace])</td>
<td>Initialize this LinearOperator.</td>
</tr>
<tr>
<td>adjoint()</td>
<td>Hermitian adjoint.</td>
</tr>
<tr>
<td>apply_columns(cols)</td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td>cond([uselobpcg])</td>
<td>Condition number of linear operator</td>
</tr>
<tr>
<td>conj()</td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td>div(y, niter)</td>
<td>Solve the linear problem $y = Ax$.</td>
</tr>
<tr>
<td>dot(x)</td>
<td>Matrix-matrix or matrix-vector multiplication.</td>
</tr>
<tr>
<td>eigs([neigs, symmetric, niter, uselobpcg])</td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td>mask(x)</td>
<td>Apply mask to input signal returning a signal of same size with values at iava locations and 0 at other locations</td>
</tr>
<tr>
<td>matmat(X)</td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td>matvec(x)</td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td>rmatmat(X)</td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td>rmatvec(x)</td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td>todense()</td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

### mask(x)

Apply mask to input signal returning a signal of same size with values at iava locations and 0 at other locations.

**Parameters**

- **x** [numpy.ndarray] Input array (can be either flattened or not)

**Returns**

- **y** [numpy.ma.core.MaskedArray] Masked array.

### Examples using pylops.Restriction

- sphx_glr_gallery_plot_restriction.py
- sphx_glr_gallery_plot_tvreg.py

### pylops.Regression

**class pylops.Regression(taxis, order, dtype=’float64’)**

Polynomial regression.

Creates an operator that applies polynomial regression to a set of points. Values along the t-axis must be provided while initializing the operator. The coefficients of the polynomial regression form the model vector to be provided in forward mode, while the values of the regression curve shall be provided in adjoint mode.
Parameters

- **taxis** [numpy.ndarray] Elements along the t-axis.
- **order** [int] Order of the regressed polynomial.
- **dtype** [str, optional] Type of elements in input array.

Raises

- **TypeError** If *t* is not numpy.ndarray.

See also:

- **LinearRegression** Linear regression

Notes

The Regression operator solves the following problem:

\[ y_i = \sum_{n=0}^{order} x_n t_i^n \quad \forall i = 1, 2, \ldots, N \]

where \( N \) represents the order of the chosen polynomial. We can express this problem in a matrix form

\[ y = Ax \]

where

\[ y = [y_1, y_2, \ldots, y_N]^T, \quad x = [x_0, x_1, \ldots, x_{\text{order}}]^T \]

and

\[ A = \begin{bmatrix} 1 & t_1 & t_1^2 & \cdots & t_1^{\text{order}} \\ 1 & t_2 & t_2^2 & \cdots & t_2^{\text{order}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_N & t_N^2 & \cdots & t_N^{\text{order}} \end{bmatrix} \]

Attributes

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

- **__init__**(taxis, order[, dtype]) Initialize this LinearOperator.
- **adjoint**() Hermitian adjoint.
- **apply**(t, x) Return values along y-axis given certain t location(s) along t-axis and regression coefficients x
- **apply_columns**(cols) Apply subset of columns of operator
- **cond**(uselobpcg) Condition number of linear operator.
- **conj**() Complex conjugate operator
- **div**(y[, niter]) Solve the linear problem \( y = Ax \).
- **dot**(x) Matrix-matrix or matrix-vector multiplication.
- **eigs**(neigs, symmetric, niter, uselobpcg) Most significant eigenvalues of linear operator.

Continued on next page
Table 16 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matmat(X)</td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td>matvec(x)</td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td>rmatmat(X)</td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td>rmatvec(x)</td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td>todense()</td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

**apply** \((t, x)\)

Return values along y-axis given certain \(t\) location(s) along t-axis and regression coefficients \(x\)

**Parameters**

- **taxis** \([\text{numpy.ndarray}]\) Elements along the t-axis.
- **x** \([\text{numpy.ndarray}]\) Regression coefficients
- **dtype** \([\text{str}, \text{optional}]\)

**Returns**

- **y** \([\text{numpy.ndarray}]\) Values along y-axis

**Examples using pylops.LinearRegression**

- sphx_glr_gallery_plot_linearregr.py
- sphx_glr_gallery_plot_regr.py

**pylops.LinearRegression**

**pylops.LinearRegression** \((taxis, dtype='float64')\)

Linear regression.

Creates an operator that applies linear regression to a set of points. Values along the t-axis must be provided while initializing the operator. Intercept and gradient form the model vector to be provided in forward mode, while the values of the regression line curve shall be provided in adjoint mode.

**Parameters**

- **taxis** \([\text{numpy.ndarray}]\) Elements along the t-axis.
- **dtype** \([\text{str}, \text{optional}]\) Type of elements in input array.

**Raises**

- **TypeError** If \(t\) is not \(\text{numpy.ndarray}\).

**See also:**

- **Regression** Polynomial regression
Notes

The LinearRegression operator solves the following problem:

\[ y_i = x_0 + x_1 t_i \quad \forall i = 1, 2, ..., N \]

We can express this problem in a matrix form

\[ y = Ax \]

where

\[ y = [y_1, y_2, ..., y_N]^T, \quad x = [x_0, x_1]^T \]

and

\[ A = \begin{bmatrix}
1 & t_1 \\
1 & t_2 \\
.. & .. \\
1 & t_N \\
\end{bmatrix} \]

Note that this is a particular case of the `pylops.Regression` operator and it is in fact just a lazy call of that operator with `order=1`.

Attributes

- `shape [tuple]` Operator shape
- `explicit [bool]` Operator contains a matrix that can be solved explicitly (True) or not (False)

Examples using `pylops.LinearRegression`

- sphx_glr_gallery_plot_linearregr.py

`pylops.CausalIntegration`

```python
class pylops.CausalIntegration(N, dims=None, dir=-1, sampling=1, halfcurrent=True, dtype='float64')
```

Causal integration.

Apply causal integration to a multi-dimensional array along `dir` axis.

Parameters

- `N [int]` Number of samples in model.
- `dims [list, optional]` Number of samples for each dimension (None if only one dimension is available)
- `dir [int, optional]` Direction along which smoothing is applied.
- `sampling [float, optional]` Sampling step \(dx\).
- `halfcurrent [float, optional]` Add half of current value (True) or the entire value (False)
- `dtype [str, optional]` Type of elements in input array.
Notes

The CausalIntegration operator applies a causal integration to any chosen direction of a multi-dimensional array. For simplicity, given a one dimensional array, the causal integration is:

\[ y(t) = \int x(t) dt \]

which can be discretised as:

\[ y[i] = \sum_{j=0}^{i} x[j] dt \]

or

\[ y[i] = \left( \sum_{j=0}^{i-1} x[j] + 0.5x[i] \right) dt \]

where \( dt \) is the sampling interval. In our implementation, the choice to add \( x[i] \) or just \( 0.5x[i] \) is made by selecting the halfcurrent parameter.

Note that the integral of a signal has no unique solution, as any constant \( c \) can be added to \( y \), for example if \( x(t) = t^2 \) the resulting integration is:

\[ y(t) = \int t^2 dt = \frac{t^3}{3} + c \]

If we apply a first derivative to \( y \) we in fact obtain:

\[ x(t) = \frac{dy}{dt} = t^2 \]

no matter the choice of \( c \).

Attributes

- shape [tuple] Operator shape
- explicit [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

```python
__init__([N, dims, dir, sampling, ...]) Initialize this LinearOperator.
adjoint() Hermitian adjoint.
apply_columns(cols) Apply subset of columns of operator
cond([uselobpcg]) Condition number of linear operator.
conj() Complex conjugate operator
div(y[, niter]) Solve the linear problem \( y = Ax \).
dot(x) Matrix-matrix or matrix-vector multiplication.
eigs([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
matmat(X) Matrix-matrix multiplication.
matvec(x) Matrix-vector multiplication.
rmatmat(X) Adjoint matrix-matrix multiplication.
rmatvec(x) Adjoint matrix-vector multiplication.
todense() Return dense matrix.
tosparse() Return sparse matrix.
transpose() Transpose this linear operator.
```
Examples using `pylops.CausalIntegration`

- sphx_glr_gallery_plot_causalintegration.py

**pylops.Spread**

```python
class pylops.Spread(dims, dims0, table=None, dtable=None, fh=None, interp=False, engine='numpy', dtype='float64')
```

Spread operator.

Spread values from the input model vector arranged as a 2-dimensional array of size $[n_x \times n_t]$ into the data vector of size $[n_x \times n_t]$. Spreading is performed along parametric curves provided as look-up table of pre-computed indices (table) or computed on-the-fly using a function handle (fh).

In adjoint mode, values from the data vector are instead stacked along the same parametric curves.

**Parameters**

- **`dims` [tuple]** Dimensions of model vector (vector will be reshaped internally into a two-dimensional array of size $[n_x \times n_t]$, where the first dimension is the spreading/stacking direction).
- **`dmsg` [tuple]** Dimensions of model vector (vector will be reshaped internal into a two-dimensional array of size $[n_x \times n_t]$).
- **`table` [np.ndarray, optional]** Look-up table of indeces of size $[n_x \times n_t \times n_x]$ (if None use function handle fh).
- **`dtable` [np.ndarray, optional]** Look-up table of decimals remainders for linear interpolation of size $[n_x \times n_t \times n_x]$ (if None use function handle fh).
- **`fh` [np.ndarray, optional]** Function handle that returns an index (and a fractional value in case of interp=True) to be used for spreading/stacking given indices in $x$ and $t$ axes (if None use look-up table table).
- **`interp` [bool, optional]** Apply linear interpolation (True) or nearest interpolation (False) during stacking/spreading along parametric curve. To be used only if engine='numba', inferred directly from the number of outputs of fh for engine='numpy'.
- **`engine` [str, optional]** Engine used for fft computation (numpy or numba). Note that numba can only be used when providing a look-up table.
- **`dtype` [str, optional]** Type of elements in input array.

**Raises**

- **`KeyError`** If engine is neither numpy nor numba.
- **`NotImplementedError`** If both table and fh are not provided.
- **`ValueError`** If table has shape different from $[n_x \times n_t \times n_x]$.

**Notes**

The Spread operator applies the following linear transform in forward mode to the model vector after reshaping it into a 2-dimensional array of size $[n_x \times n_t]$: $m(x_0, t_0) \rightarrow d(x, t = f(x_0, x, t_0))$

where $f(x_0, x, t)$ is a mapping function that returns a value t given values $x_0$, $x$, and $t_0$. 

3.6. PyLops API 155
In adjoint mode, the model is reconstructed by means of the following stacking operation:

\[ m(x_0, t_0) = \int d(x, t = f(x_0, x, t_0)) dx \]

Note that `table` (or `fh`) must return integer numbers representing indices in the axis \( t \). However, it is also possible to perform linear interpolation as part of the spreading/stacking process by providing the decimal part of the mapping function \( t - \lfloor t \rfloor \) either in `dtable` input parameter or as second value in the return of `fh` function.

**Attributes**

- `shape` [tuple] Operator shape
- `explicit` [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

- `__init__(dims, dimsd[, table, dtable, fh, ...])` Initialize this `LinearOperator`.
- `adjoint()` Hermitian adjoint.
- `apply_columns(cols)` Apply subset of columns of operator
- `cond([uselobpcg])` Condition number of linear operator.
- `conj()` Complex conjugate operator
- `div(y[, niter])` Solve the linear problem \( y = Ax \).
- `dot(x)` Matrix-matrix or matrix-vector multiplication.
- `eigs([neigs, symmetric, niter, uselobpcg])` Most significant eigenvalues of linear operator.
- `matmat(X)` Matrix-matrix multiplication.
- `matvec(x)` Matrix-vector multiplication.
- `rmatmat(X)` Adjoint matrix-matrix multiplication.
- `rmatvec(x)` Adjoint matrix-vector multiplication.
- `todense()` Return dense matrix.
- `tosparse()` Return sparse matrix.
- `transpose()` Transpose this linear operator.

**Examples using pylops.Spread**

- sphx_glr_gallery_plot_sliding.py
- sphx_glr_gallery_plot_radon.py
- 11. Radon filtering
- 12. Seismic regularization
- 16. CT Scan Imaging

**pylops.VStack**

**class** `pylops.VStack(ops, dtype=None)`

Vertical stacking.

Stack a set of \( N \) linear operators vertically.

**Parameters**
ops ([list] Linear operators to be stacked. Alternatively, numpy.ndarray or scipy.sparse matrices can be passed in place of one or more operators.

dtype [str, optional] Type of elements in input array.

Raises

ValueError If ops have different number of rows

Notes

A vertical stack of N linear operators is created such as its application in forward mode leads to

\[
\begin{bmatrix}
L_1 \\
L_2 \\
\vdots \\
L_N \\
\end{bmatrix} x =
\begin{bmatrix}
L_1 x \\
L_2 x \\
\vdots \\
L_N x \\
\end{bmatrix} =
\begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_N \\
\end{bmatrix}
\]

while its application in adjoint mode leads to

\[
\begin{bmatrix}
L_1^H \\
L_2^H \\
\vdots \\
L_N^H \\
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_N \\
\end{bmatrix} = L_1^H y_1 + L_2^H y_2 + \ldots + L_N^H y_N
\]

Attributes

shape [tuple] Operator shape
explicit [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

__init__ (ops[, dtype]) Initialize this LinearOperator.
adjoint() Hermitian adjoint.
apply_columns(cols) Apply subset of columns of operator
cond([uselobpcg]) Condition number of linear operator.
conj() Complex conjugate operator
div(y[, niter]) Solve the linear problem \( y = Ax \).
dot(x) Matrix-matrix or matrix-vector multiplication.
eigs([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
matmat(X) Matrix-matrix multiplication.
matvec(x) Matrix-vector multiplication.
rmatmat(X) Adjoint matrix-matrix multiplication.
rmatvec(x) Adjoint matrix-vector multiplication.
todense() Return dense matrix.
tosparse() Return sparse matrix.
transpose() Transpose this linear operator.

Examples using pylons.VStack

- sphx_glr_gallery_plot_derivative.py
- sphx_glr_gallery_plot_stacking.py
pylops.HStack

**class pylops.HStack(ops, dtype=None)**

Horizontal stacking.

- Stack a set of N linear operators horizontally.

**Parameters**

- **ops** ([list]) Linear operators to be stacked. Alternatively, `numpy.ndarray` or `scipy.sparse` matrices can be passed in place of one or more operators.
- **dtype** ([str, optional]) Type of elements in input array.

**Raises**

- **ValueError** If `ops` have different number of columns

**Notes**

An horizontal stack of N linear operators is created such as its application in forward mode leads to

$$
\begin{bmatrix}
L_1 & L_2 & \ldots & L_N
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_N
\end{bmatrix}
= L_1 x_1 + L_2 x_2 + \ldots + L_N x_N
$$

while its application in adjoint mode leads to

$$
\begin{bmatrix}
L_1^H \\
L_2^H \\
\vdots \\
L_N^H
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_N
\end{bmatrix}
= \begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_N
\end{bmatrix}
$$

**Attributes**

- **shape** ([tuple]) Operator shape
- **explicit** ([bool]) Operator contains a matrix that can be solved explicitly (`True`) or not (`False`)

**Methods**

- **__init__(ops[, dtype])** Initialize this LinearOperator.
- **adjoint()** Hermitian adjoint.
- **apply_columns(cols)** Apply subset of columns of operator
- **cond([uselobpcg])** Condition number of linear operator.
- **conj()** Complex conjugate operator
- **div(y[, niter])** Solve the linear problem $y = Ax$.
- **dot(x)** Matrix-matrix or matrix-vector multiplication.
- **eigs([neigs, symmetric, niter, uselobpcg])** Most significant eigenvalues of linear operator.
- **matmat(X)** Matrix-matrix multiplication.
- **matvec(x)** Matrix-vector multiplication.

Continued on next page
Table 20 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rmatmat(X)</code></td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>rmatvec(x)</code></td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>todense()</code></td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td><code>tosparse()</code></td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td><code>transpose()</code></td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

Examples using `pylops.HStack`

- sphx_glr_gallery_plot_stacking.py

`pylops.Block`

`pylops.Block(ops, dtype=None)`  
Block operator.

Create a block operator from N lists of M linear operators each.

**Parameters**

- `ops [list]` List of lists of operators to be combined in block fashion. Alternatively, `numpy.asarray` or `scipy.sparse` matrices can be passed in place of one or more operators.
- `dtype [str, optional]` Type of elements in input array.

**Notes**

In mathematics, a block or a partitioned matrix is a matrix that is interpreted as being broken into sections called blocks or submatrices. Similarly a block operator is composed of N sets of M linear operators each such that its application in forward mode leads to

\[
\begin{bmatrix}
L_{1,1} & L_{1,2} & \ldots & L_{1,M} \\
L_{2,1} & L_{2,2} & \ldots & L_{2,M} \\
\vdots & \vdots & \ddots & \vdots \\
L_{N,1} & L_{N,2} & \ldots & L_{N,M}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_M
\end{bmatrix} =
\begin{bmatrix}
L_{1,1}x_1 + L_{1,2}x_2 + L_{1,M}x_M \\
L_{2,1}x_1 + L_{2,2}x_2 + L_{2,M}x_M \\
\vdots \\
L_{N,1}x_1 + L_{N,2}x_2 + L_{N,M}x_M
\end{bmatrix}
\]

while its application in adjoint mode leads to

\[
\begin{bmatrix}
L_{1,1}^H & L_{2,1}^H & \ldots & L_{N,1}^H \\
L_{1,2}^H & L_{2,2}^H & \ldots & L_{N,2}^H \\
\vdots & \vdots & \ddots & \vdots \\
L_{1,M}^H & L_{2,M}^H & \ldots & L_{N,M}^H
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_N
\end{bmatrix} =
\begin{bmatrix}
L_{1,1}^Hy_1 + L_{2,1}^Hy_2 + L_{N,1}^Hy_N \\
L_{1,2}^Hy_1 + L_{2,2}^Hy_2 + L_{N,2}^Hy_N \\
\vdots \\
L_{1,M}^Hy_1 + L_{2,M}^Hy_2 + L_{N,M}^Hy_N
\end{bmatrix}
\]

**Attributes**

- `shape [tuple]` Operator shape
- `explicit [bool]` Operator contains a matrix that can be solved explicitly (True) or not (False)

Examples using `pylops.Block`

- sphx_glr_gallery_plot_stacking.py
class pylops.BlockDiag(ops, dtype=None)

Block-diagonal operator.

Create a block-diagonal operator from N linear operators.

Parameters

- **ops** [list] Linear operators to be stacked. Alternatively, numpy.ndarray or scipy.sparse matrices can be passed in place of one or more operators.
- **dtype** [str, optional] Type of elements in input array.

Notes

A block-diagonal operator composed of N linear operators is created such as its application in forward mode leads to

\[
\begin{bmatrix}
L_1 & 0 & \ldots & 0 \\
0 & L_2 & \ldots & 0 \\
\ldots & \ldots & \ldots & \ldots \\
0 & 0 & \ldots & L_N
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\ldots \\
x_N
\end{bmatrix}
=
\begin{bmatrix}
L_1 x_1 \\
L_2 x_2 \\
\ldots \\
L_N x_N
\end{bmatrix}
\]

while its application in adjoint mode leads to

\[
\begin{bmatrix}
L_1^H & 0 & \ldots & 0 \\
0 & L_2^H & \ldots & 0 \\
\ldots & \ldots & \ldots & \ldots \\
0 & 0 & \ldots & L_N^H
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
\ldots \\
y_N
\end{bmatrix}
=
\begin{bmatrix}
L_1^H y_1 \\
L_2^H y_2 \\
\ldots \\
L_N^H y_N
\end{bmatrix}
\]

Attributes

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

- **__init__**(ops[, dtype]) Initialize this LinearOperator.
- **adjoint()** Hermitian adjoint.
- **apply_columns(cols)** Apply subset of columns of operator
- **cond([uselobpcg])** Condition number of linear operator.
- **conj()** Complex conjugate operator
- **div(y[, niter])** Solve the linear problem \( y = Ax \).
- **dot(x)** Matrix-matrix or matrix-vector multiplication.
- **eigs([neigs, symmetric, niter, uselobpcg])** Most significant eigenvalues of linear operator.
- **matmat(X)** Matrix-matrix multiplication.
- **matvec(x)** Matrix-vector multiplication.
- **rmatmat(X)** Adjoint matrix-matrix multiplication.
- **rmatvec(x)** Adjoint matrix-vector multiplication.
- **todense()** Return dense matrix.
- **tosparse()** Return sparse matrix.
- **transpose()** Transpose this linear operator.
Examples using `pylops.BlockDiag`

- sphx_glr_gallery_plot_stacking.py

`pylops.Kronecker`

```python
class pylops.Kronecker(Op1, Op2, dtype='float64')
```

Kronecker operator.

Perform Kronecker product of two operators. Note that the combined operator is never created explicitly, rather the product of this operator with the model vector is performed in forward mode, or the product of the adjoint of this operator and the data vector in adjoint mode.

**Parameters**

- `Op1` [pylops.LinearOperator] First operator
- `Op2` [pylops.LinearOperator] Second operator
- `dtype` [str, optional] Type of elements in input array.

**Notes**

The Kronecker product (denoted with $\otimes$) is an operation on two operators $\mathbf{O}_1$ and $\mathbf{O}_2$ of sizes $[n_1 \times m_1]$ and $[n_2 \times m_2]$ respectively, resulting in a block matrix of size $[n_1 n_2 \times m_1 m_2]$.

\[
\mathbf{O}_1 \otimes \mathbf{O}_2 = \begin{bmatrix}
\mathbf{O}_1^{1,1} \mathbf{O}_2 & \cdots & \mathbf{O}_1^{1,m_1} \mathbf{O}_2 \\
\vdots & \ddots & \vdots \\
\mathbf{O}_1^{n_1,1} \mathbf{O}_2 & \cdots & \mathbf{O}_1^{n_1,m_1} \mathbf{O}_2
\end{bmatrix}
\]

The application of the resulting matrix to a vector $\mathbf{x}$ of size $[m_1 m_2 \times 1]$ is equivalent to the application of the second operator $\mathbf{O}_2$ to the rows of a matrix of size $[m_2 \times m_1]$ obtained by reshaping the input vector $\mathbf{x}$, followed by the application of the first operator to the transposed matrix produced by the first operator. In adjoint mode the same procedure is followed but the adjoint of each operator is used.

**Attributes**

- `shape` [tuple] Operator shape
- `explicit` [bool] Operator contains a matrix that can be solved explicitly (`True`) or not (`False`)

**Methods**

```python
__init__(Op1, Op2[, dtype]) Initialize this LinearOperator.
adjoint() Hermitian adjoint.
apply_columns(cols) Apply subset of columns of operator
cond([uselobpcg]) Condition number of linear operator.
conj() Complex conjugate operator
div(y[, niter]) Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
dot(x) Matrix-matrix or matrix-vector multiplication.
eigs([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
matmat(X) Matrix-matrix multiplication.
matvec(x) Matrix-vector multiplication.
```

Continued on next page
Table 22 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmatmat(X)</td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td>rmatvec(x)</td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td>todense()</td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

Examples using `pylops.Kronecker`

- sphx_glr_gallery_plot_stacking.py

Smoothing and derivatives

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pylops.Smoothing1D</code></td>
<td>1D Smoothing.</td>
</tr>
<tr>
<td><code>pylops.Smoothing2D</code></td>
<td>2D Smoothing.</td>
</tr>
<tr>
<td><code>pylops.FirstDerivative</code></td>
<td>First derivative.</td>
</tr>
<tr>
<td><code>pylops.SecondDerivative</code></td>
<td>Second derivative.</td>
</tr>
<tr>
<td><code>pylops.Laplacian</code></td>
<td>Laplacian.</td>
</tr>
<tr>
<td><code>pylops.Gradient</code></td>
<td>Gradient.</td>
</tr>
<tr>
<td><code>pylops.FirstDirectionalDerivative</code></td>
<td>First Directional derivative.</td>
</tr>
<tr>
<td><code>pylops.SecondDirectionalDerivative</code></td>
<td>Second Directional derivative.</td>
</tr>
</tbody>
</table>

**pylops.Smoothing1D**

Apply smoothing to model (and data) along a specific direction of a multi-dimensional array depending on the choice of `dir`.

**Parameters**

- `nsmooth` [int] Length of smoothing operator (must be odd)
- `dims` [tuple or int] Number of samples for each dimension
- `dir` [int, optional] Direction along which smoothing is applied
- `dtype` [str, optional] Type of elements in input array.

**Notes**

The Smoothing1D operator is a special type of convolutional operator that convolves the input model (or data) with a constant filter of size $n_{\text{smooth}}$:

$$f = [1/n_{\text{smooth}}, 1/n_{\text{smooth}}, \ldots, 1/n_{\text{smooth}}]$$

When applied to the first direction:

$$y[i, j, k] = 1/n_{\text{smooth}} \sum_{l=-(n_{\text{smooth}}-1)/2}^{(n_{\text{smooth}}-1)/2} x[l, j, k]$$
Similarly when applied to the second direction:

\[
y[i, j, k] = \frac{1}{n_{\text{smooth}}} \sum_{l=-(n_{\text{smooth}}-1)/2}^{(n_{\text{smooth}}-1)/2} x[i, l, k]
\]

and the third direction:

\[
y[i, j, k] = \frac{1}{n_{\text{smooth}}} \sum_{l=-(n_{\text{smooth}}-1)/2}^{(n_{\text{smooth}}-1)/2} x[i, j, l]
\]

Note that since the filter is symmetrical, the \textit{Smoothing1D} operator is self-adjoint.

**Attributes**

- \texttt{shape} [tuple] Operator shape
- \texttt{explicit} [bool] Operator contains a matrix that can be solved explicitly (\texttt{True}) or not (\texttt{False})

### Examples using \texttt{pylops.Smoothing1D}

- sphx\_glr\_gallery\_plot\_smoothing1d.py
- sphx\_glr\_gallery\_plot\_causalintegration.py
- sphx\_glr\_gallery\_plot\_wavest.py
- \textit{03. Solvers}

**\texttt{pylops.Smoothing2D}**

\texttt{pylops.Smoothing2D} \textit{(nsmooth, dims, nodir=None, dtype='float64')} 2D Smoothing.

Apply smoothing to model (and data) along two directions of a multi-dimensional array depending on the choice of \texttt{nodir}.

**Parameters**

- \texttt{nsmooth} [tuple or list] Length of smoothing operator in 1st and 2nd dimensions (must be odd)
- \texttt{dims} [tuple] Number of samples for each dimension
- \texttt{nodir} [int, optional] Direction along which smoothing is NOT applied (set to None for 2d arrays)
- \texttt{dtype} [str, optional] Type of elements in input array.

**See also:**

\texttt{lops.signalprocessing.Convolve2D} 2D convolution

**Notes**

The 2D Smoothing operator is a special type of convolutional operator that convolves the input model (or data) with a constant 2d filter of size \(n_{\text{smooth},1} \times n_{\text{smooth},2}\).
Its application to a two dimensional input signal is:

\[
y[i, j] = \frac{1}{(n_{\text{smooth}, 1} * n_{\text{smooth}, 2})} \sum_{l=-(n_{\text{smooth}, 1}-1)/2}^{(n_{\text{smooth}, 1}-1)/2} \sum_{m=-(n_{\text{smooth}, 2}-1)/2}^{(n_{\text{smooth}, 2}-1)/2} x[l, m]
\]

Note that since the filter is symmetrical, the `Smoothing2D` operator is self-adjoint.

**Attributes**

- `shape` [tuple] Operator shape
- `explicit` [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Examples using pylops.Smoothing2D**

- sphx_glr_gallery_plot_smoothing2d.py
- sphx_glr_gallery_plot_causalintegration.py

### pylops.FirstDerivative

**class pylops.FirstDerivative** (\(N, \text{dims=None, dir=0, sampling=1.0, edge=False, dtype='float64', kind='centered'}\))

First derivative.

Apply first derivative.

**Parameters**

- `N` [int] Number of samples in model.
- `dims` [tuple, optional] Number of samples for each dimension (None if only one dimension is available)
- `dir` [int, optional] Direction along which smoothing is applied.
- `sampling` [float, optional] Sampling step \(dx\).
- `edge` [bool, optional] Use reduced order derivative at edges (True) or ignore them (False)
- `dtype` [str, optional] Type of elements in input array.
- `kind` [str, optional] Derivative kind (forward, centered, or backward).

**Notes**

The FirstDerivative operator applies a first derivative to any chosen direction of a multi-dimensional array using either a second-order centered stencil or first-order forward/backward stencils.

For simplicity, given a one dimensional array, the second-order centered first derivative is:

\[
y[i] = (0.5x[i + 1] - 0.5x[i - 1]) / dx
\]

while the first-order forward stencil is:

\[
y[i] = (x[i + 1] - x[i]) / dx
\]

and the first-order backward stencil is:

\[
y[i] = (x[i] - x[i - 1]) / dx
\]
Attributes

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Initialize this LinearOperator.</td>
</tr>
<tr>
<td>adjoint()</td>
<td>Hermitian adjoint.</td>
</tr>
<tr>
<td>apply_columns(cols)</td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td>cond([uselobpcg])</td>
<td>Condition number of linear operator.</td>
</tr>
<tr>
<td>conj()</td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td>div(y, niter)</td>
<td>Solve the linear problem $y = Ax$.</td>
</tr>
<tr>
<td>dot(x)</td>
<td>Matrix-matrix or matrix-vector multiplication.</td>
</tr>
<tr>
<td>eigs([neigs, symmetric, niter, uselobpcg])</td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td>matmat(X)</td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td>matvec(x)</td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td>rmatmat(X)</td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td>rmatvec(x)</td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td>todense()</td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

Examples using **pylops.FirstDerivative**

- sphx_glr_gallery_plot_causalintegration.py
- sphx_glr_gallery_plot_derivative.py
- sphx_glr_gallery_plot_stacking.py
- sphx_glr_gallery_plot_tvreg.py
- 05. Image deblurring
- 16. CT Scan Imaging

**pylops.SecondDerivative**

```python
class pylops.SecondDerivative(N, dims=None, dir=0, sampling=1, edge=False, dtype='float64')
```

Second derivative.

Apply second-order second derivative.

Parameters

- **N** [int] Number of samples in model.
- **dims** [tuple, optional] Number of samples for each dimension (None if only one dimension is available)
- **dir** [int, optional] Direction along which smoothing is applied.
- **sampling** [float, optional] Sampling step $dx$.
- **edge** [bool, optional] Use reduced order derivative at edges (True) or ignore them (False)
**PyLops**

**dtype** [str, optional] Type of elements in input array.

**Notes**

The SecondDerivative operator applies a second derivative to any chosen direction of a multi-dimensional array. For simplicity, given a one dimensional array, the second-order centered first derivative is:

\[ y[i] = (x[i + 1] - 2x[i] + x[i - 1]) / dx^2 \]

**Attributes**

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__</code></td>
<td>Initialize this LinearOperator.</td>
</tr>
<tr>
<td><code>adjoint()</code></td>
<td>Hermitian adjoint.</td>
</tr>
<tr>
<td><code>apply_columns(cols)</code></td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td><code>cond([uselobpcg])</code></td>
<td>Condition number of linear operator.</td>
</tr>
<tr>
<td><code>conj()</code></td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td><code>div(y[, niter])</code></td>
<td>Solve the linear problem ( y = Ax ).</td>
</tr>
<tr>
<td><code>dot(x)</code></td>
<td>Matrix-matrix or matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>eigs([neigs, symmetric, niter, uselobpcg])</code></td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td><code>matmat(X)</code></td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>matvec(x)</code></td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>rmatmat(X)</code></td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>rmatvec(x)</code></td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>todense()</code></td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td><code>tospase()</code></td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td><code>transpose()</code></td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

**Examples using pylops.SecondDerivative**

- sphx_glr_gallery_plot_causalintegration.py
- sphx_glr_gallery_plot_derivative.py
- sphx_glr_gallery_plot_stacking.py
- sphx_glr_gallery_plot_tvreg.py
- sphx_glr_gallery_plot_wavest.py

**pylops.Laplacian**

**pylops.Laplacian**(dims, dirs=(0, 1), weights=(1, 1), sampling=(1, 1), edge=False, dtype='float64')

Laplacian.
Apply second-order centered Laplacian operator to a multi-dimensional array (at least 2 dimensions are required)

**Parameters**
- `dims [tuple]` Number of samples for each dimension.
- `dirs [tuple, optional]` Directions along which laplacian is applied.
- `weights [tuple, optional]` Weight to apply to each direction (real laplacian operator if `weights=[1,1]`)
- `sampling [tuple, optional]` Sampling steps for each direction
- `edge [bool, optional]` Use reduced order derivative at edges (True) or ignore them (False)
- `dtype [str, optional]` Type of elements in input array.

**Returns**
- `l2op [pylops.LinearOperator]` Laplacian linear operator

**Notes**
The Laplacian operator applies a second derivative along two directions of a multi-dimensional array.
For simplicity, given a two dimensional array, the Laplacian is:

\[ y[i, j] = \frac{(x[i+1, j] + x[i-1, j] + x[i, j-1] + x[i, j+1] - 4x[i, j])}{(dx * dy)} \]

**Examples using pylops.Laplacian**
- sphx_glr_gallery_plot_bilinear.py
- sphx_glr_gallery_plot_causalintegration.py
- sphx_glr_gallery_plot_derivative.py
- 06. 2D Interpolation
- 16. CT Scan Imaging

**pylops.Gradient**

`pylops.Gradient (dims, sampling=1, edge=False, dtype='float64', kind='centered')`

Apply gradient operator to a multi-dimensional array (at least 2 dimensions are required).

**Parameters**
- `dims [tuple]` Number of samples for each dimension.
- `sampling [tuple, optional]` Sampling steps for each direction.
- `edge [bool, optional]` Use reduced order derivative at edges (True) or ignore them (False).
- `dtype [str, optional]` Type of elements in input array.
- `kind [str, optional]` Derivative kind (forward, centered, or backward).

**Returns**
**Notes**

The Gradient operator applies a first-order derivative to each dimension of a multi-dimensional array in forward mode.

For simplicity, given a three dimensional array, the Gradient in forward mode using a centered stencil can be expressed as:

\[ g_{i,j,k} = \frac{(f_{i+1,j,k} - f_{i-1,j,k})}{d_1i_1} + \frac{(f_{i,j+1,k} - f_{i,j-1,k})}{d_2i_2} + \frac{(f_{i,j,k+1} - f_{i,j,k-1})}{d_3i_3} \]

which is discretized as follows:

\[
\begin{bmatrix}
df_1 \\
df_2 \\
df_3
\end{bmatrix}
\]

In adjoint mode, the adjoints of the first derivatives along different axes are instead summed together.

**Examples using Pylops.Gradient**

- sphx_glr_gallery_plot_derivative.py

**pylops.FirstDirectionalDerivative**

*pylops.FirstDirectionalDerivative*(*dims*, *v*, *sampling=1*, *edge=False*, *dtype='float64'*, *kind='centered'*)

First Directional derivative.

Apply directional derivative operator to a multi-dimensional array (at least 2 dimensions are required) along either a single common direction or different directions for each point of the array.

**Parameters**

- **dims** (*tuple*) Number of samples for each dimension.
- **v** (*np.ndarray*, optional) Single direction (array of size \(n_{dims}\)) or group of directions (array of size \([n_{dims} \times n_{d0} \times ... \times n_{d_{dims}}]\))
- **sampling** (*tuple*, optional) Sampling steps for each direction.
- **edge** (*bool*, optional) Use reduced order derivative at edges (True) or ignore them (False).
- **dtype** (*str*, optional) Type of elements in input array.
- **kind** (*str*, optional) Derivative kind (forward, centered, or backward).

**Returns**

- **ddop** (*pylops.LinearOperator*) First directional derivative linear operator

**Notes**

The FirstDirectionalDerivative applies a first-order derivative to a multi-dimensional array along the direction defined by the unitary vector v:

\[ df_v = \nabla f v \]
or along the directions defined by the unitary vectors $v(x, y)$:

$$df(x, y) = \nabla f(x, y)v(x, y)$$

where we have here considered the 2-dimensional case.

This operator can be easily implemented as the concatenation of the `pylops.Gradient` operator and the `pylops.Diagonal` operator with $v$ along the main diagonal.

**Examples using pylops.FirstDirectionalDerivative**

- sphx_glr_gallery_plot_derivative.py

**pylops.SecondDirectionalDerivative**

`pylops.SecondDirectionalDerivative(dims, v, sampling=1, edge=False, dtype='float64')`

Second directional derivative.

Apply second directional derivative operator to a multi-dimensional array (at least 2 dimensions are required) along either a single common direction or different directions for each point of the array.

Parameters

- **dims** [tuple] Number of samples for each dimension.
- **v** [np.ndarray, optional] Single direction (array of size $n_{\text{dims}}$) or group of directions (array of size $[n_{\text{dims}} \times n_{d0} \times \ldots \times n_{d_{\text{dims}}}])$
- **sampling** [tuple, optional] Sampling steps for each direction.
- **edge** [bool, optional] Use reduced order derivative at edges (True) or ignore them (False).
- **dtype** [str, optional] Type of elements in input array.

Returns

- **ddop** [pylops.LinearOperator] Second directional derivative linear operator

**Notes**

The SecondDirectionalDerivative applies a second-order derivative to a multi-dimensional array along the direction defined by the unitary vector $v$:

$$d^2 f_v = -D^T_v [D_v f]$$

where $D_v$ is the first-order directional derivative implemented by `pylops.SecondDirectionalDerivative`.

This operator is sometimes also referred to as directional Laplacian in the literature.

**Examples using pylops.SecondDirectionalDerivative**

- sphx_glr_gallery_plot_derivative.py

**Signal processing**
PyLops

Convolve1D(N, h[, offset, dims, dir, dtype, ...]) 1D convolution operator.
Convolve2D(N, h, dims[, offset, nodir, ...]) 2D convolution operator.
ConvolveND(N, h, dims[, offset, dirs, ...]) ND convolution operator.
Interp(M, iava[, dims, dir, kind, dtype]) Interpolation operator.
Bilinear[iava, dims[, dtype]] Bilinear interpolation operator.
FFT(dims[, dir, nfft, sampling, real, ...]) One dimensional Fast-Fourier Transform.
FFT2D(dims[, dirs, nffts, sampling, dtype]) Two dimensional Fast-Fourier Transform.
FFTND(dims[, dirs, nffts, sampling, dtype]) N-dimensional Fast-Fourier Transform.
DWT(dims[, dir, wavelet, level, dtype]) One dimensional Wavelet operator.
DWT2D(dims[, dirs, wavelet, level, dtype]) Two dimensional Wavelet operator.
Seislet(slopes[, sampling, level, kind, ...]) Two dimensional Seislet operator.
Radon2D(taxis, haxis, pxaxis[, kind, ...]) Two dimensional Radon transform.
Radon3D(taxis, hyaxis, hxaxis, pyaxis, pxaxis) Three dimensional Radon transform.
Sliding2D(Op, dims, dimsd, nwin, nover[, ...]) 2D Sliding transform operator.
Sliding3D(Op, dims, dimsd, nwin, nover, nop) 3D Sliding transform operator.
Fredholm1(G[, nz, saveGt, usematmul, dtype]) Fredholm integral of first kind.

pylops.signalprocessing.Convolve1D

class pylops.signalprocessing.Convolve1D(N, h, offset=0, dims=None, dir=0, dtype='float64', method=None)

1D convolution operator.

Apply one-dimensional convolution with a compact filter to model (and data) along a specific direction of a multi-dimensional array depending on the choice of dir.

Parameters

N [int] Number of samples in model.

h [numpy.ndarray] 1d compact filter to be convolved to input signal

offset [int] Index of the center of the compact filter

dims [tuple] Number of samples for each dimension (None if only one dimension is available)

dir [int, optional] Direction along which convolution is applied

method [str, optional] Method used to calculate the convolution (direct, fft, or overlapadd). Note that only direct and fft are allowed when dims=None, whilst fft and overlapadd are allowed when dims is provided.

dtype [str, optional] Type of elements in input array.

Raises

ValueError If offset is bigger than len(h) - 1

NotImplementedError If method provided is not allowed

Notes

The Convolve1D operator applies convolution between the input signal \( x(t) \) and a compact filter kernel \( h(t) \) in forward model:

\[
y(t) = \int_{-\infty}^{\infty} h(t - \tau)x(\tau)d\tau
\]
This operation can be discretized as follows

\[ y[n] = \sum_{m=-\infty}^{\infty} h[n - m]x[m] \]

as well as performed in the frequency domain.

\[ Y(f) = F(h(t)) \ast F(x(t)) \]

Convolve1D operator uses `scipy.signal.convolve` that automatically chooses the best domain for the operation to be carried out for one dimensional inputs. The fft implementation `scipy.signal.fftconvolve` is however enforced for signals in 2 or more dimensions as this routine efficiently operates on multi-dimensional arrays.

As the adjoint of convolution is correlation, Convolve1D operator applies correlation in the adjoint mode.

In time domain:

\[ x(t) = \int_{-\infty}^{\infty} h(t + \tau)x(\tau)d\tau \]

or in frequency domain:

\[ y(t) = F^{-1}(H(f)^* \ast X(f)) \]

**Attributes**

- `shape` [tuple] Operator shape
- `explicit` [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

- `__init__`(N, h[, offset, dims, dir, dtype, ...]) Initialize this LinearOperator.
- `adjoint`() Hermitian adjoint.
- `apply_columns`(cols) Apply subset of columns of operator
- `cond([uselobpcg])` Condition number of linear operator.
- `conj`() Complex conjugate operator
- `div(y[, niter])` Solve the linear problem \( y = Ax \).
- `dot(x)` Matrix-matrix or matrix-vector multiplication.
- `eigs([neigs, symmetric, niter, uselobpcg])` Most significant eigenvalues of linear operator.
- `matmat(X)` Matrix-matrix multiplication.
- `matvec(x)` Matrix-vector multiplication.
- `rmatmat(X)` Adjoint matrix-matrix multiplication.
- `rmatvec(x)` Adjoint matrix-vector multiplication.
- `todense`() Return dense matrix.
- `tosparses()` Return sparse matrix.
- `transpose`() Transpose this linear operator.

**Examples using pylops.signalprocessing.Convolve1D**

- sphi_glr_gallery_plot_smoothing1d.py
- sphi_glr_gallery_plot_convolve.py
pylops.signalprocessing.Convolve2D

`pylops.signalprocessing.Convolve2D(N, h, dims, offset=(0, 0), nodir=None, dtype='float64', method='fft')`

2D convolution operator.

Apply two-dimensional convolution with a compact filter to model (and data) along a pair of specific directions of a two or three-dimensional array depending on the choice of `nodir`.

**Parameters**

- `N` [int] Number of samples in model
- `h` [numpy.ndarray] 2d compact filter to be convolved to input signal
- `dims` [list] Number of samples for each dimension
- `offset` [tuple, optional] Indeces of the center of the compact filter
- `nodir` [int, optional] Direction along which convolution is NOT applied (set to None for 2d arrays)
- `dtype` [str, optional] Type of elements in input array.
- `method` [str, optional] Method used to calculate the convolution (direct or fft).

**Returns**

- `cop` [pylops.LinearOperator] Convolve2D linear operator

**Notes**

The Convolve2D operator applies two-dimensional convolution between the input signal $d(t, x)$ and a compact filter kernel $h(t, x)$ in forward model:

$$y(t, x) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(t - \tau, x - \chi) d(\tau, \chi) d\tau d\chi$$

This operation can be discretized as follows

$$y[i, n] = \sum_{j=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} h[i - j, n - m] d[j, m]$$

as well as performed in the frequency domain.

$$Y(f, k_x) = F(h(t, x)) \ast F(d(t, x))$$

Convolve2D operator uses `scipy.signal.convolve2d` that automatically chooses the best domain for the operation to be carried out.

As the adjoint of convolution is correlation, Convolve2D operator applies correlation in the adjoint mode.
In time domain:

\[ y(t, x) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(t + \tau, x + \chi) d\tau d\chi \]

or in frequency domain:

\[ y(t, x) = F^{-1}(H(f, k_x) \ast X(f, k_x)) \]

Examples using `pylops.signalprocessing.Convolve2D`

- sphx_glr_gallery_plot_convolve.py
- 05. Image deblurring

`pylops.signalprocessing.ConvolveND`

class `pylops.signalprocessing.ConvolveND(N, h, dims, offset=(0, 0, 0), dirs=None, method='fft', dtype='float64')`

ND convolution operator.

Apply n-dimensional convolution with a compact filter to model (and data) along a set of directions `dirs` of a n-dimensional array.

Parameters

- `N` [int] Number of samples in model
- `h` [numpy.ndarray] nd compact filter to be convolved to input signal
- `dims` [list] Number of samples for each dimension
- `offset` [tuple, optional] Indices of the center of the compact filter
- `dirs` [tuple, optional] Directions along which convolution is applied (set to None for filter of same dimension as input vector)
- `method` [str, optional] Method used to calculate the convolution (direct or fft).
- `dtype` [str, optional] Type of elements in input array.

Notes

The ConvolveND operator applies n-dimensional convolution between the input signal \( d(x_1, x_2, ..., x_N) \) and a compact filter kernel \( h(x_1, x_2, ..., x_N) \) in forward model. This is a straightforward extension to multiple dimensions of `pylops.signalprocessing.Convolve2D` operator.

Attributes

- `shape` [tuple] Operator shape
- `explicit` [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods
__init__\((N, h, \text{dims}[\], \text{offset, dirs, method, \ldots])\) Initialize this LinearOperator.

adjoint() Hermitian adjoint.

apply_columns(cols) Apply subset of columns of operator

cond([uselobpcg]) Condition number of linear operator.

conj() Complex conjugate operator

div(y[, niter]) Solve the linear problem \( y = Ax \).

dot(x) Matrix-matrix or matrix-vector multiplication.

eigs([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.

matmat(X) Matrix-matrix multiplication.

matvec(x) Matrix-vector multiplication.

rmatmat(X) Adjoint matrix-matrix multiplication.

rmatvec(x) Adjoint matrix-vector multiplication.

todense() Return dense matrix.

tosparse() Return sparse matrix.

transpose() Transpose this linear operator.

Examples using pylops.signalprocessing.ConvolveND

- sphx_glr_gallery_plot_smoothing2d.py
- sphx_glr_gallery_plot_causalintegration.py
- sphx_glr_gallery_plot_convolve.py
- 05. Image deblurring

pylops.signalprocessing.Interp

pylops.signalprocessing.Interp\((M, \text{iava}, \text{dims=}\text{None}, \text{dir=}0, \text{kind=}\text{\'linear\'}, \text{dtype=}\text{\'float64\'}\)

Interpolation operator.

Apply interpolation along direction \text{dir} from regularly sampled input vector into fractionary positions \text{iava} using one of the following algorithms:

- \textit{Nearest neighbour} interpolation is a thin wrapper around \texttt{pylops.Restriction} at \texttt{np.round(iava)} locations.

- \textit{Linear interpolation} extracts values from input vector at locations \texttt{np.floor(iava)} and \texttt{np.floor(iava)+1} and linearly combines them in forward mode, places weighted versions of the interpolated values at locations \texttt{np.floor(iava)} and \texttt{np.floor(iava)+1} in an otherwise zero vector in adjoint mode.

- \textit{Sinc interpolation} performs sinc interpolation at locations \text{iava}. Note that this is the most accurate method but it has higher computational cost as it involves multiplying the input data by a matrix of size \( N \times M \).

Note: The vector \text{iava} should contain unique values. If the same index is repeated twice an error will be raised. This also applies when values beyond the last element of the input array for \textit{linear interpolation} as those values are forced to be just before this element.

Parameters

\( M \) [\text{int}] Number of samples in model.
**iava** [list or numpy.ndarray] Floating indices of locations of available samples for interpolation.

**dims** [list, optional] Number of samples for each dimension (None if only one dimension is available)

**dir** [int, optional] Direction along which restriction is applied.

**kind** [str, optional] Kind of interpolation (nearest, linear, and sinc are currently supported)

**dtype** [str, optional] Type of elements in input array.

**Returns**

**op** [pylops.LinearOperator] Linear interpolation operator

**iava** [list or numpy.ndarray] Corrected indices of locations of available samples (samples at M-1 or beyond are forced to be at M-1-eps)

**Raises**

**ValueError** If the vector iava contains repeated values.

**NotImplementedError** If kind is not nearest, linear or sinc

**See also:**

**pylopsRestriction** Restriction operator

**Notes**

Linear interpolation of a subset of N values at locations iava from an input (or model) vector x of size M can be expressed as:

\[ y_i = (1 - w_i)x_{l_i} + w_i x_{l_i} \quad \forall i = 1, 2, \ldots, N \]

where \( l_i = \lfloor l_1 \rfloor, \lfloor l_2 \rfloor, \ldots, \lfloor l_N \rfloor \) and \( l_i' = \lfloor l_1 \rfloor + 1, \lfloor l_2 \rfloor + 1, \ldots, \lfloor l_N \rfloor + 1 \) are vectors containing the indeces of the original array at which samples are taken, and \( w = [l_1 - \lfloor l_1 \rfloor, l_2 - \lfloor l_2 \rfloor, \ldots, l_N - \lfloor l_N \rfloor] \) are the linear interpolation weights. This operator can be implemented by simply summing two pylops.Restriction operators which are weighted using pylops.basicoperators.Diagonal operators.

Sinc interpolation of a subset of N values at locations iava from an input (or model) vector x of size M can be expressed as:

\[ y_i = \sum_{j=0}^{M} x_j \text{sinc}(i - j) \quad \forall i = 1, 2, \ldots, N \]

This operator can be implemented using the pylops.MatrixMult operator with a matrix containing the values of the sinc function at all i, j possible combinations.

**Examples using pylops.signalprocessing.Interp**

- sphx_glr_gallery_plot_restriction.py
class pylops.signalprocessing.Bilinear(iava, dims, dtype='float64')

Bilinear interpolation operator.

Apply bilinear interpolation onto fractionary positions iava along the first two axes of a n-dimensional array.

**Note:** The vector iava should contain unique pairs. If the same pair is repeated twice an error will be raised.

**Parameters**

- **iava** *list or numpy.ndarray* Array of size $[2 \times n_{ava}]$ containing pairs of floating indices of locations of available samples for interpolation.
- **dims** *list* Number of samples for each dimension
- **dtype** *str, optional* Type of elements in input array.

**Raises**

ValueError If the vector iava contains repeated values.

**Notes**

Bilinear interpolation of a subset of $N$ values at locations iava from an input n-dimensional vector $x$ of size $[m_1 \times m_2 \times \ldots \times m_{ndim}]$ can be expressed as:

\[ y_i = (1 - w_{i0}^0) (1 - w_{i1}^0) x_{l_{i0}, l_{i1}, \ldots, l_{iN}}^0 + w_{i0}^0 (1 - w_{i1}^0) x_{l_{i0}, l_{i1}, \ldots, l_{iN}}^1 + (1 - w_{i0}^0) w_{i1}^0 x_{l_{i0}, l_{i1}, \ldots, l_{iN}}^1 + w_{i0}^0 w_{i1}^0 x_{l_{i0}, l_{i1}, \ldots, l_{iN}}^1 \]  

where $l_{i}^0 = [l_{i1}^0, l_{i2}^0, \ldots, l_{iN}^0], l_{i}^1 = [l_{i1}^1, l_{i2}^1, \ldots, l_{iN}^1], l_{i}^0 = [l_{i1}^0 + 1, l_{i2}^0 + 1, \ldots, l_{iN}^0 + 1], l_{i}^1 = [l_{i1}^1 + 1, l_{i2}^1 + 1, \ldots, l_{iN}^1 + 1], w_{i}^j = [l_{i1}^j - l_{i1}^0, l_{i2}^j - l_{i2}^0, \ldots, l_{iN}^j - l_{iN}^0] (\forall j = 0, 1)$ are the bilinear interpolation weights.

**Attributes**

- **shape** *tuple* Operator shape
- **explicit** *bool* Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

__init__(iava, dims[, dtype]) Initialize this LinearOperator.
adjoint() Hermitian adjoint.
apply_columns(cols) Apply subset of columns of operator
cond([uselobpcg]) Condition number of linear operator.
conj() Complex conjugate operator
div(y[, niter]) Solve the linear problem $y = Ax$.
dot(x) Matrix-matrix or matrix-vector multiplication.
eigs([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
matmat(X) Matrix-matrix multiplication.
matvec(x) Matrix-vector multiplication.
rmatmat(X) Adjoint matrix-matrix multiplication.
rmatvec(x) Adjoint matrix-vector multiplication.
Table 29 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>todense()</td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

**Examples using pylops.signalprocessing.Bilinear**

- sphx_glr_gallery_plot_bilinear.py

**pylops.signalprocessing.FFT**

*pylops.signalprocessing.FFT*(dims, dir=0, nfft=None, sampling=1.0, real=False, fftshift=False, engine='numpy', dtype='complex128', **kwargs_fftw)

One dimensional Fast-Fourier Transform.

Apply Fast-Fourier Transform (FFT) along a specific direction *dir* of a multi-dimensional array of size *dim*.

Note that the FFT operator is an overload to either the numpy *numpy.fft.fft* (or *numpy.fft.rfft* for real models) in forward mode and to the numpy *numpy.fft.ifft* (or *numpy.fft.irfft* for real models) in adjoint mode, or to the pyfftw.FFTW class.

In both cases, scaling is properly taken into account to guarantee that the operator is passing the dot-test.

**Note:** For a real valued input signal, it is possible to store the values of the Fourier transform at positive frequencies only as values at negative frequencies are simply their complex conjugates. However as the operation of removing the negative part of the frequency axis in forward mode and adding the complex conjugates in adjoint mode is nonlinear, the Linear Operator FFT with *real=True* is not expected to pass the dot-test. It is thus *only* advised to use this flag when a forward and adjoint FFT is used in the same chained operator (e.g., FFT.H*Op*FFT) such as in pylops.waveeqprocessing.mdd.MDC.

**Parameters**

- **dims** [tuple] Number of samples for each dimension
- **dir** [int, optional] Direction along which FFT is applied.
- **nfft** [int, optional] Number of samples in Fourier Transform (same as input if nfft=None)
- **sampling** [float, optional] Sampling step *dt*.
- **real** [bool, optional] Model to which fft is applied has real numbers (True) or not (False). Used to enforce that the output of adjoint of a real model is real.
- **fftshift** [bool, optional] Apply fftshift/ifftshift (True) or not (False)
- **engine** [str, optional] Engine used for fft computation (numpy or fftw)
- **dtype** [str, optional] Type of elements in input array. Note that the *dtype* of the operator is the corresponding complex type even when a real type is provided. Nevertheless, the provided dtype will be enforced on the vector returned by the *rmatvec* method.

**kwargs_fftw** Arbitrary keyword arguments for pyfftw.FFTW

**Raises**

- **ValueError** If *dims* is not provided and if *dir* is bigger than *len(dims)*
- **NotImplementedError** If *engine* is neither *numpy* nor *numba*
**Notes**

The FFT operator applies the forward Fourier transform to a signal $d(t)$ in forward mode:

$$
D(f) = F(d) = \int d(t)e^{-j2\pi ft}dt
$$

Similarly, the inverse Fourier transform is applied to the Fourier spectrum $D(f)$ in adjoint mode:

$$
d(t) = F^{-1}(D) = \int D(f)e^{j2\pi ft}df
$$

Both operators are effectively discretized and solved by a fast iterative algorithm known as Fast Fourier Transform.

Note that the FFT operator is a special operator in that the adjoint is also the inverse of the forward mode. Moreover, in case of real signal in time domain, the Fourier transform in Hermitian.

**Attributes**

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Examples using pylops.signalprocessing.FFT**

- sphx_glr_gallery_plot_fft.py
- 02. The Dot-Test
- 03. Solvers
- 04. Bayesian Inversion

**pylops.signalprocessing.FFT2D**

```python
class pylops.signalprocessing.FFT2D(dims, dirs=(0, 1), nffts=(None, None), sampling=(1.0, 1.0), dtype='complex128')
```

Two dimensional Fast-Fourier Transform.

Apply two dimensional Fast-Fourier Transform (FFT) to any pair of axes of a multi-dimensional array depending on the choice of `dirs`. Note that the FFT2D operator is a simple overload to the numpy `numpy.fft.fft2` in forward mode and to the numpy `numpy.fft.ifft2` in adjoint mode, however scaling is taken into account differently to guarantee that the operator is passing the dot-test.

**Parameters**

- **dims** [tuple] Number of samples for each dimension
- **dirs** [tuple, optional] Pair of directions along which FFT2D is applied
- **nffts** [tuple, optional] Number of samples in Fourier Transform for each direction (same as input if `nffts=(None, None)`)  
- **sampling** [tuple, optional] Sampling steps $dy$ and $dx$
- **dtype** [str, optional] Type of elements in input array. Note that the `dtype` of the operator is the corresponding complex type even when a real type is provided. Nevertheless, the provided `dtype` will be enforced on the vector returned by the `rmatvec` method.

**Raises**
ValueError: If dims has less than two elements, and if dirs, nffts, or sampling has more or less than two elements.

Notes

The FFT2D operator applies the two-dimensional forward Fourier transform to a signal $d(y, x)$ in forward mode:

$$D(k_y, k_x) = F(d) = \int \int d(y, x) e^{-j2\pi k_y y} e^{-j2\pi k_x x} dy dx$$

Similarly, the two-dimensional inverse Fourier transform is applied to the Fourier spectrum $D(k_y, k_x)$ in adjoint mode:

$$d(y, x) = F^{-1}(D) = \int \int D(k_y, k_x) e^{j2\pi k_y y} e^{j2\pi k_x x} dk_y dk_x$$

Both operators are effectively discretized and solved by a fast iterative algorithm known as Fast Fourier Transform.

Attributes

- shape [tuple] Operator shape
- explicit [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

- __init__(dims[, dirs, nffts, sampling, dtype]) Initialize this LinearOperator.
- adjoint() Hermitian adjoint.
- apply_columns(cols) Apply subset of columns of operator
- cond([uselobpcg]) Condition number of linear operator.
- conj() Complex conjugate operator
- div(y[, niter]) Solve the linear problem $y = Ax$.
- dot(x) Matrix-matrix or matrix-vector multiplication.
- eigs([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
- matmat(X) Matrix-matrix multiplication.
- matvec(x) Matrix-vector multiplication.
- rmatmat(X) Adjoint matrix-matrix multiplication.
- rmatvec(x) Adjoint matrix-vector multiplication.
- todense() Return dense matrix.
- tosparse() Return sparse matrix.
- transpose() Transpose this linear operator.

Examples using pylops.signalprocessing.FFT2D

- sphx_glr_gallery_plot_fft.py
- sphx_glr_gallery_plot_tvreg.py
- 12. Seismic regularization
- 14. Seismic wavefield decomposition
**pylops.signalprocessing.FFTND**

**class pylops.signalprocessing.FFTND** *(dims, dirs=(0, 1, 2), nffts=(None, None, None), sampling=(1.0, 1.0, 1.0), dtype='complex128')*

N-dimensional Fast-Fourier Transform.

Apply n-dimensional Fast-Fourier Transform (FFT) to any n axes of a multi-dimensional array depending on the choice of `dirs`. Note that the FFTND operator is a simple overload to the numpy `numpy.fft.fftn` in forward mode and to the numpy `numpy.fft.ifftn` in adjoint mode, however scaling is taken into account differently to guarantee that the operator is passing the dot-test.

**Parameters**

- **dims** [tuple] Number of samples for each dimension
- **dirs** [tuple, optional] Directions along which FFTND is applied
- **nffts** [tuple, optional] Number of samples in Fourier Transform for each direction (same as input if `nffts=(None, None, None, ..., None)`)
- **sampling** [tuple, optional] Sampling steps in each direction
- **dtype** [str, optional] Type of elements in input array

**Raises**

- **ValueError** If `dims`, `dirs`, `nffts`, or `sampling` have less than three elements and if the dimension of `dirs`, `nffts`, and `sampling` is not the same

**Notes**

The FFTND operator applies the n-dimensional forward Fourier transform to a multi-dimensional array. Without loss of generality we consider here a three-dimensional signal $d(z, y, x)$. The FFTND in forward mode is:

$$D(k_z, k_y, k_x) = F(d) = \int \int \int d(z, y, x) e^{-j2\pi k_z z} e^{-j2\pi k_y y} e^{-j2\pi k_x x} dz dy dx$$

Similarly, the three-dimensional inverse Fourier transform is applied to the Fourier spectrum $D(k_z, k_y, k_x)$ in adjoint mode:

$$d(z, y, x) = F^{-1}(D) = \int \int \int D(k_z, k_y, k_x) e^{j2\pi k_z z} e^{j2\pi k_y y} e^{j2\pi k_x x} dk_z dk_y dk_x$$

Both operators are effectively discretized and solved by a fast iterative algorithm known as Fast Fourier Transform.

**Attributes**

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong> (dims[, dirs, nffts, sampling, dtype])</td>
<td>Initialize this LinearOperator.</td>
</tr>
<tr>
<td>adjoint()</td>
<td>Hermitian adjoint.</td>
</tr>
<tr>
<td>apply_columns(cols)</td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td>cond([uselobpcg])</td>
<td>Condition number of linear operator.</td>
</tr>
</tbody>
</table>
Table 31 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>conj()</code></td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td><code>div(y[, niter])</code></td>
<td>Solve the linear problem ( y = Ax ).</td>
</tr>
<tr>
<td><code>dot(x)</code></td>
<td>Matrix-matrix or matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>eigs([neigs, symmetric, niter, uselobpcg])</code></td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td><code>matmat(X)</code></td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>matvec(x)</code></td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>rmatmat(X)</code></td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>rmatvec(x)</code></td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>todense()</code></td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td><code>tosparse()</code></td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td><code>transpose()</code></td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

Examples using `pylops.signalprocessing.FFTND`

- sphx_glr_gallery_plot_fft.py

`pylops.signalprocessing.DWT`

```python
class pylops.signalprocessing.DWT(dims, dir=0, wavelet='haar', level=1, dtype='float64')
```

One dimensional Wavelet operator.

Apply 1D-Wavelet Transform along a specific direction `dir` of a multi-dimensional array of size `dims`.

Note that the Wavelet operator is an overload of the `pywt` implementation of the wavelet transform. Refer to https://pywavelets.readthedocs.io for a detailed description of the input parameters.

**Parameters**

- `dims` [int or tuple] Number of samples for each dimension
- `dir` [int, optional] Direction along which DWT is applied.
- `wavelet` [str, optional] Name of wavelet type. Use `pywt.wavelist(kind='discrete')` for a list of available wavelets.
- `level` [int, optional] Number of scaling levels (must be >=0).
- `dtype` [str, optional] Type of elements in input array.

**Raises**

- `ModuleNotFoundError` If `pywt` is not installed
- `ValueError` If `wavelet` does not belong to `pywt.families`

**Notes**

The Wavelet operator applies the multilevel Discrete Wavelet Transform (DWT) in forward mode and the multilevel Inverse Discrete Wavelet Transform (IDWT) in adjoint mode.

Wavelet transforms can be used to compress signals and present a key advantage over Fourier transforms in that they capture both frequency and location information in time. Consider using this operator as sparsifying transform when using L1 solvers.

**Attributes**

- `shape` [tuple] Operator shape
explicit [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(dims[, dir, wavelet, level, dtype])</td>
<td>Initialize this LinearOperator.</td>
</tr>
<tr>
<td>adjoint()</td>
<td>Hermitian adjoint.</td>
</tr>
<tr>
<td>apply_columns(cols)</td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td>cond([uselobpcg])</td>
<td>Condition number of linear operator.</td>
</tr>
<tr>
<td>conj()</td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td>div(y[, niter])</td>
<td>Solve the linear problem ( y = Ax ).</td>
</tr>
<tr>
<td>dot(x)</td>
<td>Matrix-matrix or matrix-vector multiplication.</td>
</tr>
<tr>
<td>eigs([neigs, symmetric, niter, uselobpcg])</td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td>matmat(X)</td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td>matvec(x)</td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td>rmatmat(X)</td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td>rmatvec(x)</td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td>todense()</td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

**Examples using pylops.signalprocessing.DWT**

- sphx_glr_gallery_plot_wavelet.py

**pylops.signalprocessing.DWT2D**

class pylops.signalprocessing.DWT2D(dims, dirs=(0, 1), wavelet='haar', level=1, dtype='float64')

Two dimensional Wavelet operator.

Apply 2D-Wavelet Transform along two directions \( \text{dirs} \) of a multi-dimensional array of size \( \text{dims} \).

Note that the Wavelet operator is an overload of the pywt implementation of the wavelet transform. Refer to https://pywavelets.readthedocs.io for a detailed description of the input parameters.

**Parameters**

- **dims** [tuple] Number of samples for each dimension
- **dirs** [tuple, optional] Direction along which DWT2D is applied.
- **wavelet** [str, optional] Name of wavelet type. Use `pywt.wavelist(kind='discrete')` for a list of available wavelets.
- **level** [int, optional] Number of scaling levels (must be >=0).
- **dtype** [str, optional] Type of elements in input array.

**Raises**

- **ModuleNotFoundError** If `pywt` is not installed
- **ValueError** If `wavelet` does not belong to `pywt.families`
**Notes**

The Wavelet operator applies the 2-dimensional multilevel Discrete Wavelet Transform (DWT2) in forward mode and the 2-dimensional multilevel Inverse Discrete Wavelet Transform (IDWT2) in adjoint mode.

**Attributes**

- `shape [tuple]` Operator shape
- `explicit [bool]` Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(dims[, dirs, wavelet, level, dtype])</code></td>
<td>Initialize this LinearOperator.</td>
</tr>
<tr>
<td><code>adjoint()</code></td>
<td>Hermitian adjoint.</td>
</tr>
<tr>
<td><code>apply_columns(cols)</code></td>
<td>Apply subset of columns of operator</td>
</tr>
<tr>
<td><code>cond([uselobpcg])</code></td>
<td>Condition number of linear operator.</td>
</tr>
<tr>
<td><code>conj()</code></td>
<td>Complex conjugate operator</td>
</tr>
<tr>
<td><code>div(y[, niter])</code></td>
<td>Solve the linear problem $y = Ax$.</td>
</tr>
<tr>
<td><code>dot(x)</code></td>
<td>Matrix-matrix or matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>eigs([neigs, symmetric, uselobpcg])</code></td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td><code>matmat(X)</code></td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>matvec(x)</code></td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>rmatmat(X)</code></td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>rmatvec(x)</code></td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>todense()</code></td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td><code>tosparse()</code></td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td><code>transpose()</code></td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

**Examples using** `pylops.signalprocessing.DWT2D`

- `sphx_glr_gallery_plot_wavelet.py`
- `05. Image deblurring`

**pylops.signalprocessing.Seislet**

```python
class pylops.signalprocessing.Seislet(slopes, sampling=(1.0, 1.0), level=None, kind='haar', inv=False, dtype='float64')
```

Two dimensional Seislet operator.

Apply 2D-Seislet Transform to an input array given an estimate of its local `slopes`. In forward mode, the input array is reshaped into a two-dimensional array of size $n_x \times n_t$ and the transform is performed along the first (spatial) axis (see Notes for more details).

**Parameters**

- **slopes** [numpy.ndarray] Slope field of size $n_x \times n_t$
- **sampling** [tuple, optional] Sampling steps in x- and t-axis.
- **level** [int, optional] Number of scaling levels (must be $\geq 0$).
- **kind** [str, optional] Basis function used for predict and update steps: `haar` or `linear`. 
PyLops

inv [int, optional] Apply inverse transform when invoking the adjoint (True) or not (False). Note that in some scenario it may be more appropriate to use the exact inverse as adjoint of the Seislet operator even if this is not an orthogonal operator and the dot-test would not be satisfied (see Notes for details). Otherwise, the user can access the inverse directly as method of this class.

dtype [str, optional] Type of elements in input array.

Raises

NotImplementedError If kind is different from haar or linear

ValueError If sampling has more or less than two elements.

Notes

The Seislet transform [1] is implemented using the lifting scheme.

In its simplest form (i.e., corresponding to the Haar basis function for the Wavelet transform) the input dataset is separated into even (e) and odd (o) traces. Even traces are used to forward predict the odd traces using local slopes and the new odd traces (also referred to as residual) is defined as:

\[ o^{i+1} = r^i = o^i - P(e^i) \]

where \( P = P^+ \) is the slope-based forward prediction operator (which is here implemented as a sinc-based resampling). The residual is then updated and summed to the even traces to obtain the new even traces (also referred to as coarse representation):

\[ e^{i+1} = c^i = e^i + U(o^{i+1}) \]

where \( U = P^-/2 \) is the update operator which performs a slope-based backward prediction. At this point \( e^{i+1} \) becomes the new data and the procedure is repeated level times (at maximum until \( e^{i+1} \) is a single trace. The Seislet transform is effectively composed of all residuals and the coarsest data representation.

In the inverse transform the two operations are reverted. Starting from the coarsest scale data representation \( c \) and residual \( r \), the even and odd parts of the previous scale are reconstructed as:

\[ e^i = c^i - U(r^i) = e^{i+1} - U(o^{i+1}) \]

and:

\[ o^i = r^i + P(e^i) = o^{i+1} + P(e^i) \]

A new data is formed by interleaving \( e^i \) and \( o^i \) and the procedure repeated until the new data as the same number of traces as the original one.

Finally the adjoint operator can be easily derived by writing the lifting scheme in a matricial form:

\[
\begin{bmatrix}
  r_1 \\
  r_2 \\
  \vdots \\
  r_N \\
  c_1 \\
  c_2 \\
  \vdots \\
  c_N \\
\end{bmatrix} =
\begin{bmatrix}
  I & 0 & \ldots & 0 & -P & 0 & \ldots & 0 \\
  0 & I & \ldots & 0 & 0 & -P & \ldots & 0 \\
  \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & \ldots & I & 0 & 0 & \ldots & -P \\
  U & 0 & \ldots & 0 & I - UP & 0 & \ldots & 0 \\
  0 & U & \ldots & 0 & 0 & I - UP & \ldots & 0 \\
  \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & \ldots & U & 0 & 0 & \ldots & I - UP \\
\end{bmatrix}
\begin{bmatrix}
  o_1 \\
  o_2 \\
  \vdots \\
  o_N \\
  e_1 \\
  e_2 \\
  \vdots \\
  e_N \\
\end{bmatrix}
\]
Transposing the operator leads to:

\[
\begin{bmatrix}
  o_1 \\
o_2 \\
  \vdots \\
o_N \\
e_1 \\
e_2 \\
  \vdots \\
e_N
\end{bmatrix} =
\begin{bmatrix}
  I & 0 & \ldots & 0 & -U^T & 0 & \ldots & 0 \\
  0 & I & \ldots & 0 & 0 & -U^T & \ldots & 0 \\
  \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & \ldots & I & 0 & 0 & \ldots & -U^T \\
  P^T & 0 & \ldots & 0 & I - P^T U^T & 0 & \ldots & 0 \\
  0 & P^T & \ldots & 0 & 0 & I - P^T U^T & \ldots & 0 \\
  \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & \ldots & P^T & 0 & 0 & \ldots & I - P^T U^T
\end{bmatrix}
\begin{bmatrix}
  r_1 \\
r_2 \\
  \vdots \\
r_N \\
e_1 \\
e_2 \\
  \vdots \\
e_N
\end{bmatrix}
\]

which can be written more easily in the following two steps:

\[
o = r + U^H c
\]

and:

\[
e = c - P^H (r + U^H (c)) = c - P^H o
\]

Similar derivations follow for more complex wavelet bases.

**Attributes**

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

- **__init__**(slopes[, sampling, level, kind, ...]) Initialize this LinearOperator.
- **adjoint**() Hermitian adjoint.
- **apply_columns**(cols) Apply subset of columns of operator.
- **cond**(uselobpcg) Condition number of linear operator.
- **conj**() Complex conjugate operator.
- **div**(y[, niter]) Solve the linear problem \( y = Ax \).
- **dot**(x) Matrix-matrix or matrix-vector multiplication.
- **eigs**(neigs, symmetric, niter, uselobpcg) Most significant eigenvalues of linear operator.
- **inverse**(x) Matrix-matrix multiplication.
- **matmat**(X) Matrix-vector multiplication.
- **rmatmat**(X) Adjoint matrix-matrix multiplication.
- **rmatvec**(x) Adjoint matrix-vector multiplication.
- **matvec**(x) Matrix-vector multiplication.
- **todense**() Return dense matrix.
- **tosparse**() Return sparse matrix.
- **transpose**() Transpose this linear operator.

**Examples using pylops.signalprocessing.Seislet**

- sphx_glr_gallery_plot_seislet.py
- sphx_glr_gallery_plot_slopeest.py
Two dimensional Radon transform.

Apply two dimensional Radon forward (and adjoint) transform to a 2-dimensional array of size $[n_{px} \times n_t]$ (and $[n_x \times n_t]$).

In forward mode this entails to spreading the model vector along parametric curves (lines, parabolas, or hyperbolas depending on the choice of `kind`), while stacking values in the data vector along the same parametric curves is performed in adjoint mode.

**Parameters**

- `taxis` [np.ndarray] Time axis
- `haxis` [np.ndarray] Spatial axis
- `pxaxis` [np.ndarray] Axis of scanning variable $p_x$ of parametric curve
- `kind` [str, optional] Curve to be used for stacking/spreading (linear, parabolic, and hyperbolic are currently supported) or a function that takes $(x, t_0, px)$ as input and returns $t$ as output
- `centeredh` [bool, optional] Assume centered spatial axis (True) or not (False)
- `interp` [bool, optional] Apply linear interpolation (True) or nearest interpolation (False) during stacking/spreading along parametric curve
- `onthefly` [bool, optional] Compute stacking parametric curves on-the-fly as part of forward and adjoint modelling (True) or at initialization and store them in look-up table (False). Using a look-up table is computationally more efficient but increases the memory burden
- `engine` [str, optional] Engine used for computation (numpy or numba)
- `dtype` [str, optional] Type of elements in input array.

**Returns**

- `r2op` [pylops.LinearOperator] Radon operator

**Raises**

- `KeyError` If `engine` is neither numpy nor numba
- `NotImplementedError` If `kind` is not linear, parabolic, or hyperbolic

**See also:**

- `pylops.signalprocessing.Radon3D` Three dimensional Radon transform
- `pylops.Spread` Spread operator

**Notes**

The Radon2D operator applies the following linear transform in adjoint mode to the data after reshaping it into a 2-dimensional array of size $[n_x \times n_t]$ in adjoint mode:

$$m(p_x, t_0) = \int d(x, t = f(p_x, x, t)) dx$$
where \( f(p_x, x, t) = t_0 + p_x \times x \) where \( p_x = \sin(\theta)/v \) in linear mode, \( f(p_x, x, t) = t_0 + p_x \times x^2 \) in parabolic mode, and \( f(p_x, x, t) = \sqrt{t_0^2 + x^2/p_x^2} \) in hyperbolic mode.

As the adjoint operator can be interpreted as a repeated summation of sets of elements of the model vector along chosen parametric curves, the forward is implemented as spreading of values in the data vector along the same parametric curves. This operator is actually a thin wrapper around the `pylops.Spread` operator.

**Examples using `pylops.signalprocessing.Radon2D`**

- `sphx_glr_gallery_plot_sliding.py`
- `sphx_glr_gallery_plot_radon.py`
- *11. Radon filtering*
- *12. Seismic regularization*
- *16. CT Scan Imaging*

**`pylops.signalprocessing.Radon3D`**

Apply three dimensional Radon forward (and adjoint) transform to a 3-dimensional array of size \([n_{py} \times n_{px} \times n_t]\) (and \([n_y \times n_x \times n_t]\)).

In forward mode this entails to spreading the model vector along parametric curves (lines, parabolas, or hyperbolas depending on the choice of `kind`), while stacking values in the data vector along the same parametric curves is performed in adjoint mode.

**Parameters**

- `taxis` [np.ndarray] Time axis
- `hxaxis` [np.ndarray] Fast spatial axis
- `hyaxis` [np.ndarray] Slow spatial axis
- `pyaxis` [np.ndarray] Axis of scanning variable \( p_y \) of parametric curve
- `pxaxis` [np.ndarray] Axis of scanning variable \( p_x \) of parametric curve
- `kind` [str, optional] Curve to be used for stacking/spreading (linear, parabolic, and hyperbolic are currently supported)
- `centeredh` [bool, optional] Assume centered spatial axis (True) or not (False)
- `interp` [bool, optional] Apply linear interpolation (True) or nearest interpolation (False) during stacking/spreading along parametric curve
- `onthefly` [bool, optional] Compute stacking parametric curves on-the-fly as part of forward and adjoint modelling (True) or at initialization and store them in look-up table (False). Using a look-up table is computationally more efficient but increases the memory burden
- `engine` [str, optional] Engine used for computation (numpy or numba)
- `dtype` [str, optional] Type of elements in input array.

**Returns**
PyLops

r3op [pylops.LinearOperator] Radon operator

Raises

- **KeyError** If `engine` is neither `numpy` nor `numba`
- **NotImplementedError** If `kind` is not `linear`, `parabolic`, or `hyperbolic`

See also:

- `pylops.signalprocessing.Radon2D` Two dimensional Radon transform
- `pylops.Spread` Spread operator

Notes

The Radon3D operator applies the following linear transform in adjoint mode to the data after reshaping it into a 3-dimensional array of size \([n_y \times n_x \times n_t]\) in adjoint mode:

\[
m(p_y, p_x, t_0) = \int d(y, x, t = f(p_y, p_x, y, x, t)) \, dx \, dy
\]

where 

- \(f(p_y, p_x, y, x, t) = t_0 + p_y \ast y + p_x \ast x\) in linear mode,
- \(f(p_y, p_x, y, x, t) = t_0 + p_y \ast y^2 + p_x \ast x^2\) in parabolic mode,
- \(f(p_y, p_x, y, x, t) = \sqrt{t_0^2 + y^2/p_y^2 + x^2/p_x^2}\) in hyperbolic mode.

As the adjoint operator can be interpreted as a repeated summation of sets of elements of the model vector along chosen parametric curves, the forward is implemented as spreading of values in the data vector along the same parametric curves. This operator is actually a thin wrapper around the `pylops.Spread` operator.

Examples using `pylops.signalprocessing.Radon3D`

- sphx_glr_gallery_plot_sliding.py
- sphx_glr_gallery_plot_radon.py

`pylops.signalprocessing.Sliding2D`

`pylops.signalprocessing.Sliding2D` \((Op, \text{ dims}, \text{ dims}d, \text{ nwin}, \text{ nover}, \text{ tapertype}='hanning', \text{ design}=False)\)

2D Sliding transform operator.

Apply a transform operator \(Op\) repeatedly to patches of the model vector in forward mode and patches of the data vector in adjoint mode. More specifically, in forward mode the model vector is divided into patches each patch is transformed, and patches are then recomposed in a sliding window fashion. Both model and data should be 2-dimensional arrays in nature as they are internally reshaped and interpreted as 2-dimensional arrays. Each patch contains in fact a portion of the array in the first dimension (and the entire second dimension).

This operator can be used to perform local, overlapping transforms (e.g., `pylops.signalprocessing.FFT2` or `pylops.signalprocessing.Radon2D`) of 2-dimensional arrays.

Note: The shape of the model has to be consistent with the number of windows for this operator not to return an error. As the number of windows depends directly on the choice of `nwin` and `nover`, it is recommended to use `design=True` if unsure about the choice `dims` and use the number of windows printed on screen to define such input parameter.
**Warning:** Depending on the choice of \textit{nwin} and \textit{nover} as well as the size of the data, sliding windows may not cover the entire first dimension. The start and end indices of each window can be displayed using \texttt{design=True} while defining the best sliding window approach.

**Parameters**
- \texttt{Op} [\texttt{pylops.LinearOperator}] Transform operator
- \texttt{dims} [\texttt{tuple}] Shape of 2-dimensional model. Note that \texttt{dims[0]} should be multiple of the model size of the transform in the first dimension
- \texttt{dimsd} [\texttt{tuple}] Shape of 2-dimensional data
- \texttt{nwin} [\texttt{int}] Number of samples of window
- \texttt{nover} [\texttt{int}] Number of samples of overlapping part of window
- \texttt{tapertype} [\texttt{str}, optional] Type of taper (\texttt{hanning}, \texttt{cosine}, \texttt{cosinesquare} or None)
- \texttt{design} [\texttt{bool}, optional] Print number of sliding window (\texttt{True}) or not (\texttt{False})

**Returns**
- \texttt{Sop} [\texttt{pylops.LinearOperator}] Sliding operator

**Raises**
- \texttt{ValueError} Identified number of windows is not consistent with provided model shape (\texttt{dims}).

**Examples using** \texttt{pylops.signalprocessing.Sliding2D}
- \texttt{sphx	extunderscore glr	extunderscore gallery	extunderscore plot	extunderscore sliding.py}
- \texttt{12. Seismic regularization}

\texttt{pylops.signalprocessing.Sliding3D}

\texttt{pylops.signalprocessing.Sliding3D} (\texttt{Op}, \texttt{dims}, \texttt{dimsd}, \texttt{nwin}, \texttt{nover}, \texttt{nop}, \texttt{tapertype=’hanning’, design=False})

3D Sliding transform operator.

Apply a transform operator \texttt{Op} repeatedly to patches of the model vector in forward mode and patches of the data vector in adjoint mode. More specifically, in forward mode the model vector is divided into patches each patch is transformed, and patches are then recombined in a sliding window fashion. Both model and data should be 3-dimensional arrays in nature as they are internally reshaped and interpreted as 3-dimensional arrays. Each patch contains in fact a portion of the array in the first and second dimensions (and the entire third dimension).

This operator can be used to perform local, overlapping transforms (e.g., \texttt{pylops.signalprocessing.FFTN} or \texttt{pylops.signalprocessing.Radon3D}) of 3-dimensional arrays.

**Note:** The shape of the model has to be consistent with the number of windows for this operator not to return an error. As the number of windows depends directly on the choice of \texttt{nwin} and \texttt{nover}, it is recommended to use \texttt{design=True} if unsure about the choice \texttt{dims} and use the number of windows printed on screen to define such input parameter.
**Warning:** Depending on the choice of `nwin` and `nover` as well as the size of the data, sliding windows may not cover the entire first and/or second dimensions. The start and end indices of each window can be displayed using `design=True` while defining the best sliding window approach.

**Parameters**

- **Op** [`pylops.LinearOperator`] Transform operator
- **dims** [tuple] Shape of 3-dimensional model. Note that `dims[0]` and `dims[1]` should be multiple of the model sizes of the transform in the first and second dimensions
- **dimsd** [tuple] Shape of 3-dimensional data
- **nwin** [tuple] Number of samples of window
- **nover** [tuple] Number of samples of overlapping part of window
- **nop** [tuple] Number of samples in axes of transformed domain associated to spatial axes in the data
- **tapertype** [str, optional] Type of taper (`hanning`, `cosine`, `cosinesquare` or `None`)
- **design** [bool, optional] Print number sliding window (`True`) or not (`False`)

**Returns**

- **Sop** [`pylops.LinearOperator`] Sliding operator

**Raises**

- **ValueError** Identified number of windows is not consistent with provided model shape (`dims`).

**Examples using pylops.signalprocessing.Sliding3D**

- sphx_glr_gallery_plot_sliding.py

**pylops.signalprocessing.Fredholm1**

**class** `pylops.signalprocessing.Fredholm1(G, nz=1, saveGt=True, usematmul=True, dtype='float64')`

Fredholm integral of first kind.

Implement a multi-dimensional Fredholm integral of first kind. Note that if the integral is two dimensional, this can be directly implemented using `pylops.basicoperators.MatrixMult`. A multi-dimensional Fredholm integral can be performed as a `pylops.basicoperators.BlockDiag` operator of a series of `pylops.basicoperators.MatrixMult`. However, here we take advantage of the structure of the kernel and perform it in a more efficient manner.

**Parameters**

- **G** [numpy.ndarray] Multi-dimensional convolution kernel of size \( n_{slice} \times n_x \times n_y \)
- **nz** [numpy.ndarray, optional] Additional dimension of model
- **saveGt** [bool, optional] Save \( G \) and \( G^H \) to speed up the computation of adjoint (`True`) or create \( G^H \) on-the-fly (`False`). Note that `saveGt=True` will double the amount of required memory
**usematmul** [bool, optional] Use `numpy.matmul` (True) or for-loop with `numpy.dot` (False). As it is not possible to define which approach is more performant (this is highly dependent on the size of G and input arrays as well as the hardware used in the computation), we advise users to time both methods for their specific problem prior to making a choice.

**dtype** [str, optional] Type of elements in input array.

**Notes**

A multi-dimensional Fredholm integral of first kind can be expressed as

\[ d(sl, x, z) = \int G(sl, x, y)m(sl, y, z)dy \quad \forall sl = 1, n_{slice} \]

on the other hand its adjoint is expressed as

\[ m(sl, y, z) = \int G^*(sl, y, x)d(sl, x, z)dx \quad \forall sl = 1, n_{slice} \]

In discrete form, this operator can be seen as a block-diagonal matrix multiplication:

\[
\begin{bmatrix}
G_{sl1} & 0 & \ldots & 0 \\
0 & G_{sl2} & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & G_{slN}
\end{bmatrix}
\begin{bmatrix}
m_{sl1} \\
m_{sl2} \\
\vdots \\
m_{slN}
\end{bmatrix}
\]

**Attributes**

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

**Methods**

- `__init__`(G[, nz, saveGt, usematmul, dtype]) Initialize this LinearOperator.
- `adjoint`() Hermitian adjoint.
- `apply_columns`(cols) Apply subset of columns of operator
- `cond`([uselobpcg]) Condition number of linear operator.
- `conj`() Complex conjugate operator
- `div`(y[, niter]) Solve the linear problem \( y = Ax \).
- `dot`(x) Matrix-matrix or matrix-vector multiplication.
- `eigs`([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
- `matmat`(X) Matrix-matrix multiplication.
- `matvec`(x) Matrix-vector multiplication.
- `rmatmat`(X) Adjoint matrix-matrix multiplication.
- `rmatvec`(x) Adjoint matrix-vector multiplication.
- `todense`() Return dense matrix.
- `tosparse`() Return sparse matrix.
- `transpose`() Transpose this linear operator.

**Wave-Equation processing**
**PressureToVelocity** (nt, nr, dt, dr, rho, vel)  
Pressure to Vertical velocity conversion.

**UpDownComposition2D** (nt, nr, dt, dr, rho, vel)  
2D Up-down wavefield composition.

**UpDownComposition3D** (nt, nr, dt, dr, rho, vel)  
3D Up-down wavefield composition.

**MDC** (G, nt, nv[, dt, dr, twosided, fast, ...])  
Multi-dimensional convolution.

**PhaseShift** (vel, dz, nt, freq, kx[, ky, dtype])  
Phase shift operator

**Demigration** (z, x, t, srcs, recs, vel, wav,...)  
Kirchoff Demigration operator.

### pylops.waveeqprocessing.PressureToVelocity

**pylops.waveeqprocessing.PressureToVelocity** (nt, nr, dt, dr, rho, vel, nffts=(None, None, None), critical=100.0, ntaper=10, topressure=False, dtype='complex128')

Pressure to Vertical velocity conversion.

Apply conversion from pressure to vertical velocity seismic wavefield (or vertical velocity to pressure). The input model and data required by the operator should be created by flattening the a wavefield of size \((n_r \times n_t)\).

**Parameters**

- **nt** [int] Number of samples along the time axis
- **nr** [int or tuple] Number of samples along the receiver axis (or axes)
- **dt** [float] Sampling along the time axis
- **dr** [float or tuple] Sampling(s) along the receiver array
- **rho** [float] Density along the receiver array (must be constant)
- **vel** [float] Velocity along the receiver array (must be constant)
- **nffts** [tuple, optional] Number of samples along the wavenumber and frequency axes
- **critical** [float, optional] Percentage of angles to retain in obliquity factor. For example, if critical=100 only angles below the critical angle \(\sqrt{k_y^2 + k_x^2} < \frac{\omega}{vel}\) will be retained
- **ntaper** [float, optional] Number of samples of taper applied to obliquity factor around critical angle
- **topressure** [bool, optional] Perform conversion from particle velocity to pressure (True) or from pressure to particle velocity (False)
- **dtype** [str, optional] Type of elements in input array.

**Returns**

- **Cop** [pylops.LinearOperator] Pressure to particle velocity (or particle velocity to pressure) conversion operator

**See also:**

- **UpDownComposition2D** 2D Wavefield composition
- **UpDownComposition3D** 3D Wavefield composition
- **WavefieldDecomposition** Wavefield decomposition
Notes

A pressure wavefield \( p(x,t) \) can be converted into an equivalent vertical particle velocity wavefield \( v_z(x,t) \) by applying the following frequency-wavenumber dependant scaling \[1\]:

\[
v_z(k_x, \omega) = \frac{k_z}{\omega \rho} p(k_x, \omega)
\]

where the vertical wavenumber \( k_z \) is defined as \( k_z = \sqrt{\omega^2/c^2 - k_x^2} \).

Similarly a vertical particle velocity can be converted into an equivalent pressure wavefield by applying the following frequency-wavenumber dependant scaling \[1\]:

\[
p(k_x, \omega) = \frac{\omega \rho}{k_z} v_z(k_x, \omega)
\]

For 3-dimensional applications the only difference is represented by the vertical wavenumber \( k_z \), which is defined as \( k_z = \sqrt{\omega^2/c^2 - k_x^2 - k_y^2} \).

In both cases, this operator is implemented as a concatenation of a 2 or 3-dimensional forward FFT (\texttt{pylops.signalprocessing.FFT2} or \texttt{pylops.signalprocessing.FFTN}), a weighting matrix implemented via \texttt{pylops.basicprocessing.Diagonal}, and 2 or 3-dimensional inverse FFT.

Examples using \texttt{pylops.waveeqprocessing.PressureToVelocity}

- \texttt{14. Seismic wavefield decomposition}

\texttt{pylops.waveeqprocessingUpDownComposition2D}

\begin{verbatim}
pylops.waveeqprocessing.UpDownComposition2D(nt, nr, dt, dr, rho, vel, nffts=(None, None), critical=100.0, ntaper=10, scaling=1.0, dtype='complex128')
\end{verbatim}

2D Up-down wavefield composition.

Apply multi-component seismic wavefield composition from its up- and down-going constituents. The input model required by the operator should be created by flattening the separated wavefields of size \([n_r \times n_t]\) concatenated along the spatial axis.

Similarly, the data is also a flattened concatenation of pressure and vertical particle velocity wavefields.

Parameters

- \texttt{nt} [\texttt{int}] Number of samples along the time axis
- \texttt{nr} [\texttt{int}] Number of samples along the receiver axis
- \texttt{dt} [\texttt{float}] Sampling along the time axis
- \texttt{dr} [\texttt{float}] Sampling along the receiver array
- \texttt{rho} [\texttt{float}] Density along the receiver array (must be constant)
- \texttt{vel} [\texttt{float}] Velocity along the receiver array (must be constant)
- \texttt{nffts} [\texttt{tuple}, optional] Number of samples along the wavenumber and frequency axes
- \texttt{critical} [\texttt{float}, optional] Percentage of angles to retain in obliquity factor. For example, if \texttt{critical=100} only angles below the critical angle \( |k_z| < \frac{\lambda}{2\pi \text{vel}} \) will be retained.
**ntaper** [float, optional] Number of samples of taper applied to obliquity factor around critical angle

**scaling** [float, optional] Scaling to apply to the operator (see Notes for more details)

**dtype** [str, optional] Type of elements in input array.

**Returns**

**UDop** [pylops.LinearOperator] Up-down wavefield composition operator

**See also:**

*UpDownComposition3D* 3D Wavefield composition

*WavefieldDecomposition* Wavefield decomposition

**Notes**

Multi-component seismic data \( p(x,t) \) and \( v_z(x,t) \) can be synthesized in the frequency-wavenumber domain as the superposition of the up- and downgoing constituents of the pressure wavefield \( p^-(x,t) \) and \( p^+(x,t) \) as follows [1]:

\[
\begin{bmatrix}
p(k_x,\omega) \\
v_z(k_x,\omega)
\end{bmatrix} = \begin{bmatrix}
\frac{k_z}{\omega}\rho & -\frac{k_z}{\omega}\rho \\
\end{bmatrix} \begin{bmatrix}
p^-(k_x,\omega) \\
p^+(k_x,\omega)
\end{bmatrix}
\]

where the vertical wavenumber \( k_z \) is defined as \( k_z = \sqrt{\omega^2/c^2 - k_x^2} \).

We can write the entire composition process in a compact matrix-vector notation as follows:

\[
\begin{bmatrix}
p \\
s \ast v_z
\end{bmatrix} = \begin{bmatrix}
F & 0 \\
0 & s \ast F
\end{bmatrix} \begin{bmatrix}
I & I \\
W^+ & W^-
\end{bmatrix} \begin{bmatrix}
F^H & 0 \\
0 & F^H
\end{bmatrix} \begin{bmatrix}
p^+ \\
p^-
\end{bmatrix}
\]

where \( F \) is the 2-dimensional FFT (pylops.signalprocessing.FFT2), \( W^\pm \) are weighting matrices which contain the scalings \( \pm \frac{k_z}{\omega}\rho \) implemented via pylops.basicprocessing.Diagonal, and \( s \) is a scaling factor that is applied to both the particle velocity data and to the operator has shown above. Such a scaling is required to balance out the different dynamic range of pressure and particle velocity when solving the wavefield separation problem as an inverse problem.

As the operator is effectively obtained by chaining basic PyLops operators the adjoint is automatically implemented for this operator.

**Examples using pylops.waveeqprocessingUpDownComposition2D**

- 14. Seismic wavefield decomposition

**pylops.waveeqprocessingUpDownComposition3D**

**pylops.waveeqprocessingUpDownComposition3D**(nt, nr, dr, rho, vel, nffts=(None, None, None), critical=100.0, ntaper=10, scaling=1.0, dtype='complex128')

3D Up-down wavefield composition.

Apply multi-component seismic wavefield composition from its up- and down-going constituents. The input model required by the operator should be created by flattening the separated wavefields of size \([n_{rx} \times n_{rz} \times n_t]\) concatenated along the first spatial axis.

Similarly, the data is also a flattened concatenation of pressure and vertical particle velocity wavefields.
Parameters

- **nt** [int] Number of samples along the time axis
- **nr** [tuple] Number of samples along the receiver axes
- **dt** [float] Sampling along the time axis
- **dr** [tuple] Samplings along the receiver array
- **rho** [float] Density along the receiver array (must be constant)
- **vel** [float] Velocity along the receiver array (must be constant)
- **nffts** [tuple, optional] Number of samples along the wavenumbers and frequency axes (for the wavenumbers axes the same order as **nr** and **dr** must be followed)
- **critical** [float, optional] Percentage of angles to retain in obliquity factor. For example, if critical=100 only angles below the critical angle $\sqrt{k_y^2 + k_x^2} < \frac{\omega}{v_e}$ will be retained
- **ntaper** [float, optional] Number of samples of taper applied to obliquity factor around critical angle
- **scaling** [float, optional] Scaling to apply to the operator (see Notes for more details)
- **dtype** [str, optional] Type of elements in input array.

Returns

- **UDop** [pylops.LinearOperator] Up-down wavefield composition operator

See also:

- **UpDownComposition2D** 2D Wavefield composition
- **WavefieldDecomposition** Wavefield decomposition

Notes

Multi-component seismic data ($p(y, x, t)$ and $v_z(y, x, t)$) can be synthesized in the frequency-wavenumber domain as the superposition of the up- and downgoing constituents of the pressure wavefield ($p^-(y, x, t)$ and $p^+(y, x, t)$) as described `pylops.waveeqprocessing.UpDownComposition2D`.

Here the vertical wavenumber $k_z$ is defined as $k_z = \sqrt{\omega^2/c^2 - k_y^2 - k_x^2}$.

**pylops.waveeqprocessing.MDC**

Apply multi-dimensional convolution between two datasets. If transpose=True, model and data should be provided after flattening 2- or 3-dimensional arrays of size $[n_t \times n_{vs} \times n_t]$ and $[n_s \times n_{vs} \times n_t]$ (or $2 \times n_t - 1$ for twosided=True), respectively. If transpose=False, model and data should be provided after flattening 2- or 3-dimensional arrays of size $[n_t \times n_{vs} \times n_{vs}]$ and $[n_t \times n_s \times n_{vs}]$ (or $2 \times n_t - 1$ for twosided=True), respectively.
**Warning:** A new implementation of MDC is provided in v1.5.0. This currently affects only the inner working of the operator and end-users can use the operator in the same way as they used to do with the previous one. Nevertheless, it is now recommend to use the operator with `transpose=False`, as this behaviour will become default in version v2.0.0 and the behaviour with `transpose=True` will be deprecated.

**Parameters**

- **G** [numpy.ndarray] Multi-dimensional convolution kernel in frequency domain of size \([n_s \times n_r \times n_{f_{max}}]\) if `transpose=True` or size \([n_{f_{max}} \times n_s \times n_r]\) if `transpose=False`
- **nt** [int] Number of samples along time axis
- **nv** [int] Number of samples along virtual source axis
- **dt** [float, optional] Sampling of time integration axis
- **dr** [float, optional] Sampling of receiver integration axis
- **twosided** [bool, optional] MDC operator has both negative and positive time (True) or only positive (False)
- **fast** [bool, optional] Deprecated, will be removed in v2.0.0
- **dtype** [str, optional] Deprecated, will be removed in v2.0.0
- **fftengine** [str, optional] Engine used for fft computation (numpy or fftw)
- **transpose** [bool, optional] Transpose G and inputs such that time/frequency is placed in first dimension. This allows back-compatibility with v1.4.0 and older but will be removed in v2.0.0 where time/frequency axis will be required to be in first dimension for efficiency reasons.
- **saveGt** [bool, optional] Save G and G^H to speed up the computation of adjoint of pylops.signalprocessing.Fredholm1 (True) or create G^H on-the-fly (False) Note that saveGt=True will be faster but double the amount of required memory
- **conj** [str, optional] Perform Fredholm integral computation with complex conjugate of G
- **usematmul** [bool, optional] Use numpy.matmul (True) or for-loop with numpy.dot (False) in pylops.signalprocessing.Fredholm1 operator. Refer to Fredholm1 documentation for details.
- **prescaled** [bool, optional] Apply scaling to kernel (False) or not (False) when performing spatial and temporal summations. In case prescaled=True, the kernel is assumed to have been pre-scaled when passed to the MDC routine.

**Raises**

- **ValueError** If nt is even and twosided=True

**See also:**

- **MDD** Multi-dimensional deconvolution

**Notes**

The so-called multi-dimensional convolution (MDC) is a chained operator [1]. It is composed of a forward Fourier transform, a multi-dimensional integration, and an inverse Fourier transform:

\[
y(t, s, v) = F^{-1} \left( \int_{S} G(f, s, r) F(x(t, r, v)) dr \right)
\]
which is discretized as follows:

\[ y(t, s, v) = F^{-1}\left(\sum_{i_r=0}^{n_r} (\sqrt{n_t} \ast d_t \ast d_r)G(f, s, i_r)F(x(t, i_r, v))\right) \]

where \((\sqrt{n_t} \ast d_t \ast d_r)\) is not applied if prescaled=True.

This operation can be discretized and performed by means of a linear operator

\[ D = F^H G F \]

where \(F\) is the Fourier transform applied along the time axis and \(G\) is the multi-dimensional convolution kernel.

Examples using \texttt{pylops.waveeqprocessing.MDC}

- \texttt{sphx\_glr\_gallery\_plot\_mdc.py}
- \texttt{09. Multi-Dimensional Deconvolution}

\texttt{pylops.waveeqprocessing.PhaseShift}

\texttt{pylops.waveeqprocessing.PhaseShift} \((\texttt{vel}, \texttt{dz}, \texttt{nt}, \texttt{freq}, \texttt{kx}, \texttt{ky}=\texttt{None}, \texttt{dtype}=\texttt{'float64'})\)

Phase shift operator

Apply positive (forward) phase shift with constant velocity in forward mode, and negative (backward) phase shift with constant velocity in adjoint mode. Input model and data should be 2- or 3-dimensional arrays in time-space domain of size \([n_t \times n_x (\times n_y)]\).

Parameters

- \texttt{vel} \texttt{[float, optional]} Constant propagation velocity
- \texttt{dz} \texttt{[float, optional]} Depth step
- \texttt{nt} \texttt{[int, optional]} Number of time samples of model and data
- \texttt{freq} \texttt{[numpy.ndarray]} Positive frequency axis
- \texttt{kx} \texttt{[int, optional]} Horizontal wavenumber axis (centered around 0) of size \([n_x \times 1]\).
- \texttt{ky} \texttt{[int, optional]} Second horizontal wavenumber axis for 3d phase shift (centered around 0) of size \([n_y \times 1]\).
- \texttt{dtype} \texttt{[str, optional]} Type of elements in input array

Returns

\texttt{Pop} \texttt{[pylops.LinearOperator]} Phase shift operator

Notes

The phase shift operator implements a one-way wave equation forward propagation in frequency-wavenumber domain by applying the following transformation to the input model:

\[ d(f, k_x, k_y) = m(f, k_x, k_y) \ast e^{-j\sqrt{\omega^2/v^2-k_x^2-k_y^2}\Delta z} \]

where \(v\) is the constant propagation velocity and \(\Delta z\) is the propagation depth. In adjoint mode, the data is propagated backward using the following transformation:

\[ m(f, k_x, k_y) = d(f, k_x, k_y) \ast e^{j\sqrt{\omega^2/v^2-k_x^2-k_y^2}\Delta z} \]
Effectively, the input model and data are assumed to be in time-space domain and forward Fourier transform is applied to both dimensions, leading to the following operator:

\[ d = F_t^H F_x^H P F_x F_t m \]

where \( P \) performs the phase-shift as discussed above.

**Examples using `pylops.waveeqprocessing.PhaseShift`**

- sphx_glr_gallery_plot_phaseshift.py

**`pylops.waveeqprocessing.Demigration`**

`pylops.waveeqprocessing.Demigration(z, x, t, srcs, recs, vel, wav, wavcenter, y=None, trav=None, mode='eikonal')`

Kirchoff Demigration operator.

Traveltime based seismic demigration/migration operator.

**Parameters**

- `z` [numpy.ndarray] Depth axis
- `x` [numpy.ndarray] Spatial axis
- `t` [numpy.ndarray] Time axis for data
- `srcs` [numpy.ndarray] Sources in array of size \([2/3 \times n_s]\) The first axis should be ordered as \((y,) x, z\).
- `recs` [numpy.ndarray] Receivers in array of size \([2/3 \times n_r]\) The first axis should be ordered as \((y,) x, z\).
- `vel` [numpy.ndarray or float] Velocity model of size \([ (n_y \times n_x \times n_z) \] (or constant)
- `wav` [numpy.ndarray] Wavelet
- `wavcenter` [int] Index of wavelet center
- `y` [numpy.ndarray] Additional spatial axis (for 3-dimensional problems)
- `mode` [str, optional] Computation mode (analytic, eikonal or byot, see Notes for more details)
- `trav` [numpy.ndarray, optional] Traveltime table of size \([ (n_y \times n_x \times n_z) \times n_r] \) (to be provided if `mode='byot'`)  

**Returns**

- `demop` [pylops.LinearOperator] Demigration/Migration operator

**Raises**

- `NotImplementedError` If `mode` is neither analytic, eikonal, or byot

**Notes**

The demigration operator synthetizes seismic data given from a propagation velocity model \(v\) and a reflectivity model \(m\). In forward mode:

\[
d(x_r, x_s, t) = w(t) \ast \int_V G(x, x_s, t) G(x_r, x, t) m(x) dx
\]
where $m(x)$ is the model and it represents the reflectivity at every location in the subsurface, $G(x, x_s, t)$ and $G(x_r, x_r, t)$ are the Green’s functions from source-to-subsurface-to-receiver and finally $w(t)$ is the wavelet. Depending on the choice of mode the Green’s function will be computed and applied differently:

- **mode=analytic or mode=eikonal**: traveltime curves between source to receiver pairs are computed for every subsurface point and Green’s functions are implemented from traveltime look-up tables, placing the reflectivity values at corresponding source-to-receiver time in the data.
- **byot**: bring your own table. Traveltime table provided directly by user using trav input parameter. Green’s functions are then implemented in the same way as previous options.

The adjoint of the demigration operator is a migration operator which projects data in the model domain creating an image of the subsurface reflectivity.

**Geophysical subsurface characterization**

- **avo.AVOLinearModelling**
  - Parameters:
    - `theta` [np.ndarray] Incident angles in degrees
    - `vsvp` [np.ndarray or float] VS/VP ratio
    - `nt0` [int, optional] number of samples (if vsvp is a scalar)
    - `spatdims` [int or tuple, optional] Number of samples along spatial axis (or axes) (None if only one dimension is available)
    - `linearization` [str, optional] choice of linearization, akirich: Aki-Richards, fatti: Fatti
    - `dtype` [str, optional] Type of elements in input array.
  - Raises: NotImplementedError If `linearization` is not an implemented linearization

**pylops.avoavo.AVOLinearModelling**

- **class pylops.avo.avo.AVOLinearModelling**
  - Parameters:
    - `theta` [np.ndarray] Incident angles in degrees
    - `vsvp` [0.5, nt0=1, spatdims=None, linearization='akirich', dtype='float64']
  - Create operator to be applied to a combination of elastic parameters for generation of seismic pre-stack reflectivity.

The demigration operator requires the Green’s function $G(x, x_s, t)$ or $G(x_r, x_r, t)$ to compute the migration operator. Depending on the choice of mode the Green’s function will be computed and applied differently:

- **mode=analytic or mode=eikonal**: traveltime curves between source to receiver pairs are computed for every subsurface point and Green’s functions are implemented from traveltime look-up tables, placing the reflectivity values at corresponding source-to-receiver time in the data.
- **byot**: bring your own table. Traveltime table provided directly by user using trav input parameter. Green’s functions are then implemented in the same way as previous options.

The adjoint of the demigration operator is a migration operator which projects data in the model domain creating an image of the subsurface reflectivity.
Notes

The AVO linearized operator performs a linear combination of three (or two) elastic parameters arranged in input vector \( \mathbf{m} \) of size \( n_{t0} \times N \) to create the so-called seismic reflectivity:

\[
 r(t, \theta, x, y) = \sum_{i=1}^{N} G_i(t, \theta) m_i(t, x, y) \quad \forall \ t, \theta
\]

where \( N = 2/3 \). Note that the reflectivity can be in 1d, 2d or 3d and \( \text{spatdims} \) contains the dimensions of the spatial axis (or axes) \( x \) and \( y \).

Attributes

- **shape** [tuple] Operator shape
- **explicit** [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

- **__init__**(theta[, vsvp, nt0, spatdims, ...]) Initialize this LinearOperator.
- **adjoint()** Hermitian adjoint.
- **apply_columns(cols)** Apply subset of columns of operator
- **cond([uselobpcg])** Condition number of linear operator.
- **conj()** Complex conjugate operator
- **div(y[, niter])** Solve the linear problem \( y = Ax \).
- **dot(x)** Matrix-matrix or matrix-vector multiplication.
- **eigs([neigs, symmetric, niter, uselobpcg])** Most significant eigenvalues of linear operator.
- **matmat(X)** Matrix-matrix multiplication.
- **matvec(x)** Matrix-vector multiplication.
- **rmatmat(X)** Adjoint matrix-matrix multiplication.
- **rmatvec(x)** Adjoint matrix-vector multiplication.
- **todense()** Return dense matrix.
- **tospars()** Return sparse matrix.
- **transpose()** Transpose this linear operator.

Examples using pylops.avo.avo.AVOLinearModelling

- sphx_glr_gallery_plot_avo.py

pylops.avo.poststack.PoststackLinearModelling

pylops.avo.poststack.\texttt{PoststackLinearModelling}(wav, nt0, spatdims=\texttt{None}, explicit=\texttt{False}, sparse=\texttt{False})

Post-stack linearized seismic modelling operator.

Create operator to be applied to an elastic parameter trace (or stack of traces) for generation of band-limited seismic post-stack data. The input model and data have shape \([n_{t0} \times n_x \times n_y]\).

Parameters

- **wav** [np.ndarray] Wavelet in time domain (must have odd number of elements and centered to zero). If 1d, assume stationary wavelet for the entire time axis. If 2d, use as non-stationary
wavelet (user must provide one wavelet per time sample in an array of size \(nt0 \times n_{\text{wav}}\) where \(n_{\text{wav}}\) is the length of each wavelet)

- \textbf{nt0} \ [\text{int}] Number of samples along time axis
- \textbf{spatdims} \ [\text{int} \text{ or } \text{tuple}, \text{optional}] Number of samples along spatial axis (or axes) (\text{None} if only one dimension is available)
- \textbf{explicit} \ [\text{bool}, \text{optional}] Create a chained linear operator (\text{False}, preferred for large data) or a \text{MatrixMult} linear operator with dense matrix (\text{True}, preferred for small data)
- \textbf{sparse} \ [\text{bool}, \text{optional}] Create a sparse matrix (\text{True}) or dense (\text{False}) when \text{explicit}=\text{True}

\textbf{Returns}

- \textbf{P} \ [\text{LinearOperator}] post-stack modelling operator.

\textbf{Raises}

- \textbf{ValueError} If \text{wav} is two dimensional but does not contain \text{nt0} wavelets

\textbf{Notes}

Post-stack seismic modelling is the process of constructing seismic post-stack data from a profile of an elastic parameter of choice in time (or depth) domain. This can be easily achieved using the following forward model:

\[
d(t, \theta) = w(t) * \frac{dln(m(t))}{dt}
\]

where \(m(t)\) is the elastic parameter profile and \(w(t)\) is the time domain seismic wavelet. In compact form:

\[
d = WDm
\]

In the special case of acoustic impedance \((m(t) = AI(t))\), the modelling operator can be used to create zero-offset data:

\[
d(t, \theta = 0) = \frac{1}{2} w(t) * \frac{dln(m(t))}{dt}
\]

where the scaling factor \(\frac{1}{2}\) can be easily included in the wavelet.

\textbf{Examples using pylops.avo.poststack.PoststackLinearModelling}

- \textit{07. Post-stack inversion}

\textbf{pylops.avo.prestack.PrestackLinearModelling}

\textbf{pylops.avo.prestack.PrestackLinearModelling}(wav, theta, vsvp=0.5, nt0=1, spatdims=None, linearization='akirich', explicit=False)

Pre-stack linearized seismic modelling operator.

Create operator to be applied to elastic property profiles for generation of band-limited seismic angle gathers from a linearized version of the Zoeppritz equation.

\textbf{Parameters}
Pre-stack linearized seismic modelling operator for wavelet.

Create operator to be applied to a wavelet for generation of band-limited seismic angle gathers using a linearized version of the Zoeppritz equation.

**Parameters**

- `m` [np.ndarray] elastic parameter profiles of size $[n_{t0} \times N]$ where $N = 3/2$
theta [int] Incident angles in degrees
nwav [np.ndarray] Number of samples of wavelet to be applied/estimated
wavc [int, optional] Index of the center of the wavelet
vsvp [np.ndarray or float, optional] VS/VP ratio
linearization [str, optional] choice of linearization, akirich: Aki-Richards, fatti: Fatt

Returns

Raises
NotImplementedError If linearization is not an implemented linearization

Notes
Pre-stack seismic modelling for wavelet estimate is the process of constructing seismic reflectivities using three (or two) profiles of elastic parameters in time (or depth) domain arranged in an input vector \( \mathbf{m} \) of size \( n_t \times N \):

\[
d(t, \theta) = \sum_{i=1}^{N} G_i(t, \theta) m_i(t) * w(t)
\]

where \( w(t) \) is the time domain seismic wavelet. In compact form:

\[
d = Gw
\]

On the other hand, pre-stack wavelet estimation aims at recovering the wavelet given knowledge of the band-limited seismic pre-stack data and the elastic parameter profiles.

Examples using pylops.avo.prestack.PrestackWaveletModelling

- sphx_glr_gallery_plot_wavest.py

3.6.2 Solvers

Least-squares

leastsquares.NormalEquationsInversion(Op)
leastsquares.RegularizedInversion(Op, Regs, data)
leastsquares.PreconditionedInversion(Op, Preconditioned inversion, P, data)
pylops.optimization.leastsquares.NormalEquationsInversion

```python
pylops.optimization.leastsquares.NormalEquationsInversion(Op, Regs, data, Weight=None, dataregs=None, epsI=0, epsRs=None, x0=None, returninfo=False, NRegs=None, epsNRs=None, **kwargs_cg)
```

Inversion of normal equations.
Solve the regularized normal equations for a system of equations given the operator Op, a data weighting operator Weight and optionally a list of regularization terms Regs and/or NRegs.

**Parameters**
- **Op** [pylops.LinearOperator] Operator to invert
- **Regs** [list] Regularization operators (None to avoid adding regularization)
- **data** [numpy.ndarray] Data
- **Weight** [pylops.LinearOperator, optional] Weight operator
- **dataregs** [list, optional] Regularization data (must have the same number of elements as Regs)
- **epsI** [float, optional] Tikhonov damping
- **epsRs** [list, optional] Regularization dampings (must have the same number of elements as Regs)
- **x0** [numpy.ndarray, optional] Initial guess
- **returninfo** [bool, optional] Return info of CG solver
- **NRegs** [list] Normal regularization operators (None to avoid adding regularization). Such operators must apply the chain of the forward and the adjoint in one go. This can be convenient in cases where a faster implementation is available compared to applying the forward followed by the adjoint.
- **epsNRs** [list, optional] Regularization dampings for normal operators (must have the same number of elements as NRegs)
- **kwargs_cg** Arbitrary keyword arguments for scipy.sparse.linalg.cg solver

**Returns**
- **xinv** [numpy.ndarray] Inverted model.
- **istop** [int] Convergence information:
  - 0: successful exit
  - >0: convergence to tolerance not achieved, number of iterations
  - <0: illegal input or breakdown

**See also:**
- RegularizedInversion Regularized inversion
- PreconditionedInversion Preconditioned inversion
Notes

Solve the following normal equations for a system of regularized equations given the operator \( \text{Op} \), a data weighting operator \( \text{W} \), a list of regularization terms \( (R_i \text{ and/or } N_i) \), the data \( \text{d} \) and regularization data \( \text{d}_R \), and the damping factors \( \epsilon_I, \epsilon_R, \text{ and } \epsilon_N \):

\[
(\text{Op}^T \text{W} \text{Op} + \sum_i \epsilon_{R_i}^2 R_i^T R_i + \sum_i \epsilon_{N_i}^2 N_i + \epsilon_I^2 I)x = \text{Op}^T \text{W} \text{d} + \sum_i \epsilon_{R_i}^2 R_i^T \text{d}_R.
\]

Note that the data term of the regularizations \( N_i \) is implicitly assumed to be zero.

Examples using `pylops.optimization.leastsquares.NormalEquationsInversion`

- sphx_glr_gallery_plot_bilinear.py
- 03. Solvers
- 05. Image deblurring
- 06. 2D Interpolation

`pylops.optimization.leastsquares.RegularizedInversion`

`pylops.optimization.leastsquares.RegularizedInversion` \((\text{Op}, \text{Regs}, \text{data}, \text{Weight=none}, \text{dataregs=none}, \text{epsRs=none}, \text{x0=none}, \text{returninfo=false}, **\text{kwargs_lsqr})\)

Regularized inversion.

Solve a system of regularized equations given the operator \( \text{Op} \), a data weighting operator \( \text{Weight} \), and a list of regularization terms \( \text{Regs} \).

**Parameters**

- **Op** \([\text{pylops.LinearOperator}]\) Operator to invert
- **Regs** \([\text{list}]\) Regularization operators (None to avoid adding regularization)
- **data** \([\text{numpy.ndarray}]\) Data
- **Weight** \([\text{pylops.LinearOperator}, \text{optional}]\) Weight operator
- **dataregs** \([\text{list}, \text{optional}]\) Regularization data (if None a zero data will be used for every regularization operator in \( \text{Regs} \))
- **epsRs** \([\text{list}, \text{optional}]\) Regularization dampings
- **x0** \([\text{numpy.ndarray}, \text{optional}]\) Initial guess
- **returninfo** \([\text{bool}, \text{optional}]\) Return info of LSQR solver
- **kwargs_lsqr** Arbitrary keyword arguments for `scipy.sparse.linalg.lsqr` solver

**Returns**

- **xin** \([\text{numpy.ndarray}]\) Inverted model \( \text{Op} \)
- **istop** \([\text{int}]\) Gives the reason for termination
  
  1 means \( x \) is an approximate solution to \( \text{d} = \text{Op}x \)
  
  2 means \( x \) approximately solves the least-squares problem
**PyLops**

- `itn` [int] Iteration number upon termination
- `r1norm` [float] $||r||_2^2$, where $r = d - Op x$
- `r2norm` [float] $\sqrt{r^T r + \epsilon^2 x^T x}$. Equal to `r1norm` if $\epsilon = 0$

See also:

- `RegularizedOperator` Regularized operator
- `NormalEquationsInversion` Normal equations inversion
- `PreconditionedInversion` Preconditioned inversion

**Notes**

Solve the following system of regularized equations given the operator $Op$, a data weighting operator $W^{1/2}$, a list of regularization terms $R_i$, the data $d$ and regularization damping factors $\epsilon_{\ell_1}$ and $\epsilon_{R_i}$:

$$
\begin{bmatrix}
W^{1/2}Op \\
\epsilon_{R_1}R_1 \\
\vdots \\
\epsilon_{R_N}R_N
\end{bmatrix}
\begin{bmatrix}
x \\
\epsilon_{R_1}d_{R_1} \\
\vdots \\
\epsilon_{R_N}d_{R_N}
\end{bmatrix}
$$

where the weight provided here is equivalent to the square-root of the weight in `pylops.optimization.least squares.NormalEquationsInversion`. Note that this system is solved using the `scipy.sparse.linalg.lsqr` and an initial guess $x_0$ can be provided to this solver, despite the original solver does not allow so.

**Examples using `pylops.optimization.least squares-RegularizedInversion`**

- sphx_glr_gallery_plot_ista.py
- sphx_glr_gallery_plot_tvreg.py
- sphx_glr_gallery_plot_wavest.py
- 03. Solvers
- 06. 2D Interpolation
- 16. CT Scan Imaging

**pylops.optimization.least squares.PreconditionedInversion**

`pylops.optimization.least squares.PreconditionedInversion` ($Op$, $P$, $data$, $x0=None$, `returninfo=False`, `**kwargs_lsqr`)

Preconditioned inversion.

Solve a system of preconditioned equations given the operator $Op$ and a preconditioner $P$.

**Parameters**

- `Op` [pylops.LinearOperator] Operator to invert
- `P` [pylops.LinearOperator] Preconditioner
- `data` [numpy.ndarray] Data
**PreconditionedInversion**

- **x0** [numpy.ndarray] Initial guess
- **returninfo** [bool] Return info of LSQR solver
- **kwargs_lsqr** Arbitrary keyword arguments for scipy.sparse.linalg.lsqr solver

**Returns**

- **xinv** [numpy.ndarray] Inverted model.
- **istop** [int] Gives the reason for termination
  - 1 means \( x \) is an approximate solution to \( d = Opx \)
  - 2 means \( x \) approximately solves the least-squares problem
- **itn** [int] Iteration number upon termination
- **r1norm** [float] \(||r||_2^2\), where \( r = d - Opx \)
- **r2norm** [float] \( \sqrt{r^T r + \epsilon^2 x^T x} \). Equal to r1norm if \( \epsilon = 0 \)

**See also:**

- **RegularizedInversion** Regularized inversion
- **NormalEquationsInversion** Normal equations inversion

**Notes**

Solve the following system of preconditioned equations given the operator \( Op \), a preconditioner \( P \), the data \( d \)

\[
    d = Op(Pp)
\]

where \( p \) is the solution in the preconditioned space and \( x = Pp \) is the solution in the original space.

**Examples using pylops.optimization.leastsquares.PreconditionedInversion**

- sphx_glr_gallery_plot_wavelet.py
- 03. Solvers

**Sparsity**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sparsity.IRLS</strong>(Op, data, nouter[, threshR, ...])</td>
<td>Iteratively reweighted least squares.</td>
</tr>
<tr>
<td><strong>sparsity.OMP</strong>(Op, data[, niter_outer, ...])</td>
<td>Orthogonal Matching Pursuit (OMP).</td>
</tr>
<tr>
<td><strong>sparsity.ISTA</strong>(Op, data, niter[, eps, alpha, ...])</td>
<td>Iterative Shrinkage-Thresholding Algorithm (ISTA).</td>
</tr>
<tr>
<td><strong>sparsity.FISTA</strong>(Op, data, niter[, eps, ...])</td>
<td>Fast Iterative Shrinkage-Thresholding Algorithm (FISTA).</td>
</tr>
<tr>
<td><strong>sparsity.SPGL1</strong>(Op, data[, SOp, tau, sigma, x0])</td>
<td>Spectral Projected-Gradient for L1 norm.</td>
</tr>
<tr>
<td><strong>sparsity.SplitBregman</strong>(Op, RegsL1, data[, ...])</td>
<td>Split Bregman for mixed L2-L1 norms.</td>
</tr>
</tbody>
</table>
Iteratively reweighted least squares.

Solve an optimization problem with $L_1$ cost function (data IRLS) or $L_1$ regularization term (model IRLS) given the operator $\mathbf{O}$ and data $\mathbf{y}$.

In the data IRLS, the cost function is minimized by iteratively solving a weighted least squares problem with the weight at iteration $i$ being based on the data residual at iteration $i-1$. This IRLS solver is robust to outliers since the $L_1$ norm gives less weight to large residuals than $L_2$ norm does.

Similarly in the model IRLS, the weight at iteration $i$ is based on the model at iteration $i-1$. This IRLS solver inverts for a sparse model vector.

**Parameters**

- $\mathbf{O}$ [pylops.LinearOperator] Operator to invert
- $\mathbf{d}$ [numpy.ndarray] Data
- nouter [int] Number of outer iterations
- threshR [bool, optional] Apply thresholding in creation of weight (True) or damping (False)
- epsR [float, optional] Damping to be applied to residuals for weighting term
- epsI [float, optional] Tikhonov damping
- x0 [numpy.ndarray, optional] Initial guess
- tolIRLS [float, optional] Tolerance. Stop outer iterations if difference between inverted model at subsequent iterations is smaller than tolIRLS
- returnhistory [bool, optional] Return history of inverted model for each outer iteration of IRLS
- kind [str, optional] Kind of solver (data or model)
- **kwargs_solver Arbitrary keyword arguments for scipy.sparse.linalg.cg solver for data IRLS and scipy.sparse.linalg.lsqr solver for model IRLS

**Returns**

- xinv [numpy.ndarray] Inverted model
- nouter [int] Number of effective outer iterations
- xinv_hist [numpy.ndarray, optional] History of inverted model
- rw_hist [numpy.ndarray, optional] History of weights

**Notes**

Data IRLS solves the following optimization problem for the operator $\mathbf{O}$ and the data $\mathbf{d}$:

$$ J = ||\mathbf{d} - \mathbf{Ox}||_1 $$

by a set of outer iterations which require to repeatedly solve a weighted least squares problem of the form:

$$ \mathbf{x}^{(i+1)} = \arg \min_{\mathbf{x}} ||\mathbf{d} - \mathbf{Ox}||^2_{R(i)} + \epsilon_i^2 ||\mathbf{x}||_2^2 $$

208 Chapter 3. History
where $R^{(i)}$ is a diagonal weight matrix whose diagonal elements at iteration $i$ are equal to the absolute inverses of the residual vector $r^{(i)} = y - Op^i x^{(i)}$ at iteration $i$. More specifically the $j$-th element of the diagonal of $R^{(i)}$ is

$$R^{(i)}_{j,j} = \frac{1}{|r^{(i)}_j| + \epsilon_R}$$

or

$$R^{(i)}_{j,j} = \frac{1}{\max(|r^{(i)}_j|, \epsilon_R)}$$

depending on the choice $\text{threshR}$. In either case, $\epsilon_R$ is the user-defined stabilization/thresholding factor [1].

Similarly $\text{model IRLS}$ solves the following optimization problem for the operator $Op$ and the data $d$:

$$J = ||x||_1 \quad s.t. \quad d = Op^i x$$

by a set of outer iterations which require to repeatedly solve a weighted least squares problem of the form:

$$x^{(i+1)} = \arg \min_x ||x||_2^2 \quad s.t. \quad d = Op^i x$$

where $R^{(i)}$ is a diagonal weight matrix whose diagonal elements at iteration $i$ are equal to the absolute inverses of the model vector $x^{(i)}$ at iteration $i$. More specifically the $j$-th element of the diagonal of $R^{(i)}$ is

$$R^{(i)}_{j,j} = |x^{(i)}_j|$$

Examples using pylops.optimization.sparsity.IRLS

- sphx_glr_gallery_plot_linearregr.py
- sphx_glr_gallery_plot_ista.py
- sphx_glr_gallery_plot_regr.py

pylops.optimization.sparsity.OMP

pylops.optimization.sparsity.OMP ($Op$, $data$, $niter_outer=10$, $niter_inner=40$, $sigma=0.0001$, $normalizecols=False$, $show=False$)

Orthogonal Matching Pursuit (OMP).

Solve an optimization problem with $L0$ regularization function given the operator $Op$ and data $y$. The operator can be real or complex, and should ideally be either square $N = M$ or underdetermined $N < M$.

Parameters

- $Op$ [pylops.LinearOperator] Operator to invert
- $data$ [numpy.ndarray] Data
- $niter_outer$ [int, optional] Number of iterations of outer loop
- $niter_inner$ [int, optional] Number of iterations of inner loop. By choosing $niter_inner=0$, the Matching Pursuit (MP) algorithm is implemented.
- $sigma$ [list] Maximum L2 norm of residual. When smaller stop iterations.
**normalizecols** [list, optional] Normalize columns (True) or not (False). Note that this can be expensive as it requires applying the forward operator \( n_{cols} \) times to unit vectors (i.e., containing 1 at position \( j \) and zero otherwise); use only when the columns of the operator are expected to have highly varying norms.

**show** [bool, optional] Display iterations log

**Returns**

- \( x_{inv} \) [numpy.ndarray] Inverted model
- \( iiter \) [int] Number of effective outer iterations
- \( cost \) [numpy.ndarray, optional] History of cost function

**See also:**

- **ISTA** Iterative Shrinkage-Thresholding Algorithm (ISTA).
- **FISTA** Fast Iterative Shrinkage-Thresholding Algorithm (FISTA).
- **SPGL1** Spectral Projected-Gradient for L1 norm (SPGL1).
- **SplitBregman** Split Bregman for mixed L2-L1 norms.

**Notes**

Solves the following optimization problem for the operator \( \text{Op} \) and the data \( d \):

\[
\|x\|_0 \quad \text{subj. to} \quad \|\text{Op}x - b\|_2^2 \leq \sigma,
\]

using Orthogonal Matching Pursuit (OMP). This is a very simple iterative algorithm which applies the following step:

\[
\Lambda_k = \Lambda_{k-1} \cup \{ \text{argmax}_j |\text{Op}^H_j r_k| \}
\]

\[
x_k = \{ \text{argmin}_x \|\text{Op}_{\Lambda_k} x - b\|_2^2 \}
\]

Note that by choosing \( \text{niter\_inner} = 0 \) the basic Matching Pursuit (MP) algorithm is implemented instead. In other words, instead of solving an optimization at each iteration to find the best \( x \) for the currently selected basis functions, the vector \( x \) is just updated at the new basis function by taking directly the value from the inner product \( \text{Op}^H_j r_k \).

In this case it is highly recommend to provide a normalized basis function. If different basis have different norms, the solver is likely to diverge. Similar observations apply to OMP, even though mild unbalancing between the basis is generally properly handled.

**Examples using pylops.optimization.sparsity.OMP**

- sphx_glr_gallery_plot_ista.py

**pylops.optimization.sparsity.ISTA**

**pylops.optimization.sparsity.ISTA**(\( \text{Op}, \text{data}, \text{niter}, \text{eps}=0.1, \text{alpha}=\text{None}, \text{eigsiter}=\text{None}, \text{eigstol}=0, \text{tol}=1e-10, \text{monitorres}=\text{False}, \text{returninfo}=\text{False}, \text{show}=\text{False}, \text{threshkind}=\text{‘soft’}, \text{perc}=\text{None}, \text{callback}=\text{None})

Iterative Shrinkage-Thresholding Algorithm (ISTA).
Solve an optimization problem with $L_p$, $p = 0, 1/2, 1$ regularization, given the operator $\mathbf{Op}$ and data $\mathbf{y}$. The operator can be real or complex, and should ideally be either square $N = M$ or underdetermined $N < M$.

**Parameters**

- **Op** [pylops.LinearOperator] Operator to invert
- **data** [numpy.ndarray] Data
- **niter** [int] Number of iterations
- **eps** [float, optional] Sparsity damping
- **alpha** [float, optional] Step size ($\alpha \leq 1/\lambda_{max}(\mathbf{Op}^H \mathbf{Op})$) guarantees convergence. If None, the maximum eigenvalue is estimated and the optimal step size is chosen. If provided, the condition will not be checked internally.
- **eigsiter** [float, optional] Number of iterations for eigenvalue estimation if alpha=None
- **eigstol** [float, optional] Tolerance for eigenvalue estimation if alpha=None
- **tol** [float, optional] Tolerance. Stop iterations if difference between inverted model at subsequent iterations is smaller than tol
- **monitorres** [bool, optional] Monitor that residual is decreasing
- **returninfo** [bool, optional] Return info of CG solver
- **show** [bool, optional] Display iterations log
- **threshkind** [str, optional] Kind of thresholding ('hard', 'soft', 'half', 'hard-percentile', 'soft-percentile', or 'half-percentile' - 'soft' used as default)
- **perc** [float, optional] Percentile, as percentage of values to be kept by thresholding (to be provided when thresholding is soft-percentile or half-percentile)
- **callback** [callable, optional] Function with signature (callback(x)) to call after each iteration where $x$ is the current model vector

**Returns**

- **xinv** [numpy.ndarray] Inverted model
- **niter** [int] Number of effective iterations
- **cost** [numpy.ndarray, optional] History of cost function

**Raises**

- `NotImplementedError` If threshkind is different from hard, soft, half, soft-percentile, or half-percentile
- `ValueError` If perc=None when threshkind is soft-percentile or half-percentile
- `ValueError` If monitorres=True and residual increases

**See also:**

- **OMP** Orthogonal Matching Pursuit (OMP).
- **FISTA** Fast Iterative Shrinkage-Thresholding Algorithm (FISTA).
- **SPGL1** Spectral Projected-Gradient for L1 norm (SPGL1).
- **SplitBregman** Split Bregman for mixed L2-L1 norms.
Notes

Solves the following optimization problem for the operator $\text{Op}$ and the data $d$:

$$ J = \|d - \text{Op}x\|^2 + \epsilon \|x\|_p $$

using the Iterative Shrinkage-Thresholding Algorithms (ISTA) [1], where $p = 0, 1, 1/2$. This is a very simple iterative algorithm which applies the following step:

$$ x^{(i+1)} = T_{(\epsilon \alpha/2, p)}(x^{(i)} + \alpha \text{Op}^H(d - \text{Op}x^{(i)})) $$

where $\epsilon \alpha/2$ is the threshold and $T_{(\tau, p)}$ is the thresholding rule. The most common variant of ISTA uses the so-called soft-thresholding rule $T_{(\tau, p = 1)}$. Alternatively an hard-thresholding rule is used in the case of $p=0$ or a half-thresholding rule is used in the case of $p=1/2$. Finally, percentile bases thresholds are also implemented: the damping factor is not used anymore an the threshold changes at every iteration based on the computed percentile.

Examples using pylops.optimization.sparsityISTA

- sphx_glr_gallery_plot_ista.py
- 03. Solvers

pylops.optimization.sparsity.FISTA

```python
pylops.optimization.sparsity.FISTA(Op, data, niter, eps=0.1, alpha=None, eigsiter=None, eigstol=0, tol=1e-10, returninfo=False, show=False, threshkind='soft', perc=None, callback=None)
```

Fast Iterative Shrinkage-Thresholding Algorithm (FISTA).

Solve an optimization problem with $L1$ regularization function given the operator $\text{Op}$ and data $y$. The operator can be real or complex, and should ideally be either square $N = M$ or underdetermined $N < M$.

Parameters

- **Op** [pylops.LinearOperator] Operator to invert
- **data** [numpy.ndarray] Data
- **niter** [int] Number of iterations
- **eps** [float, optional] Sparsity damping
- **alpha** [float, optional] Step size ($\alpha \leq 1/\lambda_{max}(\text{Op}^H\text{Op})$ guarantees convergence. If None, the maximum eigenvalue is estimated and the optimal step size is chosen. If provided, the condition will not be checked internally).
- **eigsiter** [int, optional] Number of iterations for eigenvalue estimation if alpha=None
- **eigstol** [float, optional] Tolerance for eigenvalue estimation if alpha=None
- **tol** [float, optional] Tolerance. Stop iterations if difference between inverted model at subsequent iterations is smaller than tol
- **returninfo** [bool, optional] Return info of FISTA solver
- **show** [bool, optional] Display iterations log
perc [float, optional] Percentile, as percentage of values to be kept by thresholding (to be provided when thresholding is soft-percentile or half-percentile)

callback [callable, optional] Function with signature (callback(x)) to call after each iteration where x is the current model vector

Returns

xinv [numpy.ndarray] Inverted model
niter [int] Number of effective iterations
cost [numpy.ndarray, optional] History of cost function

Raises

NotImplementedError If threshkind is different from hard, soft, half, soft-percentile, or half-percentile

ValueError If perc=None when threshkind is soft-percentile or half-percentile

See also:

OMP Orthogonal Matching Pursuit (OMP).
ISTA Iterative Shrinkage-Thresholding Algorithm (ISTA).
SPGL1 Spectral Projected-Gradient for L1 norm (SPGL1).
SplitBregman Split Bregman for mixed L2-L1 norms.

Notes

Solves the following optimization problem for the operator Op and the data d:

\[ J = \|d - Op x\|_2^2 + \epsilon \|x\|_p \]

using the Fast Iterative Shrinkage-Thresholding Algorithm (FISTA) [1], where \( p = 0, 1, 1/2 \). This is a modified version of ISTA solver with improved convergence properties and limited additional computational cost. Similarly to the ISTA solver, the choice of the thresholding algorithm to apply at every iteration is based on the choice of \( p \).

Examples using pylops.optimization.sparsity.FISTA

• sphx_glr_gallery_plot_ista.py

• 03. Solvers
• 05. Image deblurring
• 11. Radon filtering
• 15. Least-squares migration

pylops.optimization.sparsity.SPGL1

pylops.optimization.sparsity.SPGL1(Op, data, SOp=None, tau=0, sigma=0, x0=None, **kwargs_spgl1)

Spectral Projected-Gradient for L1 norm.
Solve a constrained system of equations given the operator $Op$ and a sparsifying transform $SOp$ aiming to retrieve a model that is sparse in the sparsifying domain.

This is a simple wrapper to `spgl1.spgl1` which is a porting of the well-known SPGL1 MATLAB solver into Python. In order to be able to use this solver you need to have installed the `spgl1` library.

**Parameters**

- $Op$ [pylops.LinearOperator] Operator to invert
- $data$ [numpy.ndarray] Data
- $SOp$ [pylops.LinearOperator] Sparsifying transform
- $tau$ [float] Non-negative LASSO scalar. If different from 0, SPGL1 will solve LASSO problem
- $sigma$ [list] BPDN scalar. If different from 0, SPGL1 will solve BPDN problem
- $x0$ [numpy.ndarray] Initial guess
- **kwargs_spgl1 Arbitrary keyword arguments for `spgl1.spgl1` solver

**Returns**

- $xinv$ [numpy.ndarray] Inverted model in original domain.
- $pinv$ [numpy.ndarray] Inverted model in sparse domain.
- $info$ [dict] Dictionary with the following information:
  - $tau$, final value of tau (see sigma above)
  - $rnorm$, two-norm of the optimal residual
  - $rgap$, relative duality gap (an optimality measure)
  - $gnorm$, Lagrange multiplier of (LASSO)
  - $stat$, 1: found a BPDN solution, 2: found a BP solution; exit based on small gradient, 3: found a BP solution; exit based on small residual, 4: found a LASSO solution, 5: error, too many iterations, 6: error, linesearch failed, 7: error, found suboptimal BP solution, 8: error, too many matrix-vector products.
  - $niters$, number of iterations
  - $nProdA$, number of multiplications with $A$
  - $nProdAt$, number of multiplications with $A'$
  - $n_newton$, number of Newton steps
  - $time_project$, projection time (seconds)
  - $time_matprod$, matrix-vector multiplications time (seconds)
  - $time_total$, total solution time (seconds)
  - $niters_lsqr$, number of lsqr iterations (if subspace_min=True)
  - $xnorm1$, L1-norm model solution history through iterations
  - $rnorm2$, L2-norm residual history through iterations
  - $lambdaa$, Lagrange multiplier history through iterations

**Raises**

- `ModuleNotFoundError` If the `spgl1` library is not installed
Notes

Solve different variations of sparsity-promoting inverse problem by imposing sparsity in the retrieved model [1].

The first problem is called *basis pursuit denoise* (BPDN) and its cost function is

\[ \|x\|_1 \quad \text{subj.to} \quad \|\text{Op}_S H x - b\|_2^2 \leq \sigma, \]

while the second problem is the *l1-regularized least-squares* or *LASSO* problem and its cost function is

\[ \|\text{Op}_S H x - b\|_2^2 \quad \text{subj.to} \quad \|x\|_1 \leq \tau \]

Examples using `pylops.optimization.sparsity.SPGL1`

- 03. Solvers

`pylops.optimization.sparsity.SplitBregman`

`pylops.optimization.sparsity.SplitBregman(Op, RegsL1, data, niter_outer=3, niter_inner=5, RegsL2=None, dataregsL2=None, mu=1.0, epsRL1s=None, epsRL2s=None, tol=1e-10, tau=1.0, x0=None, restart=False, show=False, **kwargs_lsqr)`

Split Bregman for mixed L2-L1 norms.

Solve an unconstrained system of equations with mixed L2-L1 regularization terms given the operator `Op`, a list of L1 regularization terms `RegsL1`, and an optional list of L2 regularization terms `RegsL2`.

Parameters

- **Op** *(pylops.LinearOperator)* Operator to invert
- **RegsL1** *(list)* L1 regularization operators
- **data** *(numpy.ndarray)* Data
- **niter_outer** *(int)* Number of iterations of outer loop
- **niter_inner** *(int)* Number of iterations of inner loop of first step of the Split Bregman algorithm. A small number of iterations is generally sufficient and for many applications optimal efficiency is obtained when only one iteration is performed.
- **RegsL2** *(list)* Additional L2 regularization operators (if `None`, L2 regularization is not added to the problem)
- **dataregsL2** *(list, optional)* L2 Regularization data (must have the same number of elements of `RegsL2` or equal to `None` to use a zero data for every regularization operator in `RegsL2`)
- **mu** *(float, optional)* Data term damping
- **epsRL1s** *(list)* L1 Regularization dampings (must have the same number of elements as `RegsL1`)
- **epsRL2s** *(list)* L2 Regularization dampings (must have the same number of elements as `RegsL2`)
tol [float, optional] Tolerance. Stop outer iterations if difference between inverted model at subsequent iterations is smaller than tol

tau [float, optional] Scaling factor in the Bregman update (must be close to 1)

x0 [numpy.ndarray, optional] Initial guess

restart [bool, optional] The unconstrained inverse problem in inner loop is initialized with the initial guess (True) or with the last estimate (False)

show [bool, optional] Display iterations log

**kwargs_lsqr Arbitrary keyword arguments for scipy.sparse.linalg.lsqr solver used to solve the first subproblem in the first step of the Split Bregman algorithm.

Returns

xinv [numpy.ndarray] Inverted model

itn_out [int] Iteration number of outer loop upon termination

Notes

Solve the following system of unconstrained, regularized equations given the operator Op and a set of mixed norm (L2 and L1) regularization terms RL2,i and RL1,i, respectively:

\[ J = \mu / 2 ||d - Opx||_2^2 + \sum_i \epsilon_{RL2,i} / 2 ||d_{RL2,i} - R_{L2,i}x||_2^2 + \sum_i ||R_{L1,i}x||_1 \]

where \( \mu \) and \( \epsilon_{RL2,i} \) are the damping factors used to weight the different L2 regularization terms of the cost function.

The generalized Split Bergman algorithm is used to solve such cost function: the algorithm is composed of a sequence of unconstrained inverse problems and Bregman updates.

The original system of equations is initially converted into a constrained problem:

\[ J = \mu / 2 ||d - Opx||_2^2 + \sum_i \epsilon_{RL2,i} / 2 ||d_{RL2,i} - R_{L2,i}x||_2^2 + \sum_i ||y_i||_1 \quad s.t. \quad y_i = R_{L1,i}x \quad \forall i \]

and solved as follows:

\[ (x^{k+1},y^{k+1}) = \arg \min_{x,y} ||d - Opx||_2^2 + \sum_i \epsilon_{RL2,i} / 2 ||d_{RL2,i} - R_{L2,i}x||_2^2 + \sum_i ||y_i||_1 \sum_i \epsilon_{RL1,i} / 2 ||y_i - R_{L1,i}x - b^{k+1}_i||_2^2 \]

\[ b^{k+1}_i = b^k_i + (R_{L1,i}x^{k+1} - y^{k+1}) \]

The scipy.sparse.linalg.lsqr solver and a fast shrinkage algorithm are used within a inner loop to solve the first step. The entire procedure is repeated niter_outer times until convergence.

Examples using pylops.optimization.sparsity.SplitBregman

- sphx_glr_gallery_plot_tvreg.py
- 05. Image deblurring
- 16. CT Scan Imaging
3.6.3 Applications

Wave-Equation processing

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SeismicInterpolation(data, nrec, iava[, ...])</code></td>
<td>Seismic interpolation (or regularization).</td>
</tr>
<tr>
<td><code>Deghosting(p, nt, nr, dt, dr, vel, zrec[, ...])</code></td>
<td>Wavefield dehosting.</td>
</tr>
<tr>
<td><code>WavefieldDecomposition(p, vz, nt, nr, dt, ...)</code></td>
<td>Up-down wavefield decomposition.</td>
</tr>
<tr>
<td><code>MDD(G, d[, dt, dr, nfmax, wav, twosided, ...])</code></td>
<td>Multi-dimensional deconvolution.</td>
</tr>
<tr>
<td><code>Marchenko(R[, R1, dt, nt, dr, nfmax, wav, ...])</code></td>
<td>Marchenko redatuming</td>
</tr>
<tr>
<td><code>LSM(z, x, t, srcs, recs, vel, wav, wavecenter)</code></td>
<td>Least-squares Migration (LSM).</td>
</tr>
</tbody>
</table>

SeismicInterpolation

Seismic interpolation (or regularization).
Interpolate seismic data from irregular to regular spatial grid. Depending on the size of the input `data`, interpolation is either 2- or 3-dimensional. In case of 3-dimensional interpolation, data can be irregularly sampled in either one or both spatial directions.

**Parameters**

- `data` [np.ndarray] Irregularly sampled seismic data of size $[n_r \times n_r \times n_t]$
- `nrec` [int or tuple] Number of elements in the regularly sampled (reconstructed) spatial array, $n_{R_r}$ for 2-dimensional data and $(n_{R_r}, n_{R_r})$ for 3-dimensional data
- `iava` [list or numpy.ndarray] Integer (or floating) indices of locations of available samples in first dimension of regularly sampled spatial grid of interpolated signal. The `pylops.basicoperators.Restriction` operator is used in case of integer indices, while the `pylops.signalprocessing.Iterp` operator is used in case of floating indices.
- `iava1` [list or numpy.ndarray, optional] Integer (or floating) indices of locations of available samples in second dimension of regularly sampled spatial grid of interpolated signal. Can be used only in case of 3-dimensional data.
- `kind` [str, optional] Type of inversion: `fk` (default), `spatial`, `radon-linear`, `radon-parabolic` or `radon-hyperbolic` and `sliding`
- `nffts` [int or tuple, optional] nffts : tuple, optional Number of samples in Fourier Transform for each direction. Required if `kind='fk'`
- `sampling` [tuple, optional] Sampling steps dy, dx and dt. Required if `kind='fk'` or `kind='radon-lin'`
- `spataxis` [np.ndarray, optional] First spatial axis. Required for `kind='radon-lin'`, can also be provided instead of `sampling` for `kind='fk'`
- `spat1axis` [np.ndarray, optional] Second spatial axis. Required for `kind='radon-lin'`, can also be provided instead of `sampling` for `kind='fk'`
**timeaxis** [np.ndarray, optional] Time axis. Required for kind='radon-lin', can also be provided instead of sampling for kind='fk'

**paxis** [np.ndarray, optional] First Radon axis. Required for kind='radon-linear', kind='radon-parabolic', kind='radon-hyperbolic' and kind='sliding'

**p1axis** [np.ndarray, optional] Second Radon axis. Required for kind='radon-linear', kind='radon-parabolic' and kind='radon-hyperbolic' and kind='sliding'

**centeredh** [bool, optional] Assume centered spatial axis (True) or not (False). Required for kind='radon-linear', kind='radon-parabolic' and kind='radon-hyperbolic'

**nwins** [int or tuple, optional] Number of windows. Required for kind='sliding'

**nwin** [int or tuple, optional] Number of samples of window. Required for kind='sliding'

**nover** [int or tuple, optional] Number of samples of overlapping part of window. Required for kind='sliding'

**design** [bool, optional] Print number of sliding window (True) or not (False) when using kind='sliding'

**engine** [str, optional] Engine used for Radon computations (numpy or numba)

**dottest** [bool, optional] Apply dot-test

**kwargs_solver** Arbitrary keyword arguments for pylops.optimization.leastsquares.RegularizedInversion solver if kind='spatial' or pylops.optimization.sparsity.FISTA solver otherwise

**Returns**

**reCDATA** [np.ndarray] Reconstructed data of size \([n_{R_y} \times n_{R_x} \times n_t])

**reCDATA** [np.ndarray] Reconstructed data in the sparse or preconditioned domain in case of kind='fk', kind='radon-linear', kind='radon-parabolic', kind='radon-hyperbolic' and kind='sliding'

**cost** [np.ndarray] Cost function norm

**Raises**

**KeyError** If kind is neither spatial, fl, radon-linear, radon-parabolic, radon-hyperbolic nor sliding

**Notes**

The problem of seismic data interpolation (or regularization) can be formally written as

\[ y = Rx \]

where a restriction or interpolation operator is applied along the spatial direction(s). Here \( y = [y_{R_1}, y_{R_2}, ..., y_{R_N}]^T \) where each vector \( y_{R_i} \) contains all time samples recorded in the seismic data at the specific receiver \( R_i \). Similarly, \( x = [x_{r_1}, x_{r_2}, ..., x_{r_M}]^T \), contains all traces at the regularly and finely sampled receiver locations \( r_i \).

Several alternative approaches can be taken to solve such a problem. They mostly differ in the choice of the regularization (or preconditioning) used to mitigate the ill-posedness of the problem:
• **spatial**: least-squares inversion in the original time-space domain with an additional spatial smoothing regularization term, corresponding to the cost function 

\[ J = \|y - Rx\|_2^2 + \varepsilon \|\nabla x\|_2^2 \]

where \( \nabla \) is a second order space derivative implemented via `pylops.basicoperators.SecondDerivative` in 2-dimensional case and `pylops.basicoperators.Laplacian` in 3-dimensional case.

• **fk**: L1 inversion in frequency-wavenumber preconditioned domain corresponding to the cost function 

\[ J = \|y - RFx\|_2^2 \]

where \( F \) is frequency-wavenumber transform implemented via `pylops.signalprocessing.FFT2D` in 2-dimensional case and `pylops.signalprocessing.FFTND` in 3-dimensional case.

• **radon-linear**: L1 inversion in linear Radon preconditioned domain using the same cost function as **fk** but with \( F \) being a Radon transform implemented via `pylops.signalprocessing.Radon2D` in 2-dimensional case and `pylops.signalprocessing.Radon3D` in 3-dimensional case.

• **radon-parabolic**: L1 inversion in parabolic Radon preconditioned domain.

• **radon-hyperbolic**: L1 inversion in hyperbolic Radon preconditioned domain.

• **sliding**: L1 inversion in sliding-linear Radon preconditioned domain using the same cost function as **fk** but with \( F \) being a sliding Radon transform implemented via `pylops.signalprocessing.Sliding2D` in 2-dimensional case and `pylops.signalprocessing.Sliding3D` in 3-dimensional case.

Examples using `pylops.waveeqprocessing.SeismicInterpolation`

• **12. Seismic regularization**

`pylops.waveeqprocessing.Deghosting` 

`pylops.waveeqprocessing.Deghosting(p, nt, nr, dt, dr, vel, zrec, pd=None, win=None, npad=(11, 11), ntaper=(11, 11), restriction=None, sp-transf=None, solver=<function lsqr>, dottest=False, dtype='complex128', **kwargs_solver)`

Wavefield deghosting.

Apply seismic wavefield decomposition from single-component (pressure) data. This process is also generally referred to as model-based deghosting.

**Parameters**

- `p` [np.ndarray] Pressure data of of size \([n_{rz} \times n_{rx}] \times n_t\) (or \([n_{rz,sub} \times n_{rx,sub}] \times n_t\) in case a restriction operator is provided. Note that \( n_{rz,sub} \) (and \( n_{rx,sub} \)) must agree with the size of the output of this operator)

- `nt` [int] Number of samples along the time axis

- `nr` [int or tuple] Number of samples along the receiver axis (or axes)

- `dt` [float] Sampling along the time axis

- `dr` [float or tuple] Sampling along the receiver array of the separated pressure consituents

- `vel` [float] Velocity along the receiver array (must be constant)

- `zrec` [float] Depth of receiver array

- `pd` [np.ndarray, optional] Direct arrival to be subtracted from `p`

- `pd` [np.ndarray, optional] Time window to be applied to `p` to remove the direct arrival (if `pd=None`)
**ntaper** [float or tuple, optional] Number of samples of taper applied to propagator to avoid edge effects

**npad** [float or tuple, optional] Number of samples of padding applied to propagator to avoid edge effects

**restriction** [pylops.LinearOperator, optional] Restriction operator

**sptransf** [pylops.LinearOperator, optional] Sparsifying operator

**solver** [float, optional] Function handle of solver to be used if kind='inverse'

**dottest** [bool, optional] Apply dot-test

**dtype** [str, optional] Type of elements in input array.

**kwargs_solver** Arbitrary keyword arguments for chosen solver

**Returns**

- **pup** [np.ndarray] Up-going wavefield
- **pdown** [np.ndarray] Down-going wavefield

**Notes**

Up- and down-going components of seismic data \( p^- (x, t) \) and \( p^+ (x, t) \) can be estimated from single-component data \( p(x, t) \) using a ghost model.

The basic idea is that of using a one-way propagator in the f-k domain (also referred to as ghost model) to predict the down-going field from the up-going one (excluded the direct arrival and its source ghost referred here to as \( p_d(x, t) \)):

\[
p^+ - p_d = e^{-j k_z z_{rec}} p^-
\]

where \( k_z \) is the vertical wavenumber and \( z_{rec} \) is the depth of the array of receivers.

In a matrix form we can thus write the total wavefield as:

\[
p - p_d = (I + \Phi)p^-
\]

where \( \Phi \) is one-way propagator implemented via the `pylops.waveeqprocessing.PhaseShift` operator.

**Examples using pylops.waveeqprocessing.Deghosting**

- [3. Deghosting](#)

**pylops.waveeqprocessing.WavefieldDecomposition**

**pylops.waveeqprocessing.WavefieldDecomposition**(\( p, vz, nt, nr, dt, dr, rho, vel, nffts=(\text{None, None, None}), \text{critical}=100.0, \text{ntaper}=10, \text{scaling}=1.0, \text{kind}='inverse', \text{restriction}=\text{None}, \text{sptransf}=\text{None}, \text{solver}=<\text{function lsqr}>, \text{dottest}=\text{False}, \text{dtype}=\text{complex128}', **\text{kwargs_solver})

Up-down wavefield decomposition.

Apply seismic wavefield decomposition from multi-component (pressure and vertical particle velocity) data. This process is also generally referred to as data-based deghosting.
Parameters

- **p** [np.ndarray] Pressure data of size \([n_{rx} \times n_{ry} \times n_t]\) (or \([n_{rx,sub} \times n_{ry,sub} \times n_t]\) in case a restriction operator is provided. Note that \(n_{rx,sub}\) (and \(n_{ry,sub}\)) must agree with the size of the output of this operator)

- **vz** [np.ndarray] Vertical particle velocity data of same size as pressure data

- **nt** [int] Number of samples along the time axis

- **nr** [int or tuple] Number of samples along the receiver axis (or axes)

- **dt** [float] Sampling along the time axis

- **dr** [float or tuple] Sampling along the receiver array (or axes)

- **rho** [float] Density along the receiver array (must be constant)

- **vel** [float] Velocity along the receiver array (must be constant)

- **nffts** [tuple, optional] Number of samples along the wavenumber and frequency axes

- **critical** [float, optional] Percentage of angles to retain in obliquity factor. For example, if \(\text{critical}=100\) only angles below the critical angle \(\frac{f_{\text{ke}}}{c}\) will be retained

- **ntaper** [float, optional] Number of samples of taper applied to obliquity factor around critical angle

- **kind** [str, optional] Type of separation: inverse (default) or analytical

- **scaling** [float, optional] Scaling to apply to the operator (see Notes of `pylops.waveeqprocessing.wavedecomposition.UpDownComposition2D` for more details)

- **restriction** [pylops.LinearOperator, optional] Restriction operator

- **sptransf** [pylops.LinearOperator, optional] Sparsifying operator

- **solver** [float, optional] Function handle of solver to be used if \(\text{kind}='\text{inverse}'\)

- **dottest** [bool, optional] Apply dot-test

- **dtype** [str, optional] Type of elements in input array.

**kwargs_solver Arbitrary keyword arguments for chosen solver

Returns

- **pup** [np.ndarray] Up-going wavefield

- **pdown** [np.ndarray] Down-going wavefield

Raises

- **KeyError** If \(\text{kind}\) is neither analytical nor inverse

Notes

Up- and down-going components of seismic data \((p^- (x,t))\) and \((p^+ (x,t))\) can be estimated from multi-component data \((p(x,t))\) and \((v_z(x,t))\) by computing the following expression [1]:

\[
\begin{bmatrix}
  p^+(k_x,\omega) \\
  p^-(k_x,\omega)
\end{bmatrix}
= \frac{1}{2}
\begin{bmatrix}
  1 & \frac{\omega^2}{k_x^2} \\
  1 & -\frac{\omega^2}{k_x^2}
\end{bmatrix}
\begin{bmatrix}
  p(k_x,\omega) \\
  v_z(k_x,\omega)
\end{bmatrix}
\]

if \(\text{kind}='\text{analytical}'\) or alternatively by solving the equation in `ptcpy.waveeqprocessing.UpDownComposition2D` as an inverse problem, if \(\text{kind}='\text{inverse}'\).
The latter approach has several advantages as data regularization can be included as part of the separation process allowing the input data to be aliased. This is obtained by solving the following problem:

\[
\begin{bmatrix}
P \\ s \ast v_z
\end{bmatrix} = \begin{bmatrix}
RF & 0 \\ 0 & s \ast RF
\end{bmatrix} W \begin{bmatrix}
F^H S & 0 \\ 0 & F^H S
\end{bmatrix} P^\pm
\]

where \(R\) is a ptycho.basicoperators.Restriction operator and \(S\) is sparsifying transform operator (e.g., ptycho.signalprocessing.Radon2D).

**Examples using `pylops.waveeqprocessing.WavefieldDecomposition`**

- **14. Seismic wavefield decomposition**

`pylops.waveeqprocessing.MDD`

`pylops.waveeqprocessing.MDD(G, d, dt=0.004, dr=1.0, nfmax=None, wav=None, twosided=True, causality_precond=False, adjoint=False, psf=False, dtype='float64', dottest=False, saveGt=True, add_negative=True, smooth_precond=0, **kwargs_lsqr)`

Multi-dimensional deconvolution.

Solve multi-dimensional deconvolution problem using `scipy.sparse.linalg.lsqr` iterative solver.

**Parameters**

- **G** [numpy.ndarray] Multi-dimensional convolution kernel in time domain of size \([n_s \times n_r \times n_t]\) for \(\text{twosided}=False\) or \(\text{twosided}=True\) and \(\text{add_negative}=True\) (with only positive times) or size \([n_s \times n_r \times 2 \times n_t - 1]\) for \(\text{twosided}=True\) and \(\text{add_negative}=False\) (with both positive and negative times)

- **d** [numpy.ndarray] Data in time domain \([n_s(n_v \times n_t) \times n_t]\) if \(\text{twosided}=False\) or \(\text{twosided}=True\) and \(\text{add_negative}=True\) (with only positive times) or size \([n_s(n_v \times n_t) \times 2 \times n_t - 1]\) if \(\text{twosided}=True\)

- **dt** [float, optional] Sampling of time integration axis

- **dr** [float, optional] Sampling of receiver integration axis

- **nfmax** [int, optional] Index of max frequency to include in deconvolution process

- **wav** [numpy.ndarray, optional] Wavelet to convolve to the inverted model and psf (must be centered around its index in the middle of the array). If None, the outputs of the inversion are returned directly.

- **twosided** [bool, optional] MDC operator and data both negative and positive time (True) or only positive (False)

- **add_negative** [bool, optional] Add negative side to MDC operator and data (True) or not (False)- operator and data are already provided with both positive and negative sides. To be used only with \(\text{twosided}=True\).

- **causality_precond** [bool, optional] Apply causality mask (True) or not (False)

- **adjoint** [bool, optional] Compute and return adjoint(s)

- **psf** [bool, optional] Compute and return Point Spread Function (PSF) and its inverse

- **dtype** [bool, optional] Type of elements in input array.

- **dottest** [bool, optional] Apply dot-test
saveGt [bool, optional] Save G and G^H to speed up the computation of adjoint of `pylops.signalprocessing.Fredholm1` (True) or create G^H on-the-fly (False) Note that saveGt=True will be faster but double the amount of required memory

**kwargs_lsqr Arbitrary keyword arguments for scipy.sparse.linalg.lsqr solver

Returns

minv [numpy.ndarray] Inverted model of size \([n_r \times n_v \times n_t]\) for twosided=False or \([n_r \times n_v \times 2 \times n_t - 1]\) for twosided=True

madj [numpy.ndarray] Adjoint model of size \([n_r \times n_v \times n_t]\) for twosided=False or \([n_r \times n_r \times 2 \times n_t - 1]\) for twosided=True

psfinv [numpy.ndarray] Inverted psf of size \([n_r \times n_r \times n_t]\) for twosided=False or \([n_r \times n_r \times 2 \times n_t - 1]\) for twosided=True

psfadj [numpy.ndarray] Adjoint psf of size \([n_r \times n_r \times n_t]\) for twosided=False or \([n_r \times n_r \times 2 \times n_t - 1]\) for twosided=True

See also:

MDC Multi-dimensional convolution

Notes

Multi-dimensional deconvolution (MDD) is a mathematical ill-solved problem, well-known in the image processing and geophysical community [1].

MDD aims at removing the effects of a Multi-dimensional Convolution (MDC) kernel or the so-called blurring operator or point-spread function (PSF) from a given data. It can be written as

\[
d = Dm
\]

or, equivalently, by means of its normal equation

\[
m = (D^H D)^{-1} D^H d
\]

where \(D^H D\) is the PSF.

Examples using `pylops.waveeqprocessing.MDD`

- 09. Multi-Dimensional Deconvolution

`pylops.waveeqprocessing.Marchenko`

class `pylops.waveeqprocessing.Marchenko` (R, R1=None, dt=0.004, nt=None, dr=1.0, nfmax=None, wav=None, toff=0.0, nsmooth=10, dtype=’float64’, saveRt=True, prescaled=False)

Marchenko redatuming

Solve multi-dimensional Marchenko redatuming problem using scipy.sparse.linalg.lsqr iterative solver.

Parameters
**R** [numpy.ndarray] Multi-dimensional reflection response in time or frequency domain of size \([n_s \times n_r \times n_t / n_{f_{\text{max}}}]\). If provided in time, \(R\) should not be of complex type. Note that the reflection response should have already been multiplied by 2.

\(R1\) [bool, optional] *Deprecated*, will be removed in v2.0.0. Simply kept for back-compatibility with previous implementation

\(dt\) [float, optional] Sampling of time integration axis

\(nt\) [float, optional] Number of samples in time (not required if \(R\) is in time)

\(dr\) [float, optional] Sampling of receiver integration axis

\(nfmax\) [int, optional] Index of max frequency to include in deconvolution process

\(wav\) [numpy.ndarray, optional] Wavelet to apply to direct arrival when created using \texttt{trav}

\(toff\) [float, optional] Time-offset to apply to traveltime

\(ns\_smooth\) [int, optional] Number of samples of smoothing operator to apply to window

\(dtype\) [bool, optional] Type of elements in input array.

\(saveRt\) [bool, optional] Save \(R\) and \(R^\dagger\) to speed up the computation of adjoint of \texttt{pylops.signalprocessing.Fredholm1(True)} or create \(R^\dagger\) on-the-fly (False) Note that \(saveRt=True\) will be faster but double the amount of required memory

\(prescaled\) [bool, optional] Apply scaling to \(R\) (False) or not (False) when performing spatial and temporal summations within the \texttt{pylops.waveeqprocessing.MDC} operator. In case \(prescaled=True\), the \(R\) is assumed to have been pre-scaled by the user.

**Raises**

*TypeError* If \(t\) is not \texttt{numpy.ndarray}.

**See also:**

\texttt{MDC} Multi-dimensional convolution

\texttt{MDD} Multi-dimensional deconvolution

**Notes**

Marchenko redatuming is a method that allows to produce correct subsurface-to-surface responses given the availability of a reflection data and a macro-velocity model [1].

The Marchenko equations can be written in a compact matrix form [2] and solved by means of iterative solvers such as LSQR:

\[
\begin{bmatrix}
\Theta R f_d^+ \\
0 & \Theta R^* & 0 \\
\end{bmatrix}
= I - \begin{bmatrix}
0 & \Theta R \\
\Theta^* & 0 \\
\end{bmatrix}
\begin{bmatrix}
f^- \\
f_m^+ \\
\end{bmatrix}
\]

Finally the subsurface Green’s functions can be obtained applying the following operator to the retrieved focusing functions

\[
\begin{bmatrix}
-g^- \\
g^+ \\
\end{bmatrix}
= I - \begin{bmatrix}
0 & R \\
R^* & 0 \\
\end{bmatrix}
\begin{bmatrix}
f^- \\
f^+ \\
\end{bmatrix}
\]

Here \(R\) is the monopole-to-particle velocity seismic response (already multiplied by 2).

**Attributes**

\(ns\) [int] Number of samples along source axis
nr [int] Number of samples along receiver axis
shape [tuple] Operator shape
explicit [bool] Operator contains a matrix that can be solved explicitly (True) or not (False)

Methods

__init__ (R, R1, dt, nt, dr, nfmax, wav, ...) Initialize self.
apply_multiplepoints (trav, G0, nfft, rtm, ...) Marchenko redatuming for multiple points
apply_onepoint (trav, G0=None, nfft=None, rtm=False, greens=False, dottest=False, fast=None, **kwargs_lsqr) Marchenko redatuming for one point

apply_onepoint (trav, G0=None, nfft=None, rtm=False, greens=False, dottest=False, fast=None, **kwargs_lsqr)

Marchenko redatuming for one point
Solve the Marchenko redatuming inverse problem for a single point given its direct arrival traveltime curve (trav) and waveform (G0).

Parameters

  trav [numpy.ndarray] Traveltime of first arrival from subsurface point to surface receivers of size [nr x 1]
  G0 [numpy.ndarray, optional] Direct arrival in time domain of size [nr x nt] (if None, create arrival using trav)
  nfft [int, optional] Number of samples in fft when creating the analytical direct wave
  rtm [bool, optional] Compute and return rtm redatuming
  greens [bool, optional] Compute and return Green’s functions
  dottest [bool, optional] Apply dot-test
  fast [bool] Deprecated, will be removed in v2.0.0
  **kwargs_lsqr Arbitrary keyword arguments for scipy.sparse.linalg.lsqr solver

Returns

  f1_inv_minus [numpy.ndarray] Inverted upgoing focusing function of size [nr x nt]
  f1_inv_plus [numpy.ndarray] Inverted downgoing focusing function of size [nr x nt]
  p0_minus [numpy.ndarray] Single-scattering standard redatuming upgoing Green’s function of size [nr x nt]
  g_inv_minus [numpy.ndarray] Inverted upgoing Green’s function of size [nr x nt]
  g_inv_plus [numpy.ndarray] Inverted downgoing Green’s function of size [nr x nt]

apply_multiplepoints (trav, G0=None, nfft=None, rtm=False, greens=False, dottest=False, **kwargs_lsqr)

Marchenko redatuming for multiple points
Solve the Marchenko redatuming inverse problem for multiple points given their direct arrival traveltime curves (trav) and waveforms (G0).

Parameters

apply_multiplepoints (trav, G0=None, nfft=None, rtm=False, greens=False, dottest=False, **kwargs_lsqr)
**trav** [```numpy.ndarray```] Traveltime of first arrival from subsurface points to surface receivers of size \([n_r \times n_{vs}]\)

**G0** [```numpy.ndarray```, optional] Direct arrival in time domain of size \([n_r \times n_{vs} \times n_t]\) (if None, create arrival using trav)

**nfft** [```int```, optional] Number of samples in fft when creating the analytical direct wave

**rtm** [```bool```, optional] Compute and return rtm redatuming

**greens** [```bool```, optional] Compute and return Green’s functions

**dottest** [```bool```, optional] Apply dot-test

**kwargs_lsqr** Arbitrary keyword arguments for ```scipy.sparse.linalg.lsqr``` solver

**Returns**

**f1_inv_minus** [```numpy.ndarray```] Inverted upgoing focusing function of size \([n_r \times n_{vs} \times n_t]\)

**f1_inv_plus** [```numpy.ndarray```] Inverted downgoing focusing function of size \([n_r \times n_{vs} \times n_t]\)

**p0_minus** [```numpy.ndarray```] Single-scattering standard redatuming upgoing Green’s function of size \([n_r \times n_{vs} \times n_t]\)

**g_inv_minus** [```numpy.ndarray```] Inverted upgoing Green’s function of size \([n_r \times n_{vs} \times n_t]\)

**g_inv_plus** [```numpy.ndarray```] Inverted downgoing Green’s function of size \([n_r \times n_{vs} \times n_t]\)

**Examples using pylops.waveeqprocessing.Marchenko**

- 10. Marchenko redatuming by inversion

**pylops.waveeqprocessing.LSM**

**class** ```pylops.waveeqprocessing.LSM```(```z```, ```x```, ```t```, ```srcs```, ```recs```, ```vel```, ```wav```, ```wavcenter```, ```y=````None```, ```mode='eikonal'```, ```dottest=False```

Least-squares Migration (LSM).

Solve seismic migration as inverse problem given smooth velocity model vel and an acquisition setup identified by sources (src) and receivers (recs)

**Parameters**

**z** [```numpy.ndarray```] Depth axis

**x** [```numpy.ndarray```] Spatial axis

**t** [```numpy.ndarray```] Time axis for data

**srcs** [```numpy.ndarray```] Sources in array of size \([2/3 \times n_s]\)

**recs** [```numpy.ndarray```] Receivers in array of size \([2/3 \times n_r]\)

**vel** [```numpy.ndarray``` or ```float```] Velocity model of size \([(n_y \times) n_x \times n_z]\) (or constant)

**wav** [```numpy.ndarray```] Wavelet

**wavcenter** [```int```] Index of wavelet center
y [numpy.ndarray] Additional spatial axis (for 3-dimensional problems)

mode [numpy.ndarray, optional] Computation mode (eikonal, analytic - only for constant velocity)

dottest [bool, optional] Apply dot-test

See also:

pylops.waveeqprocessing.Demigration Demigration operator

Notes

Inverting a demigration operator is generally referred in the literature as least-squares migration (LSM) as historically a least-squares cost function has been used for this purpose. In practice any other cost function could be used, for examples if solver='pylops.optimization.sparsity.FISTA' a sparse representation of reflectivity is produced as result of the inversion.

Finally, it is worth noting that in the first iteration of an iterative scheme aimed at inverting the demigration operator, a projection of the recorded data in the model domain is performed and an approximate (band-limited) image of the subsurface is created. This process is referred to in the literature as migration.

Attributes

Demop [pylops.LinearOperator] Demigration operator

Methods

__init__ (z, x, t, srcs, recs, vel, wav, ...) Initialize self.

solve (d[, solver]) Solve least-squares migration equations with chosen solver

Parameters

d [numpy.ndarray] Input data of size \([n_s \times n_r \times n_t]\)

solver [func, optional] Solver to be used for inversion

**kwargs_solver Arbitrary keyword arguments for chosen solver

Returns

minv [np.ndarray] Inverted reflectivity model of size \([(n_y \times n_x \times n_z)]

Examples using pylops.waveeqprocessing.LSM

- 15. Least-squares migration

Geophysical subsurface characterization
PyLops

poststack.PoststackInversion(data, wav[, ...])

Post-stack linearized seismic inversion.

prestack.PrestackInversion(data, theta, wav)

Pre-stack linearized seismic inversion.

pylops.avopoststack.PoststackInversion

pylops.avopoststack.PoststackInversion(data, wav, m0=None, explicit=False, simultaneous=False, epsI=None, epsR=None, dottest=False, epsRL1=None, **kwargs_solver)

Post-stack linearized seismic inversion.

Invert post-stack seismic operator to retrieve an elastic parameter of choice from band-limited seismic post-stack data. Depending on the choice of input parameters, inversion can be trace-by-trace with explicit operator or global with either explicit or linear operator.

Parameters

data [np.ndarray] Band-limited seismic post-stack data of size \([n_{t0} \times n_x \times n_y]\)]

wav [np.ndarray] Wavelet in time domain (must have odd number of elements and centered to zero). If 1d, assume stationary wavelet for the entire time axis. If 2d of size \([n_{t0} \times n_h]\] use as non-stationary wavelet

m0 [np.ndarray, optional] Background model of size \([n_{t0} \times n_x \times n_y]\)]

explicit [bool, optional] Create a chained linear operator (False, preferred for large data) or a MatrixMult linear operator with dense matrix (True, preferred for small data)

simultaneous [bool, optional] Simultaneously invert entire data (True) or invert trace-by-trace (False) when using explicit operator (note that the entire data is always inverted when working with linear operator)

epsI [float, optional] Damping factor for Tikhonov regularization term

epsR [float, optional] Damping factor for additional Laplacian regularization term

dottest [bool, optional] Apply dot-test

epsRL1 [float, optional] Damping factor for additional blockiness regularization term

**kwargs_solver Arbitrary keyword arguments for scipy.linalg.lstsq solver (if explicit=True and epsR=None) or scipy.sparse.linalg.lsqr solver (if explicit=False and/or epsR is not None)

Returns

minv [np.ndarray] Inverted model of size \([n_{t0} \times n_x \times n_y]\)]

datar [np.ndarray] Residual data (i.e., data - background data) of size \([n_{t0} \times n_x \times n_y]\)]

Notes

The cost function and solver used in the seismic post-stack inversion module depends on the choice of explicit, simultaneous, epsI, and epsR parameters:

- explicit=True, epsI=None and epsR=None: the explicit solver scipy.linalg.lstsq is used if simultaneous=False (or the iterative solver scipy.sparse.linalg.lsqr is used if simultaneous=True)
• explicit=True with epsI and epsR=None: the regularized normal equations \( W^T d = (W^T W + \epsilon^2 I) A I \) are instead fed into the `scipy.linalg.lstsq` solver if simultaneous=False (or the iterative solver `scipy.sparse.linalg.lsqr` if simultaneous=True)

• explicit=False and epsR=None: the iterative solver `scipy.sparse.linalg.lsqr` is used

• explicit=False with epsR and epsRL1=None: the iterative solver `pylops.optimization.leastsquares.RegularizedInversion` is used to solve the spatially regularized problem.

• explicit=False with epsR and epsRL1: the iterative solver `pylops.optimization.sparsity.SplitBregman` is used to solve the blockiness-promoting (in vertical direction) and spatially regularized (in additional horizontal directions) problem.

Note that the convergence of iterative solvers such as `scipy.sparse.linalg.lsqr` can be very slow for this type of operator. It is suggested to take a two steps approach with first a trace-by-trace inversion using the explicit operator, followed by a regularized global inversion using the outcome of the previous inversion as initial guess.

### Examples using `pylops.avo.poststack.PoststackInversion`

• 07. Post-stack inversion

`pylops.avo.prestack.PrestackInversion`

`pylops.avo.prestack.PrestackInversion` (data, theta, wav, m0=None, linearization='akirich', explicit=False, simultaneous=False, epsI=None, epsR=None, dottest=False, returnres=False, epsRL1=None, **kwargs_solver)

Pre-stack linearized seismic inversion.

Invert pre-stack seismic operator to retrieve a set of elastic property profiles from band-limited seismic pre-stack data (i.e., angle gathers). Depending on the choice of input parameters, inversion can be trace-by-trace with explicit operator or global with either explicit or linear operator.

**Parameters**

- data [np.ndarray] Band-limited seismic post-stack data of size \( n_{t0} \times n_{\theta} (\times n_x \times n_y) \)
- theta [np.ndarray] Incident angles in degrees
- wav [np.ndarray] Wavelet in time domain (must had odd number of elements and centered to zero)
- m0 [np.ndarray, optional] Background model of size \( n_{t0} \times n_{m} (\times n_x \times n_y) \)
- linearization [str, optional] choice of linearization, akirich: Aki-Richards, fatti: Fatti (required only when m0 is None)
- explicit [bool, optional] Create a chained linear operator (False, preferred for large data) or a `MatrixMult` linear operator with dense matrix (True, preferred for small data)
- simultaneous [bool, optional] Simultaneously invert entire data (True) or invert trace-by-trace (False) when using explicit operator (note that the entire data is always inverted when working with linear operator)
- epsI [float or list, optional] Damping factor(s) for Tikhonov regularization term. If a list of \( n_m \) elements is provided, the regularization term will have different strenght for each elastic property
- epsR [float, optional] Damping factor for additional Laplacian regularization term
PyLops

PyLops

**PyLops** is a Python library for solving large-scale problems involving linear operators. It offers a wide range of functionalities such as linear operators, solvers, and auxiliary routines.

### dottest [bool, optional] Apply dot-test

**Return** [bool, optional] Return residuals

epsRL1 [float, optional] Damping factor for additional blockiness regularization term

**kwargs_solver Arbitrary keyword arguments for scipy.linalg.lstsq solver (if explicit=True and epsR=None) or scipy.sparse.linalg.lsqr solver (if explicit=False and/or epsR is not None)

**Returns**

**minv** [np.ndarray] Inverted model of size \(n_{\theta} \times n_m(\times n_x \times n_y)\]

**data_r** [np.ndarray] Residual data (i.e., data - background data) of size \(n_{\theta} \times n_\theta(\times n_x \times n_y)\]

**Notes**

The different choices of cost functions and solvers used in the seismic pre-stack inversion module follow the same convention of the seismic post-stack inversion module.

Refer to `pylops.avp.poststack.PoststackInversion` for more details.

**Examples using pylops.avp.prestack.PrestackInversion**

- 08. Pre-stack (AVO) inversion

### 3.7 PyLops Utilities

Alongside with its **Linear Operators** and **Solvers**, PyLops contains also a number of auxiliary routines performing universal tasks that are used by several operators or simply within one or more **Tutorials** for the preparation of input data and subsequent visualization of results.

#### 3.7.1 Shared

**Dot-test**

```python
@dottest(Op, nr, nc[, tol, complexflag, ...])
```

**Dot test.**

Generate random vectors \(u\) and \(v\) and perform dot-test to verify the validity of forward and adjoint operators. This test can help to detect errors in the operator implementation.

**Parameters**

- **Op** [pylops.LinearOperator] Linear operator to test.
- **nr** [int] Number of rows of operator (i.e., elements in data)
- **nc** [int] Number of columns of operator (i.e., elements in model)
tol [float, optional] Dottest tolerance
complexflag [bool, optional] generate random vectors with real (0) or complex numbers (1: only model, 2: only data, 3: both)
raiseerror [bool, optional] Raise error or simply return False when dottest fails
verb [bool, optional] Verbosity

Raises

ValueError If dot-test is not verified within chosen tolerance.

Notes

A dot-test is mathematical tool used in the development of numerical linear operators.
More specifically, a correct implementation of forward and adjoint for a linear operator should verify the following equality within a numerical tolerance:

\[(\text{Op} \ast \text{u})^H \ast \text{v} = \text{u}^H \ast (\text{Op}^H \ast \text{v})\]

Examples using pylops.utils.dottest

- sphx_glr_gallery_plot_seislet.py
- 02. The Dot-Test

3.7.2 Others

Synthetics

seismicevents.makeaxis(par) Create axes t, x, and y axes
seismicevents.linear2d(x, t, v, t0, theta, ...) Linear 2D events
seismicevents.parabolic2d(x, t, t0, px, pxx, ...) Parabolic 2D events
seismicevents.hyperbolic2d(x, t, t0, vrms, ...) Hyperbolic 2D events
seismicevents.linear3d(x, y, t, v, t0, ...) Linear 3D events
seismicevents.hyperbolic3d(x, y, t, t0, ...) Hyperbolic 3D events

pylops.utils.seismicevents.makeaxis

pylops.utils.seismicevents.makeaxis(par) Create axes t, x, and y axes
Create space and time axes from dictionary containing initial values (ot, ox, oy), sampling steps (dt, dx, dy) and number of elements (nt, nx, ny) for each axis

Parameters

par [dict] Dictionary containing initial values, sampling steps, and number of elements

Returns

t [numpy.ndarray] time axis

3.7. PyLops Utilities
**t2** [numpy.ndarray] double time axis (symmetric to zero)
**x** [numpy.ndarray] x axis
**y** [numpy.ndarray] y axis (None, if oy, dy, ny are not provided)

### Examples

```python
>>> par = {'ox':0, 'dx':2, 'nx':60,
    'oy':0, 'dy':2, 'ny':100,
    'ot':0, 'dt':4, 'nt':400}
>>> # Create axis
>>> t, t2, x, y = makeaxis(par)
```

### Examples using pylops.utils.seismicevents.makeaxis

- sphx_glr_gallery_plot_sliding.py
- sphx_glr_gallery_plot_mdc.py
- sphx_glr_gallery_plot_phaseshift.py
- sphx_glr_gallery_plot_seismicevents.py
- 09. Multi-Dimensional Deconvolution
- 11. Radon filtering
- 12. Seismic regularization
- 14. Seismic wavefield decomposition

**pylops.utils.seismicevents.linear2d**

**pylops.utils.seismicevents.linear2d**(*x, t, v, t0, theta, amp, wav*)

Linear 2D events

Create 2d linear events given propagation velocity, intercept time, angle, and amplitude of each event

**Parameters**

- **x** [numpy.ndarray] space axis
- **t** [numpy.ndarray] time axis
- **v** [float] propagation velocity
- **t0** [tuple or float] intercept time at *x* = 0 of each linear event
- **theta** [tuple or float] angle (in degrees) of each linear event
- **amp** [tuple or float] amplitude of each linear event
- **wav** [numpy.ndarray] wavelet to be applied to data

**Returns**

- **d** [numpy.ndarray] data without wavelet of size [*n_x x n_t*]
- **dwav** [numpy.ndarray] data with wavelet of size [*n_x x n_t*]
Notes

Each event is created using the following relation:

\[ t_i(x) = t_{0,i} + p_{x,i}x \]

where \[ p_{x,i} = \frac{\sin(\theta_i)}{v} \]

Examples using `pylops.utils.seismicevents.linear2d`

- sphx_glr_gallery_plot_seismicevents.py
- 11. Radon filtering
- 12. Seismic regularization

`pylops.utils.seismicevents.parabolic2d`

`pylops.utils.seismicevents.parabolic2d(x, t, t0, px, pxx, amp, wav)`

Parabolic 2D events

Create 2d parabolic events given intercept time, slowness, curvature, and amplitude of each event

Parameters

- `x` [numpy.ndarray] space axis
- `t` [numpy.ndarray] time axis
- `t0` [tuple or float] intercept time at \( x = 0 \) of each parabolic event
- `px` [tuple or float] slowness of each parabolic event
- `pxx` [tuple or float] curvature of each parabolic event
- `amp` [tuple or float] amplitude of each parabolic event
- `wav` [numpy.ndarray] wavelet to be applied to data

Returns

- `d` [numpy.ndarray] data without wavelet of size \([n_x \times n_t]\)
- `dwav` [numpy.ndarray] data with wavelet of size \([n_x \times n_t]\)

Notes

Each event is created using the following relation:

\[ t_i(x) = t_{0,i} + p_{x,i}x + p_{xx,i}x^2 \]

Examples using `pylops.utils.seismicevents.parabolic2d`

- sphx_glr_gallery_plot_sliding.py
- sphx_glr_gallery_plot_seismicevents.py
- 11. Radon filtering
pylops.utils.seismicevents.hyperbolic2d

pylops.utils.seismicevents.hyperbolic2d(x, t, t0, vrms, amp, wav)

Hyperbolic 2D events

Create 2d hyperbolic events given intercept time, root-mean-square velocity, and amplitude of each event

Parameters

- **x** [numpy.ndarray] space axis
- **t** [numpy.ndarray] time axis
- **t0** [tuple or float] intercept time at $x = 0$ of each of hyperbolic event
- **vrms** [tuple or float] root-mean-square velocity of each hyperbolic event
- **amp** [tuple or float] amplitude of each hyperbolic event
- **wav** [numpy.ndarray] wavelet to be applied to data

Returns

- **d** [numpy.ndarray] data without wavelet of size $[n_x \times n_t]$
- **dwav** [numpy.ndarray] data with wavelet of size $[n_x \times n_t]$

Notes

Each event is created using the following relation:

$$t_i(x) = \sqrt{t_{0,i}^2 + x^2/v_{rms,i}^2}$$

Examples using pylops.utils.seismicevents.hyperbolic2d

- sphx_glr_gallery_plot_mdc.py
- sphx_glr_gallery_plot_phaseshift.py
- sphx_glr_gallery_plot_seismicevents.py
- 09. Multi-Dimensional Deconvolution
- 14. Seismic wavefield decomposition

pylops.utils.seismicevents.linear3d

pylops.utils.seismicevents.linear3d(x, y, t, v, t0, theta, phi, amp, wav)

Linear 3D events

Create 3d linear events given propagation velocity, intercept time, angles, and amplitude of each event.

Parameters

- **x** [numpy.ndarray] space axis in x direction
- **y** [numpy.ndarray] space axis in y direction
- **t** [numpy.ndarray] time axis
- **v** [float] propagation velocity
t0 [tuple or float] intercept time at $x = 0$ of each linear event
theta [tuple or float] angle in x direction (in degrees) of each linear event
phi [tuple or float] angle in y direction (in degrees) of each linear event
amp [tuple or float] amplitude of each linear event
wav [numpy.ndarray] wavelet to be applied to data

Returns

d [numpy.ndarray] data without wavelet of size $[n_y \times n_x \times n_t]$
dwav [numpy.ndarray] data with wavelet of size $[n_y \times n_x \times n_t]$

Notes

Each event is created using the following relation:

$$t_i(x, y) = t_{0,i} + p_{x,i}x + p_{y,i}y$$

where $p_{x,i} = \sin(\theta_i)\cos(\phi_i)/v$ and $p_{x,i} = \sin(\theta_i)\sin(\phi_i)/v$.

Examples using `pylops.utils.seismicevents.linear3d`

• sphx_glr_gallery_plot_seismicevents.py

`pylops.utils.seismicevents.hyperbolic3d`

`pylops.utils.seismicevents.hyperbolic3d(x, y, t, t0, vrms_x, vrms_y, amp, wav)`

Hyperbolic 3D events

Create 3D hyperbolic events given intercept time, root-mean-square velocities, and amplitude of each event

Parameters

x [numpy.ndarray] space axis in x direction
y [numpy.ndarray] space axis in y direction
t [numpy.ndarray] time axis
t0 [tuple or float] intercept time at $x = 0$ of each of hyperbolic event
vrms_x [tuple or float] root-mean-square velocity in x direction for each hyperbolic event
vrms_y [tuple or float] root-mean-square velocity in y direction for each hyperbolic event
amp [tuple or float] amplitude of each hyperbolic event
wav [numpy.ndarray] wavelet to be applied to data

Returns

d [numpy.ndarray] data without wavelet of size $[n_y \times n_x \times n_t]$
dwav [numpy.ndarray] data with wavelet of size $[n_y \times n_x \times n_t]$
Notes

Each event is created using the following relation:

\[ t_i(x, y) = \sqrt{t_{0,i}^2 + x^2/v_{rms_{x,i}}^2} + y^2/v_{rms_{y,i}}^2 \]

Note that velocities do not have a physical meaning here (compared to the corresponding `pylops.utils.seismicevents.hyperbolic2d`), they rather simply control the curvature of the hyperboloid along the spatial axes.

Examples using `pylops.utils.seismicevents.hyperbolic3d`

- sphx_glr_gallery_plot_sliding.py
- sphx_glr_gallery_plot_phaseshift.py
- sphx_glr_gallery_plot_seismicevents.py

```
marchenko.directwave(wav, trav, nt, dt[, ....])  Analytical direct wave in acoustic media
```

`pylops.waveeqprocessing.marchenko.directwave`

An analytical direct wave in acoustic media

Compute the analytical acoustic 2d or 3d Green’s function in frequency domain given a wavelet `wav`, traveltime curve `trav` and distance `dist` (for 3d case only).

Parameters

- `wav` [numpy.ndarray] Wavelet in time domain to apply to direct arrival when creating `trav`. Phase will be discarded resulting in a zero-phase wavelet with same amplitude spectrum as provided by `wav`
- `trav` [numpy.ndarray] Traveltime of first arrival from subsurface point to surface receivers of size \([nr \times 1]\)
- `nt` [float, optional] Number of samples in time
- `dt` [float, optional] Sampling in time
- `nfft` [int, optional] Number of samples in fft time (if None, `nfft = nt`)
- `dist` :obj:`numpy.ndarray` Distance between subsurface point to surface receivers of size \([nr \times 1]\)
- `kind` [str, optional] 2-dimensional (2d) or 3-dimensional (3d)
- `derivative` [bool, optional] Apply time derivative (True) or not (False)

Returns

- `direct` [numpy.ndarray] Direct arrival in time domain of size \([nt \times nr]\)
Notes

The analytical Green’s function in 2D [1] is:

\[ G^{2D}(\mathbf{r}) = -\frac{i}{4} H^{(1)}_0(k|\mathbf{r}|) \]

and in 3D [1] is:

\[ G^{3D}(\mathbf{r}) = \frac{e^{-jk\mathbf{r}}}{4\pi\mathbf{r}} \]

Note that these Green’s functions represent the acoustic response to a point source of volume injection. In case the response to a point source of volume injection rate is desired, a \( j\omega \) scaling (which is equivalent to applying a first derivative in time domain) must be applied. Here this is accomplished by setting \texttt{derivative==True}.


Signal-processing

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{pylops.utils.signalprocessing.convmtx}</td>
<td>Convolution matrix</td>
</tr>
<tr>
<td>\texttt{pylops.utils.signalprocessing.nonstationary_convmtx}</td>
<td>Convolution matrix from a bank of filters</td>
</tr>
<tr>
<td>\texttt{signalprocessing.slope_estimate}</td>
<td>Local slope estimation</td>
</tr>
</tbody>
</table>

**\texttt{pylops.utils.signalprocessing.convmtx}**

\texttt{pylops.utils.signalprocessing.convmtx(h, n)}

Convolution matrix

Equivalent of MATLAB’s \texttt{convmtx} function. Makes a dense convolution matrix \( C \) such that the dot product \( \text{np.dot}(C, x) \) is the convolution of the filter \( h \) and the input signal \( x \).

- **Parameters**
  - \( h \) [np.ndarray] Convolution filter (1D array)
  - \( n \) [int] Number of columns (if \( \text{len}(h) < n \)) or rows (if \( \text{len}(h) \geq n \)) of convolution matrix

- **Returns**
  - \( C \) [np.ndarray] Convolution matrix of size \( \text{len}(h) + n - 1 \times n \) (if \( \text{len}(h) < n \)) or \( n \times \text{len}(h) + n - 1 \) (if \( \text{len}(h) \geq n \))

**\texttt{pylops.utils.signalprocessing.nonstationary_convmtx}**

\texttt{pylops.utils.signalprocessing.nonstationary_convmtx(H, n, hc=0, pad=(0, 0))}

Convolution matrix from a bank of filters

Makes a dense convolution matrix \( C \) such that the dot product \( \text{np.dot}(C, x) \) is the nonstationary convolution of the bank of filters \( H = [h_1, h_2, h_n] \) and the input signal \( x \).

- **Parameters**
  - \( H \) [np.ndarray] Convolution filters (2D array of shape \([n_{filters} \times n_h]\))
  - \( n \) [int] Number of columns of convolution matrix
PyLops

hc [np.ndarray, optional] Index of center of first filter

pad [np.ndarray] Zero-padding to apply to the bank of filters before and after the provided values (use it to avoid wrap-around or pass filters with enough padding)

Returns

C [np.ndarray] Convolution matrix

pylops.utils.signalprocessing.slope_estimate

pylops.utils.signalprocessing.slope_estimate(d, dz, dx, smooth=20)
Local slope estimation

Local slopes are estimated using the Structure Tensor algorithm [1]. Note that slopes are returned as $\arctan(\theta)$ where $\theta$ is an angle defined in a RHS coordinate system with $z$ axis pointing upward.

Parameters

- d [np.ndarray] Input dataset of size $n_z \times n_x$
- dz [float] Sampling in $z$-axis
- dx [float] Sampling in $x$-axis
- smooth [float, optional] Length of smoothing filter to be applied to the estimated gradients

Returns

- slopes [np.ndarray] Estimated local slopes
- linearity [np.ndarray] Estimated linearity

Notes

For each pixel of the input dataset $d$ the local gradients $dd/dz$ and $g_z = dd/dx$ are computed and used to define the following three quantities: $g_{zz} = (dd/dz)^2$, $g_{xx} = (dd/dx)^2$, and $g_{xz} = dd/dz \times dd/dx$. Such quantities are spatially smoothed and at each pixel their smoothed versions are arranged in a $2 \times 2$ matrix called the smoothed gradient-square tensor:

$$G = \begin{bmatrix} g_{zz} & g_{xz} \\ g_{xz} & g_{xx} \end{bmatrix}$$

Local slopes can be expressed as $p = \arctan(\frac{\lambda_{max} - g_{xx}}{g_{xz}})$.

Examples using pylops.utils.signalprocessing.slope_estimate

- sphx_glr_gallery_plot_seislet.py
- sphx_glr_gallery_plot_slopeest.py

Tapers

tapers.taper2d(nt, nmask, ntap[, tapertype]) 2D taper
tapers.taper3d(nt, nmask, ntap[, tapertype]) 3D taper
pylops.utils.tapers.taper2d

`pylops.utils.tapers.taper2d(nt, nmask, ntap, tapertype='hanning')`

2D taper

Create 2d mask of size \([n_{\text{mask}} \times n_t]\) with tapering of size \(n_{\text{tap}}\) along the first dimension

Parameters

- `nt [int]` Number of time samples of mask along second dimension
- `nmask [int]` Number of space samples of mask along first dimension
- `ntap [int]` Number of samples of tapering at edges of first dimension
- `tapertype [str, optional]` Type of taper (`hanning`, `cosine`, `cosinesquare` or `None`)

Returns

- `taper [numpy.ndarray]` 2d mask with tapering along first dimension of size \([n_{\text{mask}} \times n_t]\)

Examples using `pylops.utils.tapers.taper2d`

- sphx_glr_gallery_plot_phaseshift.py
- sphx_glr_gallery_plot_tapers.py

pylops.utils.tapers.taper3d

`pylops.utils.tapers.taper3d(nt, nmask, ntap, tapertype='hanning')`

3D taper

Create 2d mask of size \([n_{\text{mask}}[0] \times n_{\text{mask}}[1] \times n_t]\) with tapering of size \(n_{\text{tap}}\) along the first and second dimension

Parameters

- `nt [int]` Number of time samples of mask along third dimension
- `nmask [tuple]` Number of space samples of mask along first dimension
- `ntap [tuple]` Number of samples of tapering at edges of first dimension
- `tapertype [int]` Type of taper (`hanning`, `cosine`, `cosinesquare` or `None`)

Returns

- `taper [numpy.ndarray]` 2d mask with tapering along first dimension of size \([n_{\text{mask},0} \times n_{\text{mask},1} \times n_t]\)

Examples using `pylops.utils.tapers.taper3d`

- sphx_glr_gallery_plot_mdc.py
- sphx_glr_gallery_plot_phaseshift.py
- sphx_glr_gallery_plot_tapers.py
- 09. Multi-Dimensional Deconvolution

3.7. PyLops Utilities
PyLops

Wavelets

<table>
<thead>
<tr>
<th>wavelets.ricker(t[, f0])</th>
<th>Ricker wavelet</th>
</tr>
</thead>
<tbody>
<tr>
<td>wavelets.gaussian(t[, std])</td>
<td>Gaussian wavelet</td>
</tr>
</tbody>
</table>

pylops.utils.wavelets.ricker

pylops.utils.wavelets.ricker(t, f0=10)

Ricker wavelet

Create a Ricker wavelet given time axis t and central frequency \( f_0 \)

**Parameters**

- **t** [numpy.ndarray] Time axis (positive part including zero sample)
- **f0** [float, optional] Central frequency

**Returns**

- **w** [numpy.ndarray] Wavelet
- **t** [numpy.ndarray] Symmetric time axis
- **wcenter** [int] Index of center of wavelet

**Examples using pylops.utils.wavelets.ricker**

- sphx_glr_gallery_plot_sliding.py
- sphx_glr_gallery_plot_avo.py
- sphx_glr_gallery_plot_convolve.py
- sphx_glr_gallery_plot_ista.py
- sphx_glr_gallery_plot_mdc.py
- sphx_glr_gallery_plot_phaseshift.py
- sphx_glr_gallery_plot_prestack.py
- sphx_glr_gallery_plot_slopeest.py
- sphx_glr_gallery_plot_seismicwaves.py
- sphx_glr_gallery_plot_wave.py

- 07. Post-stack inversion
- 08. Pre-stack (AVO) inversion
- 09. Multi-Dimensional Deconvolution
- 11. Radon filtering
- 12. Seismic regularization
- 14. Seismic wavefield decomposition
- 15. Least-squares migration
pylops.utils.wavelets.gaussian

pylops.utils.wavelets.gaussian(t, std=1)
Gaussian wavelet

Create a Gaussian wavelet given time axis t and standard deviation std using scipy.signal.windows.gaussian.

Parameters
- t [numpy.ndarray] Time axis (positive part including zero sample)
- std [float, optional] Standard deviation of gaussian

Returns
- w [numpy.ndarray] Wavelet
- t [numpy.ndarray] Symmetric time axis
- wcenter [int] Index of center of wavelet

Geophysical Reservoir characterization

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>avo.zoeppritz_scattering</td>
<td>Zoeppritz solution.</td>
</tr>
<tr>
<td>avo.zoeppritz_element</td>
<td>Single element of Zoeppritz solution.</td>
</tr>
<tr>
<td>avo.zoeppritz_pp</td>
<td>PP reflection coefficient from the Zoeppritz scattering matrix.</td>
</tr>
<tr>
<td>avo.approx_zoeppritz_pp</td>
<td>PP reflection coefficient from the approximate Zoeppritz equation.</td>
</tr>
<tr>
<td>avo.akirichards</td>
<td>Three terms Aki-Richards approximation.</td>
</tr>
<tr>
<td>avo.fatti</td>
<td>Three terms Fatti approximation.</td>
</tr>
</tbody>
</table>

pylops.avo.avo.zoeppritz_scattering

pylops.avo.avo.zoeppritz_scattering(vp1, vs1, rho1, ...
Zoeppritz solution.

Calculates the angle dependent p-wave reflectivity of an interface between two media for a set of incident angles.

Parameters
- vp1 [float] P-wave velocity of the upper medium
- vs1 [float] S-wave velocity of the upper medium
- rho1 [float] Density of the upper medium
- vp0 [float] P-wave velocity of the lower medium
- vs0 [float] S-wave velocity of the lower medium
- rho0 [float] Density of the lower medium
- theta1 [np.ndarray or float] Incident angles in degrees

Returns
\textbf{zoepep} [\texttt{np.ndarray}] $4 \times 4$ matrix representing the scattering matrix for the incident angle $\theta_1$

See also:

\textbf{zoeppritz\_element} Single reflectivity element of Zoeppritz solution

\textbf{zoeppritz\_PP} PP reflectivity element of Zoeppritz solution

\texttt{pylops.avo.avo.zoeppritz\_element}

\texttt{pylops.avo.avo.zoeppritz\_element} ($vp_1, vs_1, rho_1, vp_0, vs_0, rho_0, theta_1, element='PdPu'$)  
Single element of Zoeppritz solution.

Simple wrapper to \texttt{pylops.avo.avo.scattering\_matrix}, returning any mode reflection coefficient from the Zoeppritz scattering matrix for specific combination of incident and reflected wave and a set of incident angles

\textbf{Parameters}

- $vp_1$ [\texttt{float}] P-wave velocity of the upper medium
- $vs_1$ [\texttt{float}] S-wave velocity of the upper medium
- $rho_1$ [\texttt{float}] Density of the upper medium
- $vp_0$ [\texttt{float}] P-wave velocity of the lower medium
- $vs_0$ [\texttt{float}] S-wave velocity of the lower medium
- $rho_0$ [\texttt{float}] Density of the lower medium
- $theta_1$ [\texttt{np.ndarray} or \texttt{float}] Incident angles in degrees
- $element$ [\texttt{str}, optional] specific choice of incident and reflected wave combining any two of the following strings: Pd P-wave downgoing, Sd S-wave downgoing, Pu P-wave upgoing, Su S-wave upgoing (e.g., PdPu)

\textbf{Returns}

- $refl$ [\texttt{np.ndarray}] reflectivity values for all input angles for specific combination of incident and reflected wave.

See also:

\textbf{zoeppritz\_scattering} Zoeppritz solution

\textbf{zoeppritz\_PP} PP reflectivity element of Zoeppritz solution

\texttt{pylops.avo.avo.zoeppritz\_pp}

\texttt{pylops.avo.avo.zoeppritz\_pp} ($vp_1, vs_1, rho_1, vp_0, vs_0, rho_0, theta_1$)  
PP reflection coefficient from the Zoeppritz scattering matrix.

Simple wrapper to \texttt{pylops.avo.avo.scattering\_matrix}, returning the PP reflection coefficient from the Zoeppritz scattering matrix for a set of incident angles

\textbf{Parameters}

- $vp_1$ [\texttt{float}] P-wave velocity of the upper medium
- $vs_1$ [\texttt{float}] S-wave velocity of the upper medium
rho1 [float] Density of the upper medium
vp0 [float] P-wave velocity of the lower medium
vs0 [float] S-wave velocity of the lower medium
rho0 [float] Density of the lower medium
theta1 [np.ndarray or float] Incident angles in degrees

Returns
PPrefl [np.ndarray] PP reflectivity values for all input angles.

See also:

zoeppritz_scattering Zoeppritz solution
zoeppritz_element Single reflectivity element of Zoeppritz solution

pylops.avo.avo.approx_zoeppritz_pp

pylops.avo.avo.approx_zoeppritz_pp(vp1, vs1, rho1, vp0, vs0, rho0, theta1)
PP reflection coefficient from the approximate Zoeppritz equation.
Approximate calculation of PP reflection from the Zoeppritz scattering matrix for a set of incident angles [1].

Parameters
vp1 [np.ndarray or list or tuple] P-wave velocity of the upper medium
vs1 [np.ndarray or list or tuple] S-wave velocity of the upper medium
rho1 [np.ndarray or list or tuple] Density of the upper medium
vp0 [np.ndarray or list or tuple] P-wave velocity of the lower medium
vs0 [np.ndarray or list or tuple] S-wave velocity of the lower medium
rho0 [np.ndarray or list or tuple] Density of the lower medium
theta1 [np.ndarray or float] Incident angles in degrees

Returns
PPrefl [np.ndarray] PP reflectivity values for all input angles.

See also:

zoeppritz_scattering Zoeppritz solution
zoeppritz_element Single reflectivity element of Zoeppritz solution
zoeppritz_pp PP reflectivity element of Zoeppritz solution

pylops.avo.avo.akirichards

pylops.avo.avo.akirichards(theta, vsvp, n=1)
Three terms Aki-Richards approximation.
Computes the coefficients of the of three terms Aki-Richards approximation for a set of angles and a constant or variable VS/VP ratio.

Parameters
theta [np.ndarray] Incident angles in degrees
vsvp [np.ndarray or float] VS/VP ratio
n [int, optional] number of samples (if vsvp is a scalar)

Returns

G1 [np.ndarray] first coefficient of three terms Aki-Richards approximation \[ n_{\text{theta}} \times n_{\text{vsvp}} \]
G2 [np.ndarray] second coefficient of three terms Aki-Richards approximation \[ n_{\text{theta}} \times n_{\text{vsvp}} \]
G3 [np.ndarray] third coefficient of three terms Aki-Richards approximation \[ n_{\text{theta}} \times n_{\text{vsvp}} \]

Notes

The three terms Aki-Richards approximation is used to compute the reflection coefficient as linear combination of contrasts in \( V_P \), \( V_S \), and \( \rho \). More specifically:

\[
R(\theta) = G_1(\theta) \frac{\Delta V_P}{V_P} + G_2(\theta) \frac{\Delta V_S}{V_S} + G_3(\theta) \frac{\Delta \rho}{\rho}
\]

where \( G_1(\theta) = \frac{1}{2 \cos^2 \theta} \), \( G_2(\theta) = -4 \frac{V_S}{V_P} \sin^2 \theta \), \( G_3(\theta) = 0.5 - 2 \frac{V_S}{V_P} \sin^2 \theta \), \( \Delta V_P = 2 \frac{V_{p,2} - V_{p,1}}{V_{p,2} + V_{p,1}} \), and \( \Delta \rho = 2 \frac{\rho_2 - \rho_1}{\rho_2 + \rho_1} \).

pylops.avo.avo.fatti

pylops.avo.avo.fatti (theta, vsvp, n=1)
Three terms Fatti approximation.
Computes the coefficients of the three terms Fatti approximation for a set of angles and a constant or variable VS/VP ratio.

Parameters

theta [np.ndarray] Incident angles in degrees
vsvp [np.ndarray or float] VS/VP ratio
n [int, optional] number of samples (if vsvp is a scalar)

Returns

G1 [np.ndarray] first coefficient of three terms Smith-Gidlow approximation \[ n_{\text{theta}} \times n_{\text{vsvp}} \]
G2 [np.ndarray] second coefficient of three terms Smith-Gidlow approximation \[ n_{\text{theta}} \times n_{\text{vsvp}} \]
G3 [np.ndarray] third coefficient of three terms Smith-Gidlow approximation \[ n_{\text{theta}} \times n_{\text{vsvp}} \]
The three terms Fatti approximation is used to compute the reflection coefficient as linear combination of contrasts in \( AI, SI, \) and \( \rho \). More specifically:

\[
R(\theta) = G_1(\theta) \frac{\Delta AI}{AI} + G_2(\theta) \frac{\Delta SI}{SI} + G_3(\theta) \frac{\Delta \rho}{\rho},
\]

where \( G_1(\theta) = 0.5(1 + \tan^2 \theta) \), \( G_2(\theta) = -4(V_S/V_P)^2 \sin^2 \theta \), \( G_3(\theta) = 0.5(4(V_S/V_P)^2 \sin^2 \theta - \tan^2 \theta) \),

\[
\frac{\Delta AI}{AI} = 2 \frac{A_{I2} - A_{I1}}{A_{I2} + A_{I1}}. \quad \frac{\Delta SI}{SI} = 2 \frac{S_{I2} - S_{I1}}{S_{I2} + S_{I1}}. \quad \frac{\Delta \rho}{\rho} = 2 \frac{\rho_2 - \rho_1}{\rho_2 + \rho_1}.
\]

### 3.8 Implementing new operators

Users are welcome to create new operators and add them to the PyLops library.

In this tutorial, we will go through the key steps in the definition of an operator, using the `pylops.Diagonal` as an example. This is a very simple operator that applies a diagonal matrix to the model in forward mode and to the data in adjoint mode.

#### 3.8.1 Creating the operator

The first thing we need to do is to create a new file with the name of the operator we would like to implement. Note that as the operator will be a class, we need to follow the UpperCaseCamelCase convention both for the class itself and for the filename.

Once we have created the file, we will start by importing the modules that will be needed by the operator. While this varies from operator to operator, you will always need to import the `pylops.LinearOperator` class, which will be used as parent class for any of our operators:

```python
from pylops import LinearOperator
```

This class is a child of the `scipy.sparse.linalg.LinearOperator` class itself which implements the same methods of its parent class as well as an additional method for quick inversion: such method can be easily accessed by using \( \backslash \) between the operator and the data (e.g., \( A \backslash y \)).

After that we define our new object:

```python
class Diagonal(LinearOperator):
```

followed by a `numpydoc` docstring (starting with `""""` and ending with `""""`) containing the documentation of the operator. Such docstring should contain at least a short description of the operator, a `Parameters` section with a detailed description of the input parameters and a `Notes` section providing a mathematical explanation of the operator. Take a look at some of the core operators of PyLops to get a feeling of the level of details of the mathematical explanation.

We then need to create the `__init__` where the input parameters are passed and saved as members of our class. While the input parameters change from operator to operator, it is always required to create three members, the first called `shape` with a tuple containing the dimensions of the operator in the data and model space, the second called `dtype` with the data type object (np.dtype) of the model and data, and the third called `explicit` with a boolean (True or False) identifying if the operator can be inverted by a direct solver or requires an iterative solver. This member is True if the operator has also a member `A` that contains the matrix to be inverted like for example in the `pylops.MatrixMult` operator, and it will be False otherwise. In this case we have another member called `d` which is equal to the input vector containing the diagonal elements of the matrix we want to multiply to the model and data.
We can then move onto writing the forward mode in the method `__matvec`. In other words, we will need to write the piece of code that will implement the following operation $y = Ax$. Such method is always composed of two inputs (the object itself `self` and the input model `x`). In our case the code to be added to the forward is very simple, we will just need to apply element-wise multiplication between the model `x` and the elements along the diagonal contained in the array `d`. We will finally need to return the result of this operation:

```python
def __matvec__(self, x):
    return self.d * x
```

Finally we need to implement the adjoint mode in the method `__rmatvec`. In other words, we will need to write the piece of code that will implement the following operation $x = A^H y$. Such method is also composed of two inputs (the object itself `self` and the input data `y`). In our case the code to be added to the forward is the same as the one from the forward (but this will obviously be different from operator to operator):

```python
def __rmatvec__(self, x):
    return self.d * x
```

And that’s it, we have implemented our first linear operator!

### 3.8.2 Testing the operator

Being able to write an operator is not yet a guarantee of the fact that the operator is correct, or in other words that the adjoint code is actually the adjoint of the forward code. Luckily for us, a simple test can be performed to check the validity of forward and adjoint operators, the so called dot-test.

We can generate random vectors `u` and `v` and verify the the following equality within a numerical tolerance:

$$(A * u)^H * v = u^H * (A^H * v)$$

The method `pylops.utils.dottest` implements such a test for you, all you need to do is create a new test within an existing `test_*.py` file in the `pytests` folder (or in a new file).

Generally a test file will start with a number of dictionaries containing different parameters we would like to use in the testing of one or more operators. The test itself starts with a `decorator` that contains a list of all (or some) of dictionaries that will would like to use for our specific operator, followed by the definition of the test

```python
@pytest.mark.parametrize("par", [(par1), (par2)])
def test_Diagonal(par):
```

At this point we can first of all create the operator and run the `pylops.utils.dottest` preceded by the `assert` command. Moreover, the forward and adjoint methods should tested towards expected outputs or even better, when the operator allows it (i.e., operator is invertible), a small inversion should be run and the inverted model tested towards the input model.

```python
"""Dot-test and inversion for diagonal operator"""

    d = np.arange(par['nx']) + 1.
    Dop = Diagonal(d)
```

(continues on next page)
assert dottest(Dop, par['nx'], par['nx'], complexflag=0 if par['imag'] == 1 else 3)

x = np.ones(par['nx'])
xlsqr = lsqr(Dop, Dop * x, damp=1e-20, iter_lim=300, show=0)[0]
assert_array_almost_equal(x, xlsqr, decimal=4)

3.8.3 Documenting the operator

Once the operator has been created, we can add it to the documentation of PyLops. To do so, simply add the name of the operator within the index.rst file in docs/source/api directory.

Moreover, in order to facilitate the user of your operator by other users, a simple example should be provided as part of the Sphinx-gallery of the documentation of the PyLops library. The directory examples contains several scripts that can be used as template.

3.8.4 Final checklist

Before submitting your new operator for review, use the following checklist to ensure that your code adheres to the guidelines of PyLops:

- you have created a new file containing a single class (or a function when the new operator is a simple combination of existing operators - see pylops.Laplacian for an example of such operator) and added to a new or existing directory within the pylops package.
- the new class contains at least __init__, _matvec and _matvec methods.
- the new class (or function) has a numpydoc docstring documenting at least the input Parameters and with a Notes section providing a mathematical explanation of the operator
- a new test has been added to an existing test_*.py file within the pytests folder. The test should verify that the new operator passes the pylops.utils.dottest. Moreover it is advisable to create a small toy example where the operator is applied in forward mode and the resulting data is inverted using \ from pylops.LinearOperator.
- the new operator is used within at least one example (in examples directory) or one tutorial (in tutorials directory).

3.9 Contributing

Contributions are welcome and greatly appreciated!

The best way to get in touch with the core developers and maintainers is to join the PyLops slack channel as well as open new Issues directly from the github repo.

Moreover, take a look at the Roadmap page for a list of current ideas for improvements and additions to the PyLops library.

3.9.1 Types of Contributions
Report Bugs

If you are playing with the PyLops library and find a bug, please report it including:

• Your operating system name and version.
• Any details about your Python environment.
• Detailed steps to reproduce the bug.

Propose New Operators or Features

Open an issue at https://github.com/equinor/pylops/issues with tag enhancement.
If you are proposing a new operator or a new feature:

• Explain in detail how it should work.
• Keep the scope as narrow as possible, to make it easier to implement.

Implement Operators or Features

Look through the Git issues for operator or feature requests. Anything tagged with enhancement is open to whoever wants to implement it.

Add Examples or Improve Documentation

Writing new operators is not the only way to get involved and contribute. Create examples with existing operators as well as improving the documentation of existing operators is as important as making new operators and very much encouraged.

3.9.2 Getting Started to contribute

Ready to contribute?

1. Fork the PyLops repo.
2. Clone your fork locally:

```bash
$ git clone https://github.com/your_name_here/pylops.git
```
3. Follow the installation instructions for developers that you find in Installation page. Ensure that you are able to pass all the tests before moving forward.
4. Add the main repository to the list of your remotes (this will be important to ensure you pull the latest changes before trying to merge your local changes):

```bash
$ git remote add upstream https://github.com/equinor/pylops
```
5. Create a branch for local development:

```bash
$ git checkout -b name-of-your-branch
```
Now you can make your changes locally.

6. When you’re done making changes, check that your code follows the guidelines for Implementing new operators and that the both old and new tests pass successfully:

```
>> make tests
```

7. Commit your changes and push your branch to GitLab:

```
>> git add .
>> git commit -m "Your detailed description of your changes."
>> git push origin name-of-your-branch
```

Remember to add \texttt{--u} when pushing the branch for the first time.

8. Submit a pull request through the GitHub website.

### 3.9.3 Pull Request Guidelines

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

1. The pull request should include new tests for all the core routines that have been developed.
2. If the pull request adds functionality, the docs should be updated accordingly.
3. Ensure that the updated code passes all tests.

### 3.10 Changelog

#### 3.10.1 Version 1.10.0

*Released on: 13/08/2020*

- Added \texttt{tosparses} method to \texttt{pylops.LinearOperator}.
- Added \texttt{kind=linear} in \texttt{pylops.signalprocessing.Seislet} operator.
- Added \texttt{kind} to \texttt{pylops.FirstDerivative} operator to perform forward and backward (as well as centered) derivatives.
- Added \texttt{kind} to \texttt{pylops.optimization.sparsity.irls} solver to choose between data or model sparsity.
- Added possibility to use \texttt{scipy.sparse.linalg.lobpcg} in \texttt{pylops LinearOperator.eigs} and \texttt{pylops.LinearOperator.cond}.
- Added possibility to use \texttt{scipy.signal.oaconvolve} in \texttt{pylops.signalprocessing.Convolve1D}.
- Added \texttt{NRegs} to \texttt{pylops.optimization.leastsquares.NormalEquationsInversion} to allow providing regularization terms directly in the form of $H^T H$.

#### 3.10.2 Version 1.9.1

*Released on: 25/05/2020*

- Changed internal behaviour of \texttt{pylops.sparsity.omp} when \texttt{niter_inner=0}. Automatically reverts to Matching Pursuit algorithm.
• Changed handling of dtype in `pylops.signalprocessing.FFT` and `pylops.signalprocessing.FFT2D` to ensure that the type of the input vector is retained when applying forward and adjoint.

• Added dtype parameter to the FFT calls in the definition of the `pylops.waveeqprocessing.MDD` operation. This ensure that the type of the real part of G input is enforced to the output vectors of the forward and adjoint operations.

### 3.10.3 Version 1.9.0

Released on: 13/04/2020

• Added `pylops.waveeqprocessing.Deghosting` and `pylops.signalprocessing.Seislet` operators

• Added hard and half thresholds in `pylops.optimization.sparsityISTA` and `pylops.optimization.sparsity.FISTA` solvers

• Added prescaled input parameter to `pylops.waveeqprocessing.MDC` and `pylops.waveeqprocessing.Marchenko`

• Added sinc interpolation to `pylops.signalprocessing.Interp(kind == 'sinc')`

• Modified `pylops.waveeqprocessing.marchenko.directwave` to to model analytical responses from both sources of volume injection (derivative=False) and source of volume injection rate (derivative=True)

• Added `pylops.LinearOperator.asoperator` method to `pylops.LinearOperator`

• Added `pylops.utils.signalprocessing.slope_estimate` function

• Fix bug in `pylops.signalprocessing.Radon2D` and `pylops.signalprocessing.Radon3D` when onthefly=True returning the same result as when onthefly=False

### 3.10.4 Version 1.8.0

Released on: 12/01/2020

• Added `pylops.LinearOperator.todense` method to `pylops.LinearOperator`

• Added `pylops.signalprocessing.Bilinear`, `pylops.signalprocessing.DWT`, and `pylops.signalprocessing.DWT2` operators

• Added `pylops.waveeqprocessing.PressureToVelocity`, `pylops.waveeqprocessingUpDownComposition3D`, and `pylops.waveeqprocessing.PhaseShift` operators

• Fix bug in `pylops.basicoperators.Kronecker` (see Issue #125)

### 3.10.5 Version 1.7.0

Released on: 10/11/2019

• Added `pylops.Gradient`, `pylops.Sum`, `pylops.FirstDirectionalDerivative`, and `pylops.SecondDirectionalDerivative` operators

• Added `pylops.LinearOperator._ColumnLinearOperator` private operator

• Added possibility to directly mix Linear operators and numpy/scipy 2d arrays in `pylops.VStack` and `pylops.HStack` and `pylops.BlockDiag` operators
• Added `pylops.optimization.sparsity.OMP` solver

3.10.6 Version 1.6.0

Released on: 10/08/2019

• Added `pylops.signalprocessing.ConvolveND` operator
• Added `pylops.utils.signalprocessing.nonstationary_convmtx` to create matrix for non-stationary convolution
• Added possibility to perform seismic modelling (and inversion) with non-stationary wavelet in `pylops.avo.poststack.PoststackLinearModelling`
• Create private methods for `pylops.Block`, `pylops.avo.poststack.PoststackLinearModelling`, `pylops.waveeqprocessing.MDC` to allow calling different operators (e.g., from pylops-distributed or pylops-gpu) within the method

3.10.7 Version 1.5.0

Released on: 30/06/2019

• Added `conj` method to `pylops.LinearOperator`
• Added `pylops.Kronecker, pylops.Roll, and pylops.Transpose` operators
• Added `pylops.signalprocessing.Fredholm1` operator
• Added `pylops.optimization.sparsity.SPGL1 and pylops.optimization.sparsity.SplitBregman` solvers
• Sped up `pylops.signalprocessing.Convolve1D` using `scipy.signal.fftconvolve` for multi-dimensional signals
• Changes in implementation of `pylops.waveeqprocessing.MDC` and `pylops.waveeqprocessing.Marchenko` to take advantage of primitives operators
• Added `epsRL1` option to `pylops.avo.poststack.PoststackInversion` and `pylops.avo.prestack.PrestackInversion` to include TV-regularization terms by means of `pylops.optimization.sparsity.SplitBregman` solver

3.10.8 Version 1.4.0

Released on: 01/05/2019

• Added `numba` engine to `pylops.Spread` and `pylops.signalprocessing.Radon2D` operators
• Added `pylops.signalprocessing.Radon3D` operator
• Added `pylops.signalprocessing.Sliding2D` and `pylops.signalprocessing.Sliding3D` operators
• Added `pylops.signalprocessing.FFTND` operator
• Added `pylops.signalprocessing.Radon3D` operator
• Added `niter` option to `pylops.LinearOperator.eigs` method
• Added `show` option to `pylops.optimization.sparsityISTA and pylops.optimization.sparsity.FISTA` solvers
• Added `pylops.waveeqprocessing.seismicinterpolation`, `pylops.waveeqprocessing.waveeqdecomposition` and `pylops.waveeqprocessing.lsm` submodules
• Added tests for engine in various operators
• Added documentation regarding usage of pylops Docker container

3.10.9 Version 1.3.0

Released on: 24/02/2019

• Added `fftw` engine to `pylops.signalprocessing.FFT` operator
• Added `pylops.optimization.sparsity.ISTA` and `pylops.optimization.sparsity.FISTA` sparse solvers
• Added possibility to broadcast (handle multi-dimensional arrays) to `pylops.Diagonal` and `pylops..Restriction` operators
• Added `pylops.signalprocessing.Interp` operator
• Added `pylops.Spread` operator
• Added `pylops.signalprocessing.Radon2D` operator

3.10.10 Version 1.2.0

Released on: 13/01/2019

• Added `pylops.LinearOperator.eigs` and `pylops.LinearOperator.cond` methods to estimate eigenvalues and conditioning number using scipy wrapping of ARPACK
• Modified default dtype for all operators to be `float64` (or `complex128`) to be consistent with default dtypes used by numpy (and scipy) for real and complex floating point numbers.
• Added `pylops.CausalIntegration` operator
• Added `pylops.Symmetrize` operator
• Added `pylops.Block` operator
• Added `pylops.Regression` operator performing polynomial regression and modified `pylops.LinearRegression` to be a simple wrapper of `pylops.Regression` when order=1
• Modified `pylops.MatrixMult` operator to work with both numpy ndarrays and scipy sparse matrices
• Added `pylops.avo.prestack.PrestackInversion` routine
• Added possibility to have a data weight via `Weight` input parameter to `pylops.optimization.leastsquares.NormalEquationsInversion` and `pylops.optimization.leastsquares.RegularizedInversion` solvers
• Added `pylops.optimization.sparsity.IRLS` solver

3.10.11 Version 1.1.0

Released on: 13/12/2018

• Added `pylops.CausalIntegration` operator
3.10.12 Version 1.0.1

Released on: 09/12/2018

- Changed module from lops to pylops for consistency with library name (and pip install).
- Removed quickplots from utilities and matplotlib from requirements of PyLops.

3.10.13 Version 1.0.0

Released on: 04/12/2018

- First official release.

3.11 Roadmap

This roadmap is aimed at providing an high-level overview on the bug fixes, improvements and new functionality that are planned for the PyLops library.

Any of the fixes/additions mentioned in the roadmap are directly linked to a Github Issue that provides more details onto the reason and initial thoughts for the implementation of such a fix/addition.

Striked tasks have been completed and related github issue closed with more details regarding how this task has been carried out.

3.11.1 Library structure

- Create a child repository and python library called geolops (just a suggestion) where geoscience-related operators and examples are moved across, keeping the core pylops library very generic and multi-purpose - Issue #22.

3.11.2 Code cleaning

- Change all np.flatten() into np.ravel() - Issue #24.
- Fix all if: return ... else: ... statements to enforce a single return with the same number of outputs - Issue #26.
- Protected attributes and @property attributes in linear operator classes? - Issue #27.

3.11.3 Code optimization

- Investigate speed-up given by decorating _matvec and _rmatvec methods with numba @jit and @stencil decorators - Issue #23.
- Replace np.fft.* routines used in several submodules with double engine, numpy and pyFFTW - Issue #20.
3.11.4 Modules

avo

- Add possibility to choose different damping factors for each elastic parameter to invert for in `pylops.avo.prestack.PrestackInversion` - Issue #25.

basicoperators

- Create Kronecker operator - Issue #28.
- Deal with edges in FirstDerivative and SecondDerivative operators - Issue #34.

optimization

- Sparse solvers - Issue #44.

signalprocessing

- Compare performance in FTT operator of performing `np.swap+np.fft.fft(…, axis=-1)` versus `np.fft.fft(…, axis=chosen)` - Issue #33.
- Add `Wavelet` operator performing the wavelet transform. `pywavelets` can be used as back-end - Issue #21.
- Fredholm1 operator applying Fredholm integrals of first kind - Issue #31.
- Fredholm2 operators applying Fredholm integrals of second kind - Issue #31.

utils

Nothing so far

waveeqprocessing

- `numpy.matmul` as a way to speed up integral computation (i.e., inner for loop) in the `MDC` operator - Issue #32.
- NMO operator performing NMO modelling - Issue #29.
- WavefieldDecomposition operator performing acoustic wavefield separation by inversion - Issue #30.

3.12 Papers

This section lists various conference abstracts and papers using the PyLops framework:

3.13 Citing

When using pylops in scientific publications, please cite the following paper:


3.14 Contributors

- Matteo Ravasi, mrava87
- Carlos da Costa, cako
- Dieter Werthmüller, prisae
- Tristan van Leeuwen, TristanvanLeeuwen
- Leonardo Uieda, leouieda
- Filippo Broggini, filippo82
- Tyler Hughes, twhughes
Bibliography


[1] Snieder, R. “A Guided Tour of Mathematical Methods for the

A
akirichards() (in module pylops.avo.avo), 243
apply() (pylops.Regression method), 152
apply_columns() (pylops.LinearOperator method), 134
apply_multiplepoints() (pylops.waveeqprocessing.Marchenko method), 225
apply_onepoint() (pylops.waveeqprocessing.Marchenko method), 225
approx_zoeppritz_pp() (in module pylops.avo.avo), 243
AVOLinearModelling (class in pylops.avo.avo), 199

B
Bilinear (class in pylops.signalprocessing), 176
Block() (in module pylops), 159
BlockDiag (class in pylops), 160

C
CausalIntegration (class in pylops), 153
cond() (pylops.LinearOperator method), 133
conj() (pylops.LinearOperator method), 134
convmtx() (in module pylops.utils.signalprocessing), 237
Convolve1D (class in pylops.signalprocessing), 170
Convolve2D() (in module pylops.signalprocessing), 172
ConvolveND (class in pylops.signalprocessing), 173

D
Deghosting() (in module pylops.waveeqprocessing), 219
Demigration() (in module pylops.waveeqprocessing), 198
Diagonal (class in pylops), 141
directwave() (in module pylops.waveeqprocessing.marchenko), 236
div() (pylops.LinearOperator method), 132
dottest() (in module pylops.utils), 230
DWT (class in pylops.signalprocessing), 181
DWT2D (class in pylops.signalprocessing), 182

E
eigs() (pylops.LinearOperator method), 133

F
fatti() (in module pylops.avo.avo), 244
FFT() (in module pylops.signalprocessing), 177
FFT2D (class in pylops.signalprocessing), 178
FFTND (class in pylops.signalprocessing), 180
FirstDerivative (class in pylops), 164
FirstDirectionalDerivative() (in module pylops), 168
FISTA() (in module pylops.optimization.sparsity), 212
Flip (class in pylops), 143
Fredholm1 (class in pylops.signalprocessing), 190
FunctionOperator (class in pylops), 136

G
gaussian() (in module pylops.utils.wavelets), 241
Gradient() (in module pylops), 167

H
HStack (class in pylops), 158
hyperbolic2d() (in module pylops.utils.seismicevents), 234
hyperbolic3d() (in module pylops.utils.seismicevents), 235

I
Identity (class in pylops), 138
Interp() (in module pylops.signalprocessing), 174
inv() (pylops.MatrixMult method), 138
IRLS() (in module pylops.optimization.sparsity), 208
ISTA() (in module pylops.optimization.sparsity), 210
K
Kronecker (class in pylops), 161
L
Laplacian() (in module pylops), 166
linear2d() (in module pylops.utils.seismicevents), 232
linear3d() (in module pylops.utils.seismicevents), 234
LinearOperator (class in pylops), 131
LinearRegression() (in module pylops), 152
LSM (class in pylops.waveeqprocessing), 226
Marchenko (class in pylops.waveeqprocessing), 223
mask() (pylops.Restriction method), 150
matrix() (pylops.Diagonal method), 142
MatrixMult (class in pylops), 137
MDC() (in module pylops.waveeqprocessing), 195
MDD() (in module pylops.waveeqprocessing), 222
N
nonstationary_convmtx() (in module pylops.utils.signalprocessing), 237
Normalequationsinversion() (in module pylops.optimization.leastsquares), 204
O
OMP() (in module pylops.optimization.sparsity), 209
P
Pad (class in pylops), 145
parabolic2d() (in module pylops.utils.seismicevents), 233
PhaseShift() (in module pylops.waveeqprocessing), 197
PoststackInversion() (in module pylops.avo.poststack), 228
PoststackLinearModelling() (in module pylops.avo.poststack), 200
PreconditionedInversion() (in module pylops.optimization.leastsquares), 206
PressureToVelocity() (in module pylops.waveeqprocessing), 192
PrestackInversion() (in module pylops.avo.prestack), 229
PrestackLinearModelling() (in module pylops.avo.prestack), 201
PrestackWaveletModelling() (in module pylops.avo.prestack), 202
R
Radon2D() (in module pylops.signalprocessing), 186
Radon3D() (in module pylops.signalprocessing), 187
Regression (class in pylops), 150
RegularizedInversion() (in module pylops.optimization.leastsquares), 205
Restriction (class in pylops), 149
ricker() (in module pylops.utils.wavelists), 240
Roll (class in pylops), 144
S
SecondDerivative (class in pylops), 165
SecondDirectionalDerivative() (in module pylops), 169
Seislet (class in pylops.signalprocessing), 183
SeismicInterpolation() (in module pylops.waveeqprocessing), 217
Sliding2D() (in module pylops.signalprocessing), 188
Sliding3D() (in module pylops.signalprocessing), 189
slope_estimate() (in module pylops.signalprocessing), 237
Smoothing1D() (in module pylops), 162
Smoothing2D() (in module pylops), 163
solve() (pylops.waveeqprocessing.LSM method), 227
SPGL1() (in module pylops.optimization.sparsity), 213
SplitBregman() (in module pylops.optimization.sparsity), 215
Spread (class in pylops), 155
Sum (class in pylops), 146
Symmetrize (class in pylops), 148
T
taper2d() (in module pylops.utils.tapers), 239
taper3d() (in module pylops.utils.tapers), 239
todense() (pylops.LinearOperator method), 132
tosparsel() (pylops.LinearOperator method), 132
Transpose (class in pylops), 142
U
UpDownComposition2D() (in module pylops.waveeqprocessing), 193
UpDownComposition3D() (in module pylops.waveeqprocessing), 194
V
VStack (class in pylops), 156
W
WavefieldDecomposition() (in module pylops.waveeqprocessing), 220
Z

Zero (class in pylons), 140
zoeppritz_element() (in module pylons.avosavo), 242
zoeppritz_pp() (in module pylons.avosavo), 242
zoeppritz_scattering() (in module pylons.avosavo), 241