
PyLops

May 02, 2022

Getting started:

1	Terminology	3
2	Implementation	5
3	History	7
	Bibliography	463
	Index	465

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:

```
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 = Dx
y = np.dot(D, x)
# x = D'y
xadj = np.dot(D.T, y)
# xinv = D^-1 y
xinv = lstsq(D, y)[0]
```

and similarly using PyLops commands:

```
from pylops import FirstDerivative

nx = 7
x = range(-(nx // 2), nx // 2 + (1 if nx % 2 else 0))

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

# y = Dx
y = Dlop*x
# x = D'y
xadj = Dlop.H*y
# xinv = D^-1 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.

CHAPTER 1

Terminology

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)**

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

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

$$\mathbf{x} = \mathbf{A}^H \mathbf{y}$$

where \mathbf{x} is called *model* and \mathbf{y} is called *data*. The *operator* $\mathbf{A} : \mathbb{F}^m \rightarrow \mathbb{F}^n$ 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* $\mathbf{A}^H : \mathbb{F}^n \rightarrow \mathbb{F}^m$ 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 \mathbf{A} from the data \mathbf{y} to retrieve the model \mathbf{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](#).

Implementation

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 $\mathbf{y} = \mathbf{Ax}$.

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 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. Since June 2021, PyLops is a [NUMFOCUS](#) Affiliated Project.

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 standar installation (see

below for details)

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:

```
>> pip install pylops
```

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

```
>> 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:

```
>> pip install git+https://git@github.com/PyLops/pylops.git@master
```

or just clone the repository

```
>> git clone https://github.com/PyLops/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:

```
>> 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):

```
>> 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:

```
>> git clone https://github.com/your_name_here/pylops.git
```

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

```
>> make dev-install
```

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

```
>> make dev-install_conda
```

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

```
>> make tests
```

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

Finally, to ensure consistency in the coding style of our developers we rely on [pre-commit](#) to perform a series of checks when you are ready to commit and push some changes. This is accomplished by means of git hooks that have been configured in the `.pre-commit-config.yaml` file.

In order to setup such hooks in your local repository, run:

```
>> pre-commit install
```

Later on, `pre-commit` will automatically run for you and propose additional stylistic changes to your commits.

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

```
>> 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`:

```
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:

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

and run the following code in 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):

```
>> 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:

```
>> 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:

```
>> pip install pylops[advanced]
```

Dependencies are however installed from their PyPI wheels.

An exception is however represented by `cupy`. This library is NOT installed automatically. Users interested to accelerate their computations with the aid of GPUs should install it prior to installing `pylops` (see below for more details).

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 traveltimes 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.

spgl1

This library is used to solve sparsity-promoting BP, BPDN, and LASSO problems in `pylops.optimization.sparsity.SPGL1` solver.

If interested to use `spgl1`, you can manually install it:

```
>> pip install spgl1
```

pywt

This library is used to implement the Wavelet operators.

If interested to use `pywt`, you can manually install it:

```
>> conda install pywavelets
```

or pip equivalent.

sympy

This library is used to implement the `describe` method, which transforms PyLops operators into their mathematical expression.

If interested to use `sympy`, you can manually install it:

```
>> conda install sympy
```

or pip equivalent.

Note: If you are a developer, all the above optional dependencies can also be installed automatically by cloning the repository and installing `pylops` via `make dev-install` or `make dev-install_conda`.

3.2.3 Optional Dependencies (GPU)

cupy

This library is used as a drop-in replacement to `numpy` for GPU-accelerated computations. Since many different versions of `cupy` exist (based on the CUDA drivers of the GPU), users must install `cupy` prior to installing `pylops`. PyLops will automatically check if `cupy` is installed and in that case use it any time the input vector passed to an operator is of `cupy` type. Users can however disable this option even if `cupy` is installed. For more details of GPU-accelerated PyLops read the [GPU support](#) section.

cusignal

This library is used as a drop-in replacement to `scipy.signal` for GPU-accelerated computations. Similar to `cupy`, users must install `cusignal` prior to installing `pylops`. PyLops will automatically check if `cusignal` is installed and in that case use it any time the input vector passed to an operator is of `cusignal` type. Users can however disable this option even if `cupy` is installed. For more details of GPU-accelerated PyLops read the [GPU support](#) section.

3.3 GPU support

From v1.12.0, PyLops supports computations on GPUs powered by `cupy` and `cusignal`.

Note: For this to work, ensure to have installed `cupy-cudaXX>=8.1.0` and `cusignal>=0.16.0`. If neither `cupy` nor `cusignal` is installed, `pylops` will work just fine using its CPU backend. Note, however, that setting the environment variable `CUPY_PYLOPS` (or `CUSIGNAL_PYLOPS`) to 0 will force `pylops` not to use the `cupy` backend. This can be also used if a previous version of `cupy` or `cusignal` is installed in your system, otherwise you will get an error when importing `pylops`.

Apart from a few exceptions, all operators and solvers in PyLops can seamlessly work with `numpy` arrays on CPU as well as with `cupy` arrays on GPU. Users do simply need to consistently create operators and provide data vectors to the solvers - e.g., when using `pylops.MatrixMult` the input matrix must be a `cupy` array if the data provided to a solver is a `cupy` array.

`pylops.LinearOperator` methods that are currently not available for GPU computations are:

- `eigs`, `cond`, and `tosparse`, and `estimate_spectral_norm`

Operators that are currently not available for GPU computations are:

- `pylops.Spread`
- `pylops.signalprocessing.Radon2D`
- `pylops.signalprocessing.Radon3D`
- `pylops.signalprocessing.DWT`
- `pylops.signalprocessing.DWT2D`
- `pylops.signalprocessing.Seislet`
- `pylops.waveeqprocessing.Demigration`
- `pylops.waveeqprocessing.LSM`

Solvers that are currently not available for GPU computations are:

- `pylops.optimization.sparsity.SPGL1`

3.3.1 Example

Finally, let's briefly look at an example. First we write a code snippet using `numpy` arrays which PyLops will run on your CPU:

```
ny, nx = 400, 400
G = np.random.normal(0,1,(ny, nx)).astype(np.float32)
x = np.ones(nx, dtype=np.float32)

Gop = MatrixMult(G, dtype='float32')
y = Gop * x
xest = Gop / y
```

Now we write a code snippet using `cupy` arrays which PyLops will run on your GPU:

```
ny, nx = 400, 400
G = cp.random.normal(0,1,(ny, nx)).astype(np.float32)
x = cp.ones(nx, dtype=np.float32)

Gop = MatrixMult(G, dtype='float32')
y = Gop * x
xest = Gop / y
```

The code is almost unchanged apart from the fact that we now use `cupy` arrays, PyLops will figure this out! For more examples head over to these [notebooks](#).

3.4 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**. Moreover, some independent libraries are created to use third party software that cannot be included as dependencies to our main library for licensing issues but may be useful for academic purposes.

Spin-off projects that aim at extending the capabilities of PyLops are:

- [PyLops-GPU](#)
- [PyLops-Distributed](#)
- [PyProximal](#)
- [Curvelops](#)

3.5 Tutorials

3.5.1 01. The LinearOperator

This first tutorial 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.

```
import timeit

import matplotlib.pyplot as plt
import numpy as np

import pylops

n = 10
d = np.arange(n) + 1.0
x = np.ones(n)
Dop = pylops.Diagonal(d)
```

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.

```
# setup command
cmd_setup = """\
import numpy as np
import pylops
n = 10
d = np.arange(n) + 1.
x = np.ones(n)
Dop = pylops.Diagonal(d)
DopH = Dop.H
"""

# __matvec
cmd1 = "Dop.__matvec(x) "

# matvec
cmd2 = "Dop.matvec(x) "

# @
cmd3 = "Dop@x"

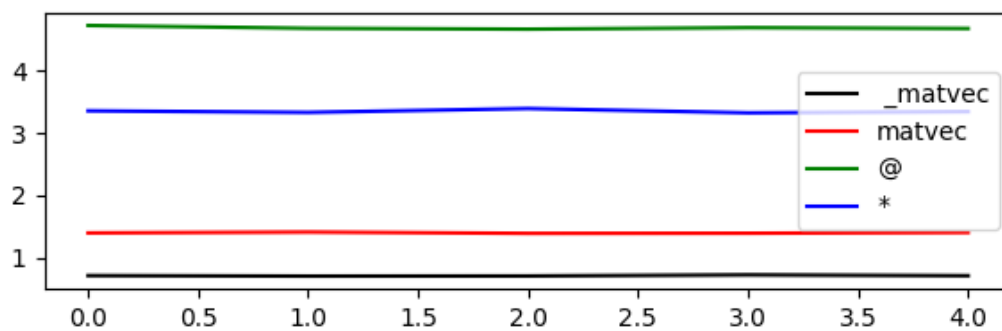
# *
cmd4 = "Dop*x"
```

(continues on next page)

(continued from previous page)

```
# timing
t1 = 1.0e3 * np.array(timeit.repeat(cmd1, setup=cmd_setup, number=500, repeat=5))
t2 = 1.0e3 * np.array(timeit.repeat(cmd2, setup=cmd_setup, number=500, repeat=5))
t3 = 1.0e3 * np.array(timeit.repeat(cmd3, setup=cmd_setup, number=500, repeat=5))
t4 = 1.0e3 * np.array(timeit.repeat(cmd4, setup=cmd_setup, number=500, repeat=5))

plt.figure(figsize=(7, 2))
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()
```



Out:

```
<matplotlib.legend.Legend object at 0x7f6ae9955cc0>
```

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 = "Dop.H*x"

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

# timing
```

(continues on next page)

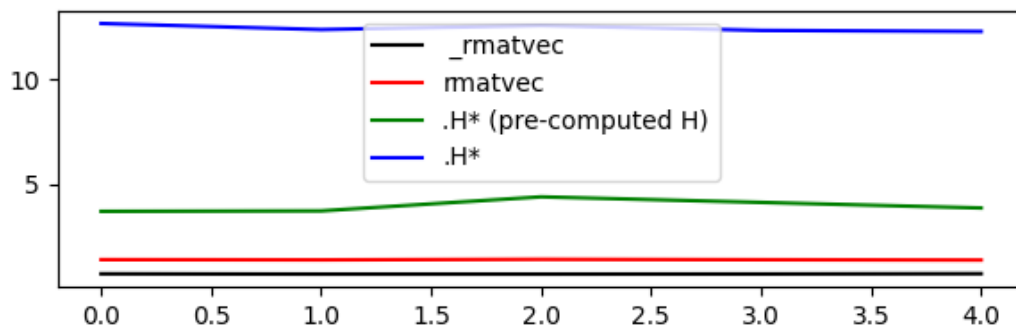
(continued from previous page)

```

t1 = 1.0e3 * np.array(timeit.repeat(cmd1, setup=cmd_setup, number=500, repeat=5))
t2 = 1.0e3 * np.array(timeit.repeat(cmd2, setup=cmd_setup, number=500, repeat=5))
t3 = 1.0e3 * np.array(timeit.repeat(cmd3, setup=cmd_setup, number=500, repeat=5))
t4 = 1.0e3 * 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 0x7f6aecda8cf8>
```

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)

```

(continues on next page)

(continued from previous page)

```
# -
print(-Dop)
print(Dop - 0.5 * Dop)

# **
print(Dop ** 3)

# * and /
y = Dop * x
print(Dop / y)

# eigs
print(Dop.eigs(neigs=3))

# cond
print(Dop.cond())

# conj
print(Dop.conj())
```

Out:

```
<10x10 LinearOperator with dtype=float64>
<10x10 LinearOperator with dtype=float64>
<10x10 LinearOperator with dtype=float64>
<10x10 LinearOperator with dtype=float64>
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
[10.+0.j 9.+0.j 8.+0.j]
(9.999999999999986+0j)
<10x10 _ConjLinearOperator with dtype=float64>
```

To understand the effect of `conj` we need to look into a problem with an operator in the complex domain. Let's create again our `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.

```
n = 5
d = 1j * (np.arange(n) + 1.0)
x = np.ones(n)
Dop = pylops.Diagonal(d)

print("y = Dx = ", Dop * x)
print("y = conj(D)x = ", Dop.conj() * x)
```

Out:

```
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 convenience method `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

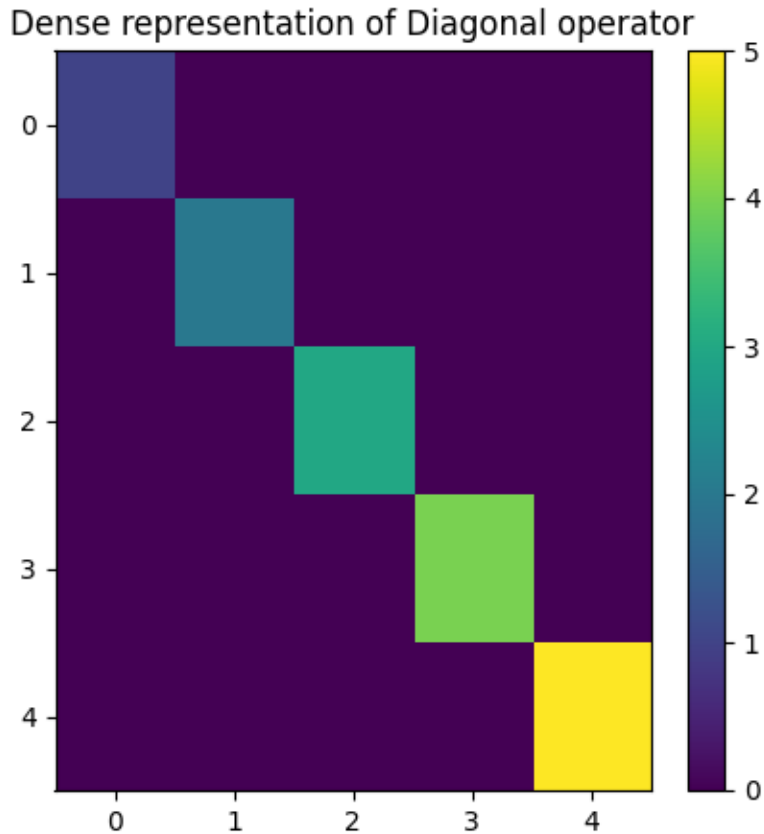
```
D = Dop.todense()

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

(continues on next page)

(continued from previous page)

```
plt.title("Dense representation of Diagonal operator")
plt.axis("tight")
plt.colorbar()
```



Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f6aec5117f0>
```

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 `/`

```
Dop1 = Dop - Dop.conj()

y = Dop1 * x
print("x = (Dop - conj(Dop))/y = ", Dop1 / y)

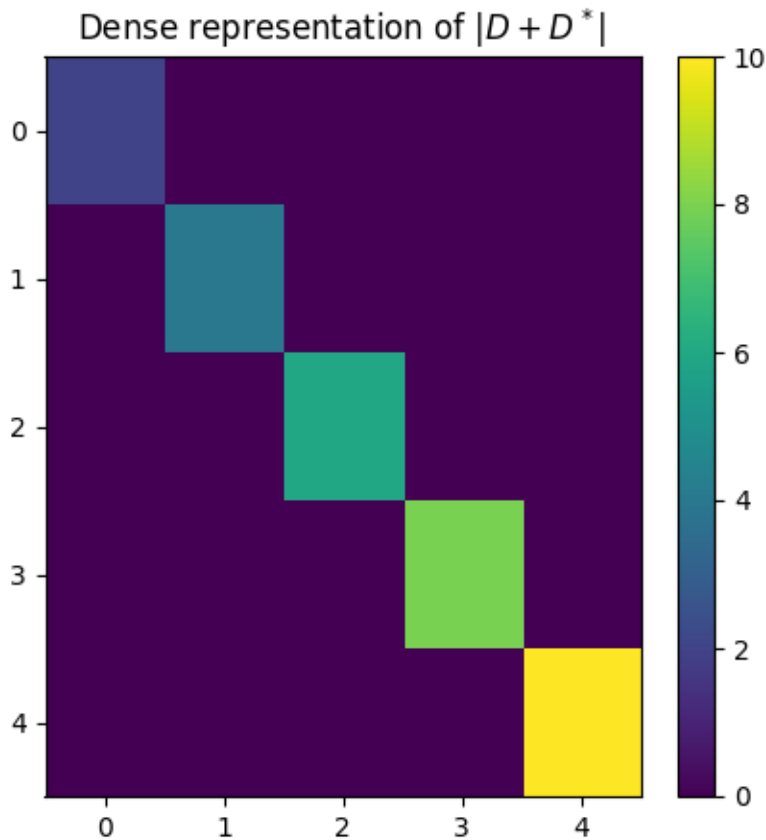
D1 = Dop1.todense()

plt.figure(figsize=(5, 5))
plt.imshow(np.abs(D1))
plt.title(r"Dense representation of  $|D + D^*|$ ")
plt.axis("tight")
```

(continues on next page)

(continued from previous page)

```
plt.colorbar()
```



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 0x7f6ae950a978>
```

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 0.636 seconds)

3.5.2 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.

```

import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt

# pylint: disable=C0103
import numpy as np

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.

```

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

```

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")
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])

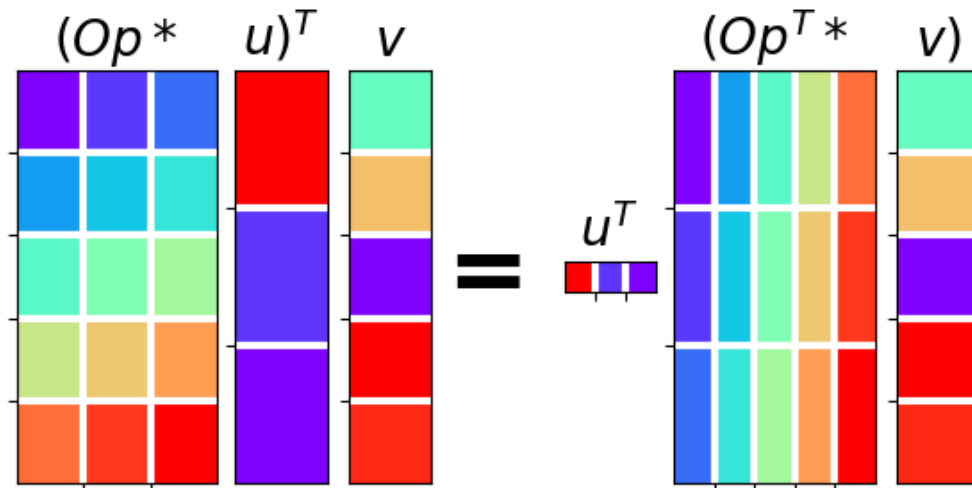
```

(continues on next page)

```

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([])

```



Out:

```
[Text(0, 0.5, ''), Text(0, 1.5, ''), Text(0, 2.5, ''), Text(0, 3.5, '')]

```

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 successful. 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 successful 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^H(Opu)=-6.91965436475742 - u^H(Op^Hv)=-6.91965436475742
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:

```
Dot test passed, v^H(Opu)=(-17.13910350422714+3.8288339959112507j) - u^H(Op^Hv)=(-17.
↪139103504227144+3.8288339959112503j)
Dot test passed, v^H(Opu)=(11.42080013111478-3.688151329246325j) - u^H(Op^Hv)=(11.
↪420800131114781-3.6881513292463266j)
```

```
True
```

Total running time of the script: (0 minutes 0.262 seconds)

3.5.3 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:

$$\mathbf{y} = \mathbf{R}\mathbf{x}$$

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

$$\mathbf{y} = [y_1, y_2, \dots, y_N]^T, \quad \mathbf{x} = [x_1, x_2, \dots, x_M]^T,$$

with $M \gg N$.

```
import matplotlib.pyplot as plt

# pylint: disable=C0103
import numpy as np

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.

```
# Signal creation in frequency domain
ifreqs = [41, 25, 66]
amps = [1.0, 1.0, 1.0]
N = 200
nfft = 2 ** 11
dt = 0.004
t = np.arange(N) * dt
f = np.fft.rfftfreq(nfft, dt)

FFTop = 10 * pylops.signalprocessing.FFT(N, nfft=nfft, real=True)
```

(continues on next page)

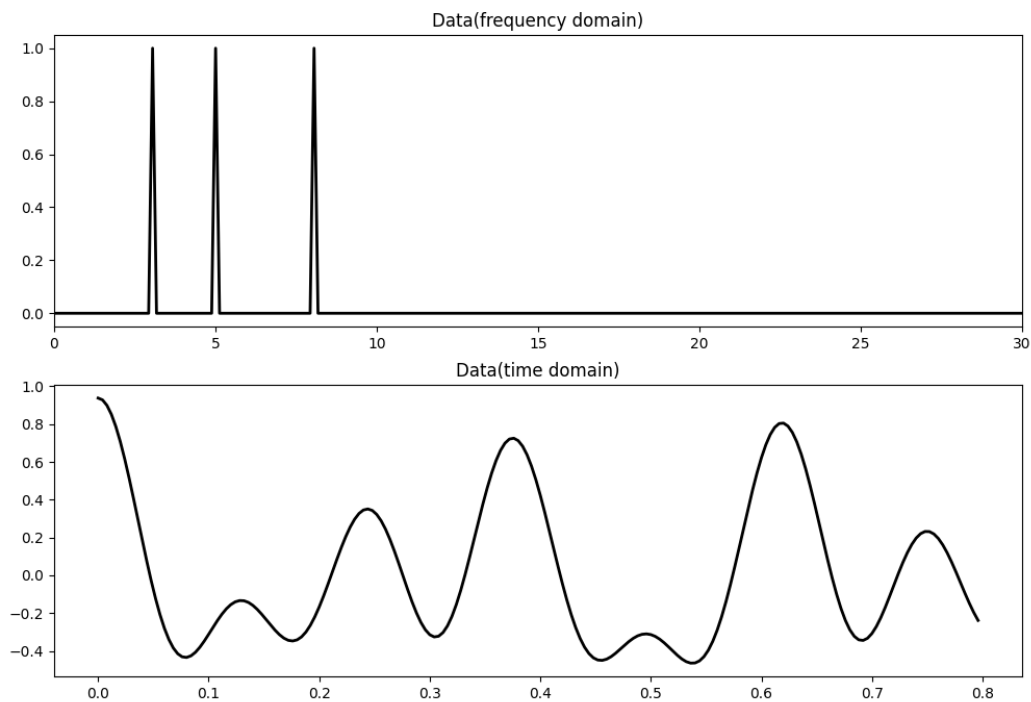
(continued from previous page)

```

X = np.zeros(nfft // 2 + 1, dtype="complex128")
X[ifreqs] = amps
x = FFTop.H * X

fig, axs = plt.subplots(2, 1, figsize=(12, 8))
axs[0].plot(f, np.abs(X), "k", lw=2)
axs[0].set_xlim(0, 30)
axs[0].set_title("Data(frequency domain)")
axs[1].plot(t, x, "k", lw=2)
axs[1].set_title("Data(time domain)")
axs[1].axis("tight")

```



Out:

```
(-0.0398, 0.8358000000000001, -0.5341666861485157, 1.007579366007072)
```

We now define the locations at which the signal will be sampled.

```

# 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")

```

(continues on next page)

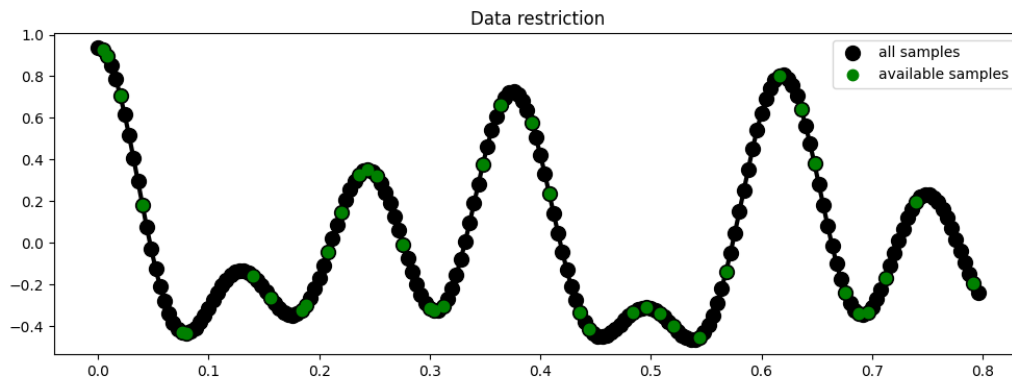
(continued from previous page)

```

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, x, ".k", ms=20, label="all samples")
plt.plot(t, ymask, ".g", ms=15, label="available samples")
plt.legend()
plt.title("Data restriction")

```



Out:

```
Text(0.5, 1.0, '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:

$$J = \|y - \mathbf{R}x\|_2^2$$

Depending on the choice of the operator \mathbf{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 *LSQR* solver - i.e., `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 `scipy.sparse.linalg.lsqr` with some default parameters.

```
xinv = Rop / y
```

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

```

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 = 0.0000000000000000e+00   calc_var =          0
atol = 1.00e-08                 conlim = 1.00e+08
btol = 1.00e-08                 iter_lim =         10

   Itn      x[0]      rlnorm      r2norm  Compatible    LS      Norm A    Cond A
   --  -----  -
   0  0.00000e+00  2.658e+00  2.658e+00    1.0e+00  3.8e-01
   1  0.00000e+00  0.000e+00  0.000e+00    0.0e+00  0.0e+00  0.0e+00  0.0e+00

LSQR finished
Ax - b is small enough, given atol, btol

istop =          1   rlnorm = 0.0e+00   anorm = 0.0e+00   arnorm = 0.0e+00
itn    =          1   r2norm = 0.0e+00   acond = 0.0e+00   xnorm  = 2.7e+00

```

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

```

xinv_fromx0 = pyllops.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 *normal equations*

$$\mathbf{x}_{ne} = (\mathbf{R}^T \mathbf{R})^{-1} \mathbf{R}^T \mathbf{y}$$

The method `pyllops.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.

```

xne = pyllops.optimization.leastsquares.NormalEquationsInversion(Rop, [], y)

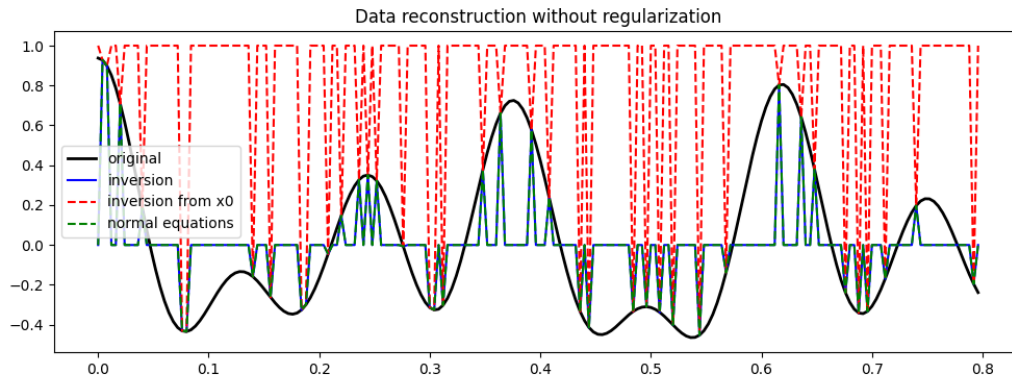
```

Let's now visualize the different inversion results

```

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

- regularization via `pylops.optimization.leastsquares.NormalEquationsInversion` or `pylops.optimization.leastsquares.RegularizedInversion`
- preconditioning via `pylops.optimization.leastsquares.PreconditionedInversion`

Let's start by regularizing the normal equations using a second derivative operator

$$\mathbf{x} = (\mathbf{R}^T \mathbf{R} + \epsilon_{\nabla}^2 \nabla^T \nabla)^{-1} \mathbf{R}^T \mathbf{y}$$

```
# 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:

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

xne1 = pylops.optimization.leastsquares.NormalEquationsInversion(
    Rop,
```

(continues on next page)

(continued from previous page)

```

[],
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

$$\begin{bmatrix} \mathbf{R} \\ \epsilon \nabla \nabla \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{y} \\ 0 \end{bmatrix}$$

```

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

$$J = \|\mathbf{y} - \mathbf{R}\mathbf{P}\mathbf{p}\|_2^2$$

where \mathbf{P} is the preconditioned operator, \mathbf{p} is the projected model in the preconditioned space, and $\mathbf{x} = \mathbf{P}\mathbf{p}$ 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`.

```

# 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), iter_lim=20, show=0)
)

```

Let's finally visualize these solutions

```

# sphinx_gallery_thumbnail_number=4
fig = plt.figure(figsize=(12, 4))
plt.plot(t[iava], y, ".k", ms=20, label="available samples")
plt.plot(t, x, "k", lw=3, label="original")
plt.plot(t, xne, "b", lw=3, label="normal equations")
plt.plot(t, xne1, "--c", lw=3, label="normal equations (with direct D^T D)")
plt.plot(t, xreg, "-.r", lw=3, label="regularized")
plt.plot(t, xprec, "--g", lw=3, label="preconditioned equations")
plt.legend()

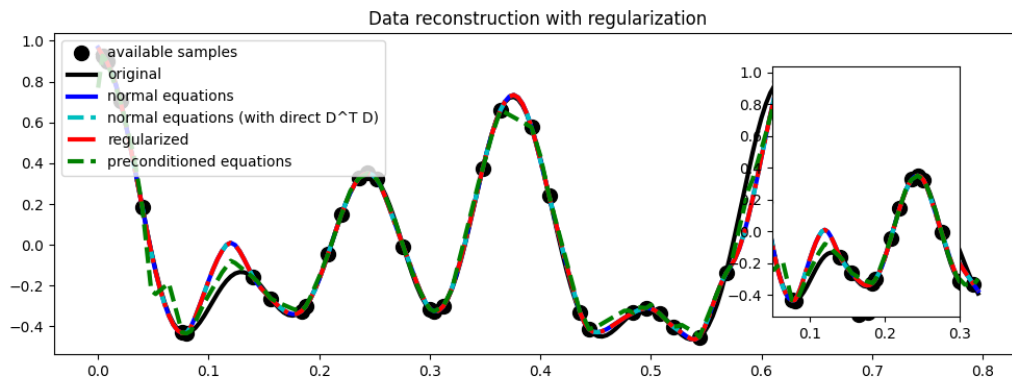
```

(continues on next page)

(continued from previous page)

```
plt.title("Data reconstruction with regularization")

subax = fig.add_axes([0.7, 0.2, 0.15, 0.6])
subax.plot(t[iava], y, ".k", ms=20)
subax.plot(t, x, "k", lw=3)
subax.plot(t, xne, "b", lw=3)
subax.plot(t, xnel, "--c", lw=3)
subax.plot(t, xreg, "-.r", lw=3)
subax.plot(t, xprec, "--g", lw=3)
subax.set_xlim(0.05, 0.3)
```



Out:

(0.05, 0.3)

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 recasted 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:

$$J_1 = \|\mathbf{y} - \mathbf{R}\mathbf{F}\mathbf{p}\|_2^2 + \epsilon \|\mathbf{p}\|_1$$

where \mathbf{F} is the FFT operator. We will thus use the `pylops.optimization.sparsity.ISTA` and `pylops.optimization.sparsity.FISTA` solvers to estimate our input signal.

```
pista, niteri, costi = pylops.optimization.sparsity.ISTA(
    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
```

(continues on next page)

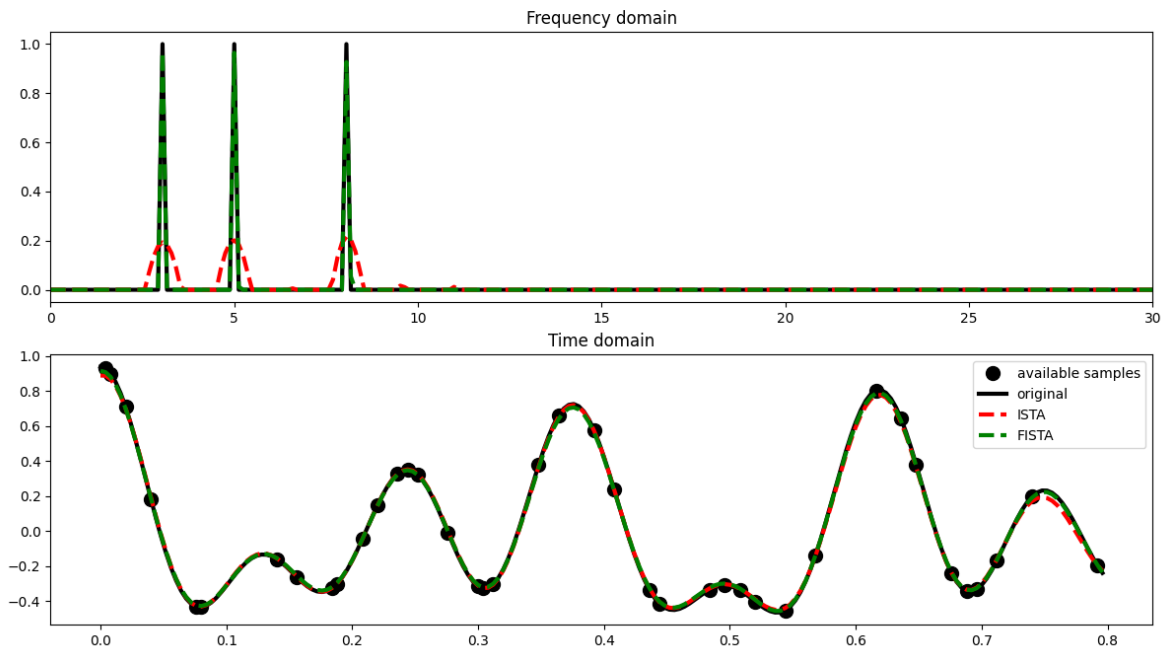
(continued from previous page)

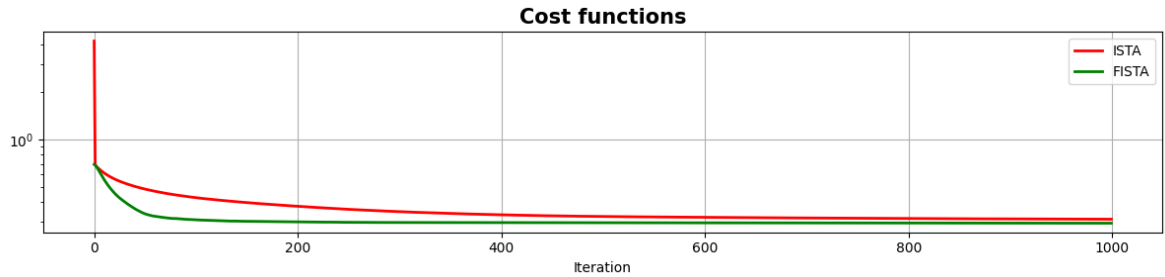
```

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 sparsity



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):

$$J_1 = \|\mathbf{p}\|_1 \quad \text{subject to} \quad \|\mathbf{y} - \mathbf{R}\mathbf{F}\mathbf{p}\|$$

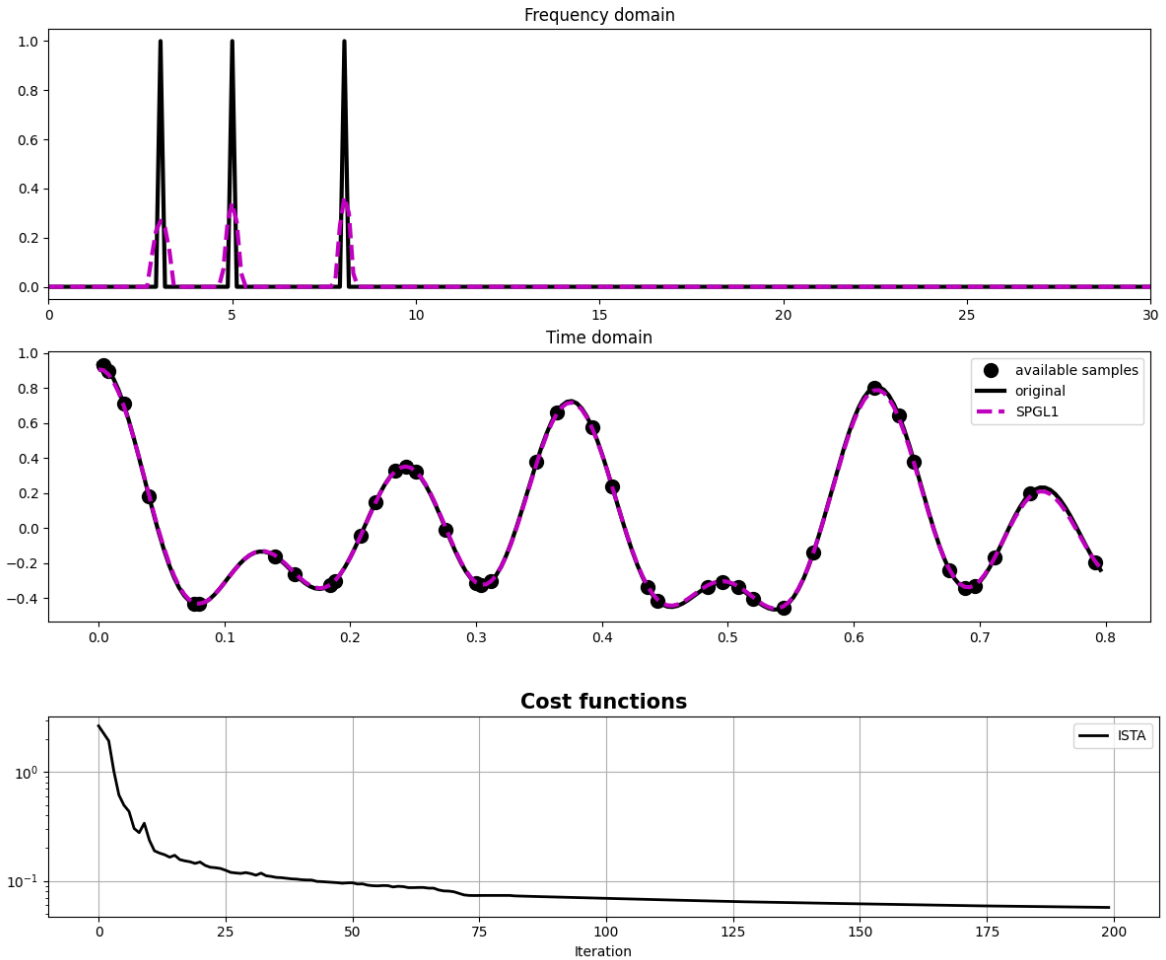
A very popular solver to solve such kind of cost function is called *spgll* and can be accessed via `pylops.optimization.sparsity.SPGL1`.

```
xspgll, pspgll, info = pylops.optimization.sparsity.SPGL1(
    Rop, y, FFTop, tau=3, iter_lim=200
)

fig, axs = plt.subplots(2, 1, figsize=(12, 8))
fig.suptitle("Data reconstruction with SPGL1", fontsize=14, fontweight="bold", y=0.9)
axs[0].plot(f, np.abs(X), "k", lw=3)
axs[0].plot(f, np.abs(pspgll), "--m", 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, xspgll, "--m", lw=3, label="SPGL1")
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(info["rnorm2"], "k", lw=2, label="ISTA")
ax.set_title("Cost functions", size=15, fontweight="bold")
ax.set_xlabel("Iteration")
ax.legend()
ax.grid(True)
plt.tight_layout()
```


Data reconstruction with SPGL1



Total running time of the script: (0 minutes 3.923 seconds)

3.5.4 04. Bayesian Inversion

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 $\mu_{\mathbf{x}}$ and model covariance $\mathbf{C}_{\mathbf{x}}$.

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:

$$\mathbf{x} = \mathbf{x}_0 + \mathbf{C}_x \mathbf{R}^T (\mathbf{R} \mathbf{C}_x \mathbf{R}^T + \mathbf{C}_y)^{-1} (\mathbf{y} - \mathbf{R} \mathbf{x}_0)$$

```
import matplotlib.pyplot as plt

# sphinx_gallery_thumbnail_number = 2
import numpy as np
from scipy.sparse.linalg import lsqr

import pylops

plt.close("all")
np.random.seed(10)
```

Let's start by creating our true model and prior realizations

```
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() * 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, 0.6]
a0 = [1.0, 1.0, 1.0]
sigmaa = [0.1, 0.5, 0.6]
phi0 = [-90.0, 0.0, 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(
```

(continues on next page)

(continued from previous page)

```

[
    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

```

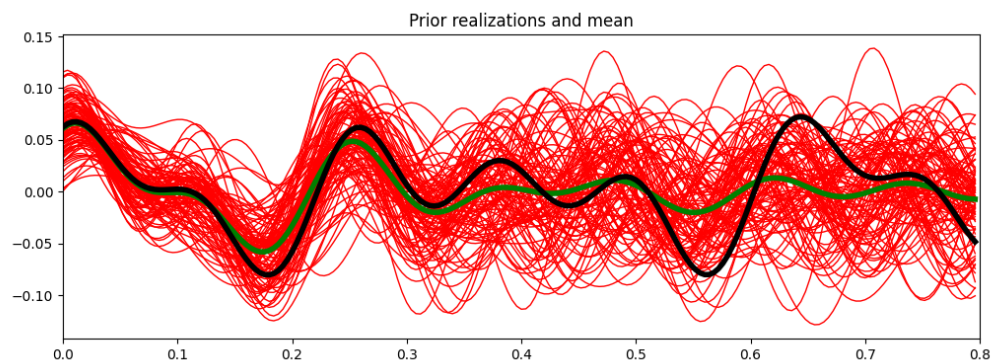
x0 = np.average(xs, axis=0)
Cm = ((xs - x0).T @ (xs - x0)) / nreals

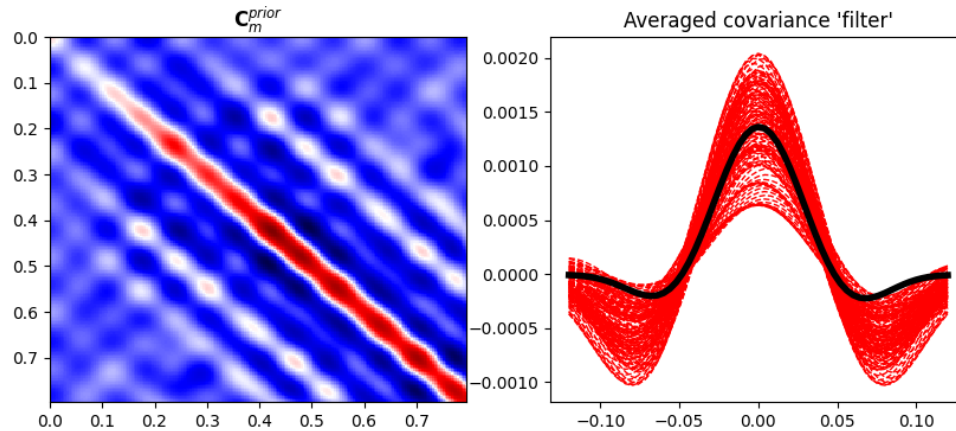
N = 30 # lenght of decorrelation
diags = np.array([Cm[i, i - N : i + N + 1] for i in range(N, nt - N)])
diag_ave = np.average(diags, axis=0)
# add a taper at the end to avoid edge effects
diag_ave *= np.hamming(2 * N + 1)

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"$\mathbf{C}_m^{\text{prior}}$")
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'")

```





Out:

```
Text(0.5, 1.0, "Averaged covariance 'filter'")
```

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.

```
# 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

```
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

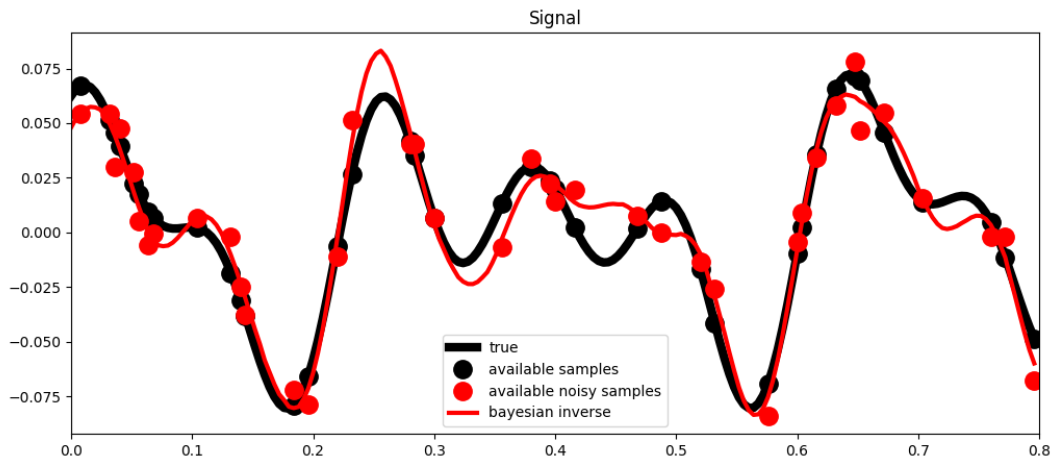
```
xbayes = x0 + Cm_op * Rop.H * (
    lsqr(Rop * Cm_op * Rop.H + Cd_op, yn - Rop * x0, iter_lim=400)[0]
)

# Visualize
fig, ax = plt.subplots(1, 1, figsize=(12, 5))
ax.plot(t, x, "k", lw=6, label="true")
ax.plot(t, ymask, ".k", ms=25, label="available samples")
ax.plot(t, ynmask, ".r", ms=25, label="available noisy samples")
ax.plot(t, xbayes, "r", lw=3, label="bayesian inverse")
ax.legend()
```

(continues on next page)

(continued from previous page)

```
ax.set_title("Signal")
ax.set_xlim(0, 0.8)
```



Out:

```
(0.0, 0.8)
```

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:

```
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(
```

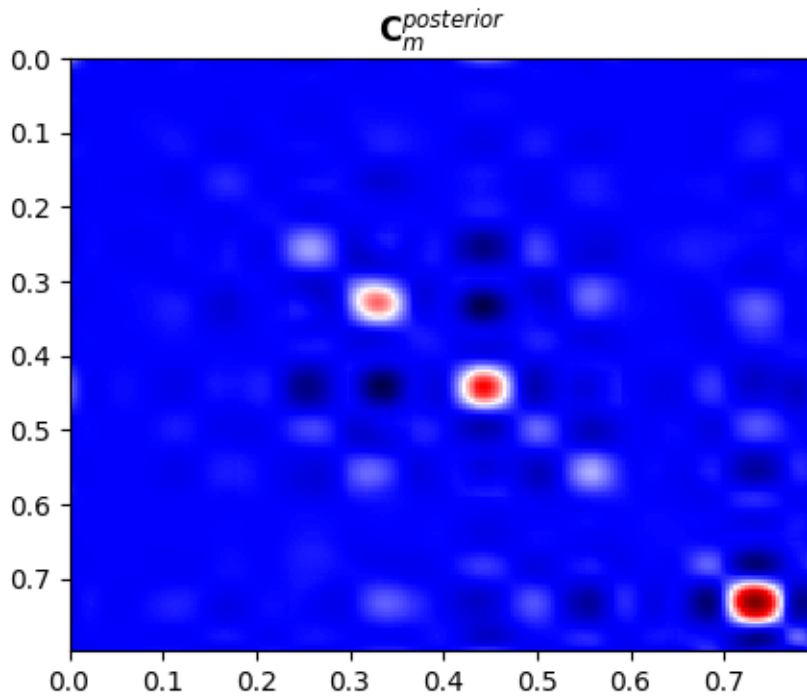
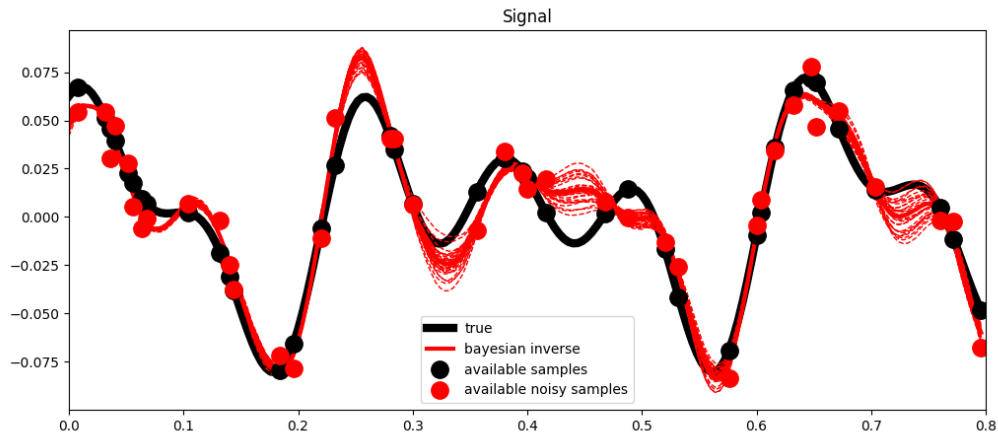
(continues on next page)

(continued from previous page)

```

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")

```



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 1.549 seconds)

3.5.5 05. 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.

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

import pylops
```

Let's start by importing a 2d image and defining the blurring operator

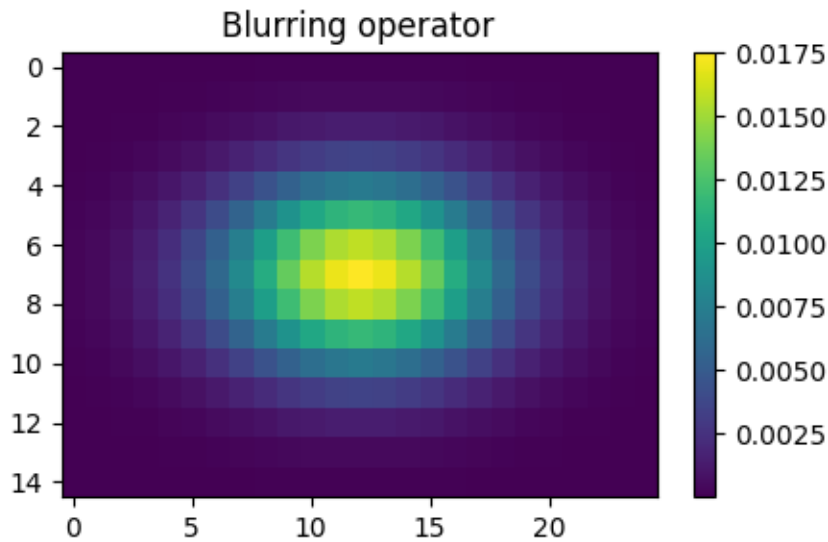
```
im = np.load("../testdata/python.npy")[:,5, :5, 0]

Nz, Nx = im.shape

# Blurring gaussian 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.ravel()

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

imdeblurtv = pylops.optimization.sparsity.SplitBregman(
    Cop,
    Dop,
    imblur.ravel(),
    niter_outer=10,
    niter_inner=5,
    mu=1.5,
    epsRL1s=[2e0, 2e0],
    tol=1e-4,
    tau=1.0,
    show=False,
    **dict(iter_lim=5, damp=1e-4)
```

(continues on next page)

(continued from previous page)

```

) [0]

imdeblurtv1 = pylops.optimization.sparsity.SplitBregman(
    Cop,
    DWop,
    imblur.ravel(),
    niter_outer=10,
    niter_inner=5,
    mu=1.5,
    epsRLls=[1e0, 1e0, 1e0],
    tol=1e-4,
    tau=1.0,
    show=False,
    **dict(iter_lim=5, damp=1e-4)
) [0]

# Reshape images
imblur = imblur.reshape((Nz, Nx))
imdeblur = imdeblur.reshape((Nz, Nx))
imdeblurfista = imdeblurfista.reshape((Nz, Nx))
imdeblurtv = imdeblurtv.reshape((Nz, Nx))
imdeblurtv1 = imdeblurtv1.reshape((Nz, Nx))

```

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

```

# sphinx_gallery_thumbnail_number = 2
fig = plt.figure(figsize=(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), (1, 0))
ax5 = plt.subplot2grid((2, 5), (1, 1))
ax6 = plt.subplot2grid((2, 5), (1, 2))
ax7 = plt.subplot2grid((2, 5), (0, 3), colspan=2)
ax8 = plt.subplot2grid((2, 5), (1, 3), colspan=2)
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+Haar deblurred")
ax7.plot(im[Nz // 2], "k")
ax7.plot(imblur[Nz // 2], "--r")
ax7.plot(imdeblur[Nz // 2], "--b")

```

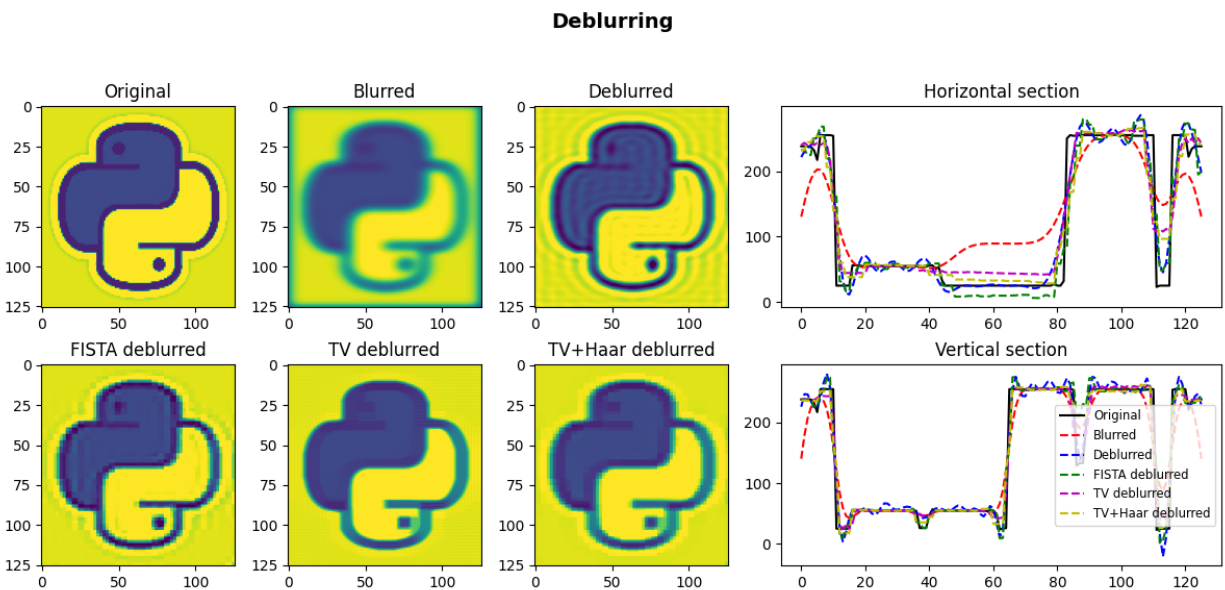
(continues on next page)

(continued from previous page)

```

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)

```



Total running time of the script: (0 minutes 7.216 seconds)

3.5.6 06. 2D Interpolation

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:

$$\mathbf{y} = \mathbf{R}\mathbf{x}$$

where the restriction operator \mathbf{R} selects M elements from the regularly sampled signal \mathbf{x} at random locations. The

input and output signals are:

$$\mathbf{y} = [y_1, y_2, \dots, y_N]^T, \quad \mathbf{x} = [x_1, x_2, \dots, x_M]^T,$$

with $M \gg N$.

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

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

```
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.ravel()
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.

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

# Invert for interpolated signal, lsqr
(
    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)
)
```

(continues on next page)

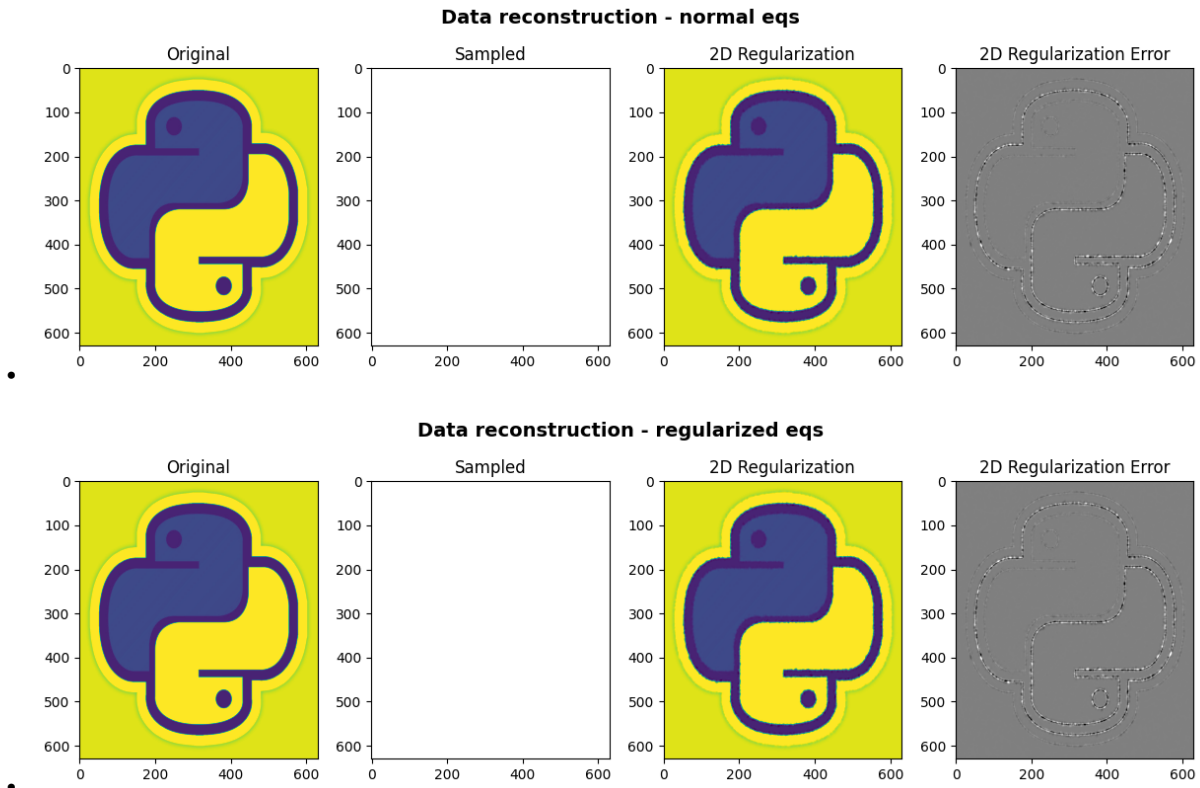
(continued from previous page)

```
# 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

```
fig, axs = plt.subplots(1, 4, figsize=(12, 4))
fig.suptitle("Data reconstruction - normal eqs", fontsize=14, fontweight="bold", y=0.
↪95)
axs[0].imshow(im, cmap="viridis", vmin=0, vmax=250)
axs[0].axis("tight")
axs[0].set_title("Original")
axs[1].imshow(im_sampled, cmap="viridis", vmin=0, vmax=250)
axs[1].axis("tight")
axs[1].set_title("Sampled")
axs[2].imshow(im_rec_lap_cg, cmap="viridis", vmin=0, vmax=250)
axs[2].axis("tight")
axs[2].set_title("2D Regularization")
axs[3].imshow(im - im_rec_lap_cg, cmap="gray", vmin=-80, vmax=80)
axs[3].axis("tight")
axs[3].set_title("2D Regularization Error")
plt.tight_layout()
plt.subplots_adjust(top=0.8)

fig, axs = plt.subplots(1, 4, figsize=(12, 4))
fig.suptitle(
    "Data reconstruction - regularized eqs", fontsize=14, fontweight="bold", y=0.95
)
axs[0].imshow(im, cmap="viridis", vmin=0, vmax=250)
axs[0].axis("tight")
axs[0].set_title("Original")
axs[1].imshow(im_sampled, cmap="viridis", vmin=0, vmax=250)
axs[1].axis("tight")
axs[1].set_title("Sampled")
axs[2].imshow(im_rec_lap_lsqr, cmap="viridis", vmin=0, vmax=250)
axs[2].axis("tight")
axs[2].set_title("2D Regularization")
axs[3].imshow(im - im_rec_lap_lsqr, cmap="gray", vmin=-80, vmax=80)
axs[3].axis("tight")
axs[3].set_title("2D Regularization Error")
plt.tight_layout()
plt.subplots_adjust(top=0.8)
```



Total running time of the script: (0 minutes 15.942 seconds)

3.5.7 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:

$$\mathbf{d} = \mathbf{W} \mathbf{D} \mathbf{ai}$$

where \mathbf{W} is a convolution operator, \mathbf{D} is a first derivative operator, and \mathbf{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.

```
import matplotlib.pyplot as plt

# sphinx_gallery_thumbnail_number = 4
import numpy as np
from scipy.signal import filtfilt

import pylops
from pylops.utils.wavelets import ricker
```

(continues on next page)

(continued from previous page)

```
plt.close("all")
np.random.seed(10)
```

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.poststack.PoststackLinearModelling` operator.

```
# model
nt0 = 301
dt0 = 0.004
t0 = np.arange(nt0) * dt0
vp = 1200 + np.arange(nt0) + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 80,
↪nt0))
rho = 1000 + vp + filtfilt(np.ones(5) / 5.0, 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
PPop = pylops.avo.poststack.PoststackLinearModelling(wav / 2, nt0=nt0)

# data
d_dense = PPop_dense * m.ravel()
d = PPop * m

# add noise
dn_dense = d_dense + np.random.normal(0, 2e-2, d_dense.shape)
```

We can now estimate the acoustic profile from band-limited data using either the dense operator or linear operator.

```
# 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)
```

(continues on next page)

(continued from previous page)

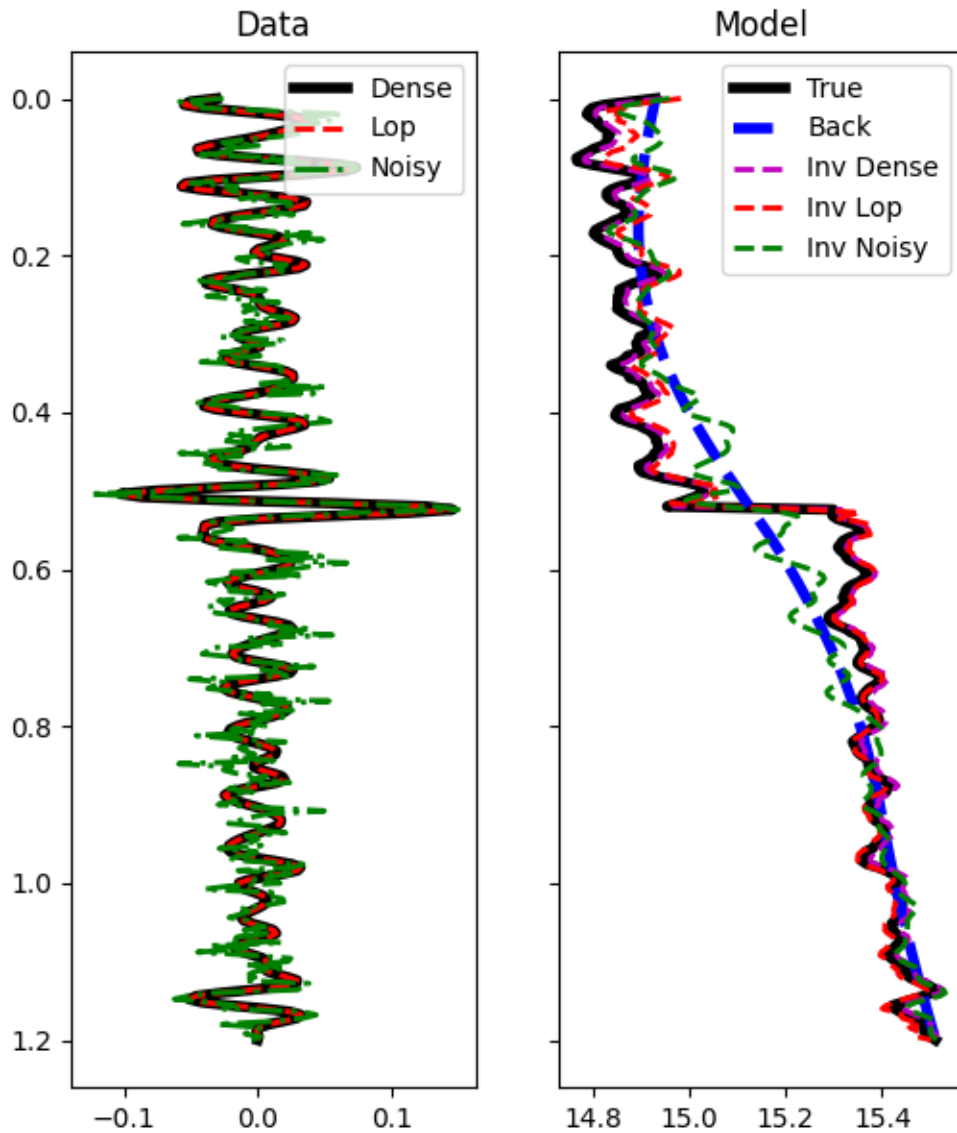
```

)[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)

```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aeda29630>
```

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


```

# 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")

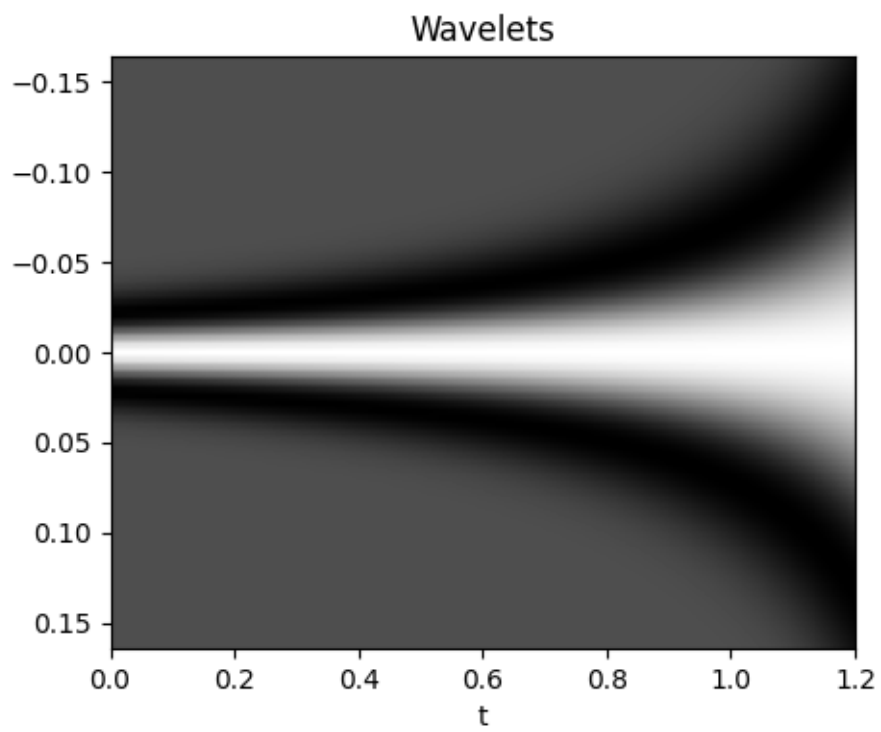
# operator
PPop = pylops.avo.poststack.PoststackLinearModelling(wavs / 2, nt0=nt0, explicit=True)

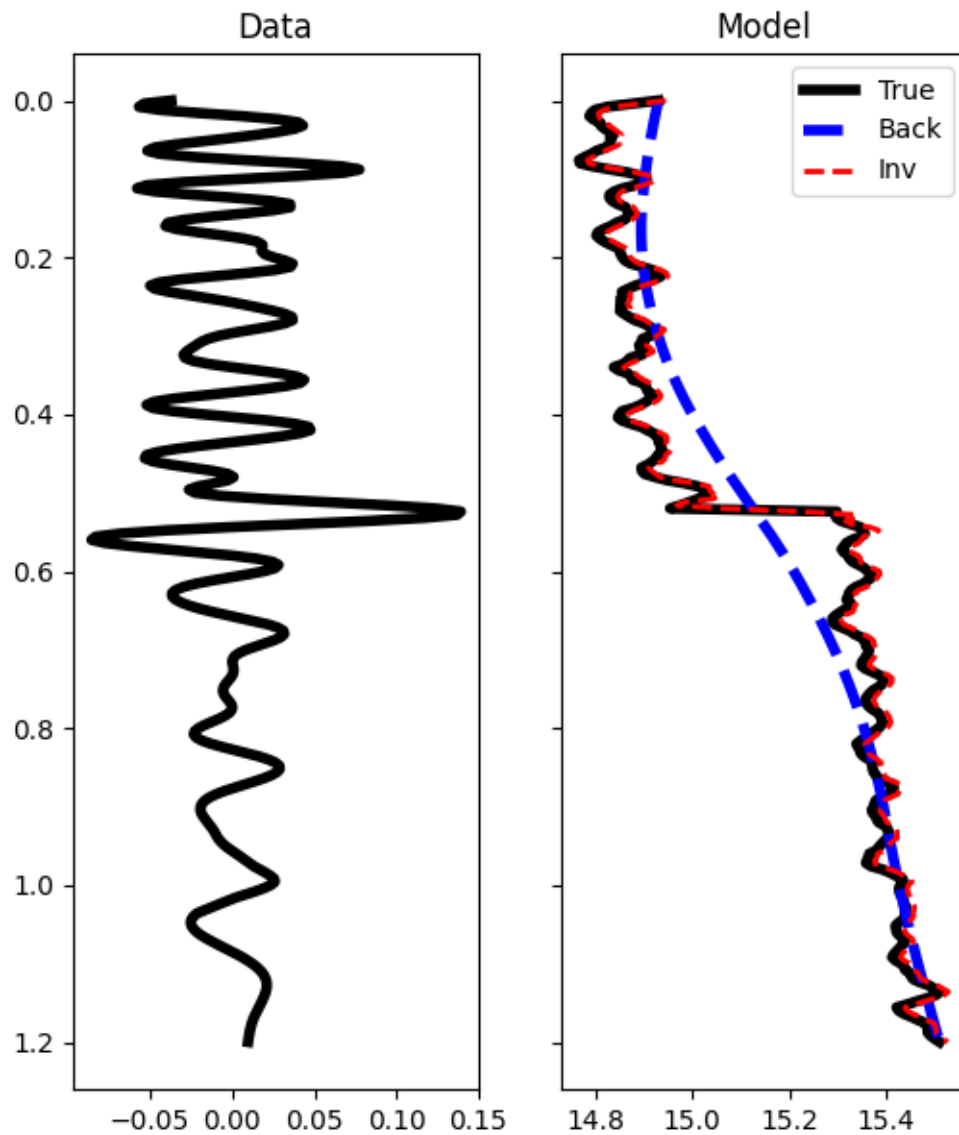
# data
d = PPop * m

# 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)

```





Out:

```
<matplotlib.legend.Legend object at 0x7f6ae94df400>
```

We move now to a 2d example. First of all the model is loaded and data generated.

```
# model
inputfile = "../testdata/avo/poststack_model.npz"

model = np.load(inputfile)
m = np.log(model["model"][:, :, :3])
```

(continues on next page)

(continued from previous page)

```

x, z = model["x"][:, :3] / 1000.0, model["z"] / 1000.0
nx, nz = len(x), len(z)

# smooth model
nsmoothz, nsmoothx = 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.avopoststack.PoststackLinearModelling(
    wav / 2, nt0=nz, spatdims=nx, explicit=True
)

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

# data
d = (PPop_dense * m.ravel()).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

$$J = \|\Delta \mathbf{d} - \mathbf{W} \Delta \mathbf{a}\|_2 + \epsilon_{\nabla}^2 \|\nabla \mathbf{a}\|_2$$

where $\Delta \mathbf{d} = \mathbf{d} - \mathbf{W} \mathbf{a} \mathbf{I}_0$ is the residual data

- multi-trace blocky inversion with iterative solver and linear operator

```

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

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

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

# blockiness promoting inversion with noisy data
minv_lop_blocky = pylops.avopoststack.PoststackInversion(

```

(continues on next page)

(continued from previous page)

```

    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]

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,

```

(continues on next page)

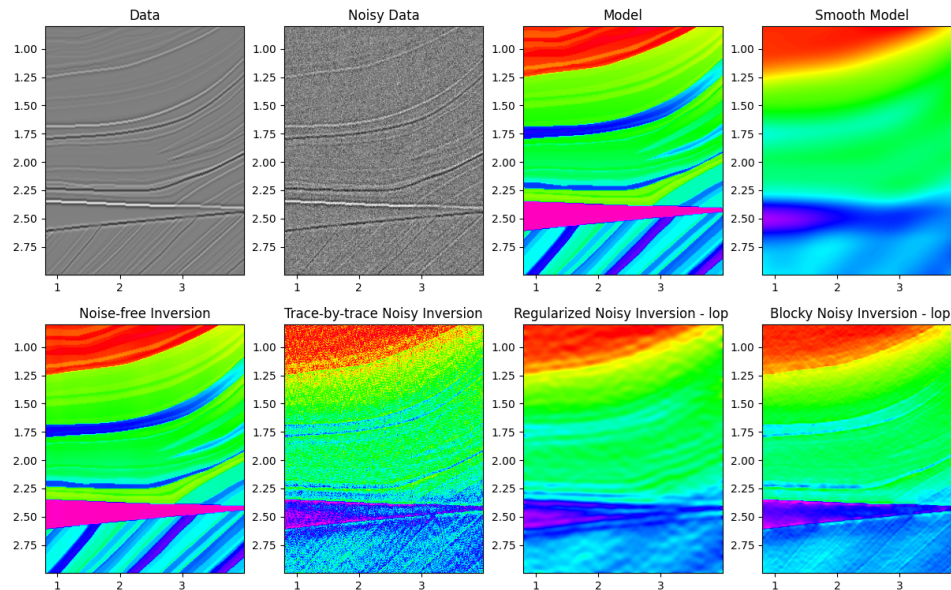
(continued from previous page)

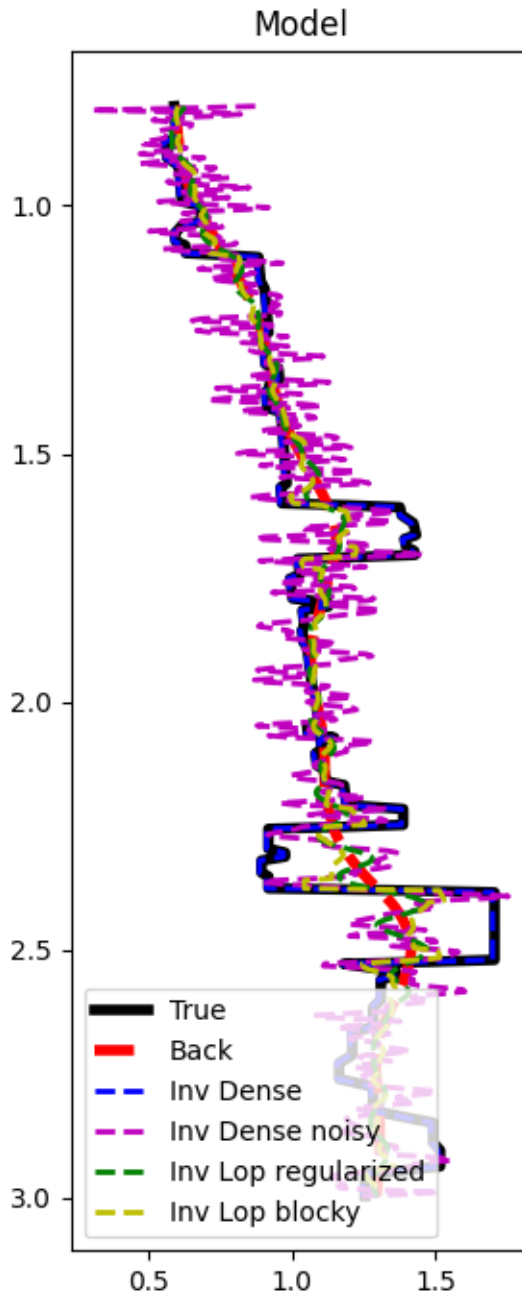
```

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")
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 [segymio](#) 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 13.668 seconds)

3.5.8 08. Pre-stack (AVO) inversion

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 $\mathbf{m}(t) = [V_P(t), V_S(t), \rho(t)]$ 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:

$$\mathbf{d} = \mathbf{WGDm}$$

where \mathbf{W} is a convolution operator, \mathbf{G} is the AVO modelling operator, \mathbf{D} is a block-diagonal derivative operator, and \mathbf{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.

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

```
# sphinx_gallery_thumbnail_number = 5

# model
nt0 = 301
dt0 = 0.004

t0 = np.arange(nt0) * dt0
vp = 1200 + np.arange(nt0) + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 80, nt0))
vs = 600 + vp / 2 + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 20, nt0))
rho = 1000 + vp + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 30, nt0))
vp[131:] += 500
vs[131:] += 200
rho[131:] += 100
vsvp = 0.5
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
```

(continues on next page)

(continued from previous page)

```

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.ravel()
dPP = dPP.reshape(nt0, ntheta)

# data dense
dPP_dense = PPop_dense * m.T.ravel()
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)
)

# dense noisy

```

(continues on next page)

(continued from previous page)

```

minv_dense_noise, dPPn_dense_res = pylops.avo.prestack.PrestackInversion(
    dPPn_dense,
    theta,
    wav,
    m0=mback,
    linearization="akirich",
    explicit=True,
    returnres=True,
    **dict(cond=1e-1)
)

# lop noisy (with vertical smoothing)
minv_noise, dPPn_res = pylops.avo.prestack.PrestackInversion(
    dPPn_dense,
    theta,
    wav,
    m0=mback,
    linearization="akirich",
    explicit=False,
    epsR=5e-1,
    returnres=True,
    **dict(damp=1e-1, iter_lim=100)
)

```

The data, inverted models and residuals are now displayed.

```

# 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")

```

(continues on next page)

(continued from previous page)

```

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
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"$\theta$")

```

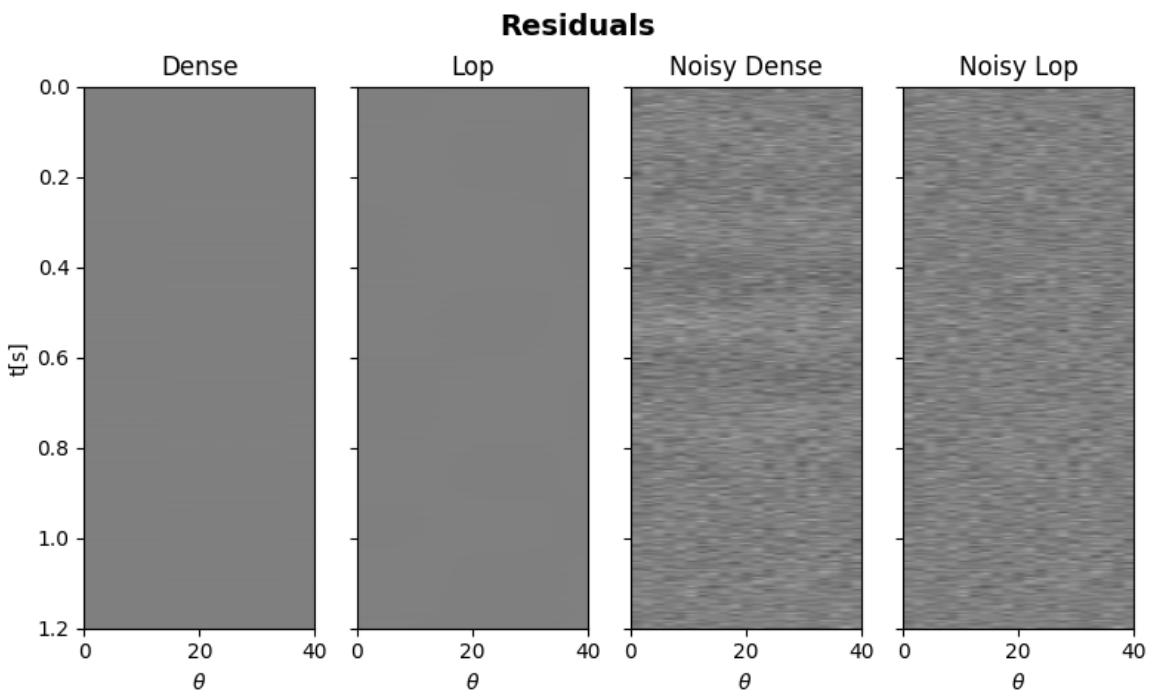
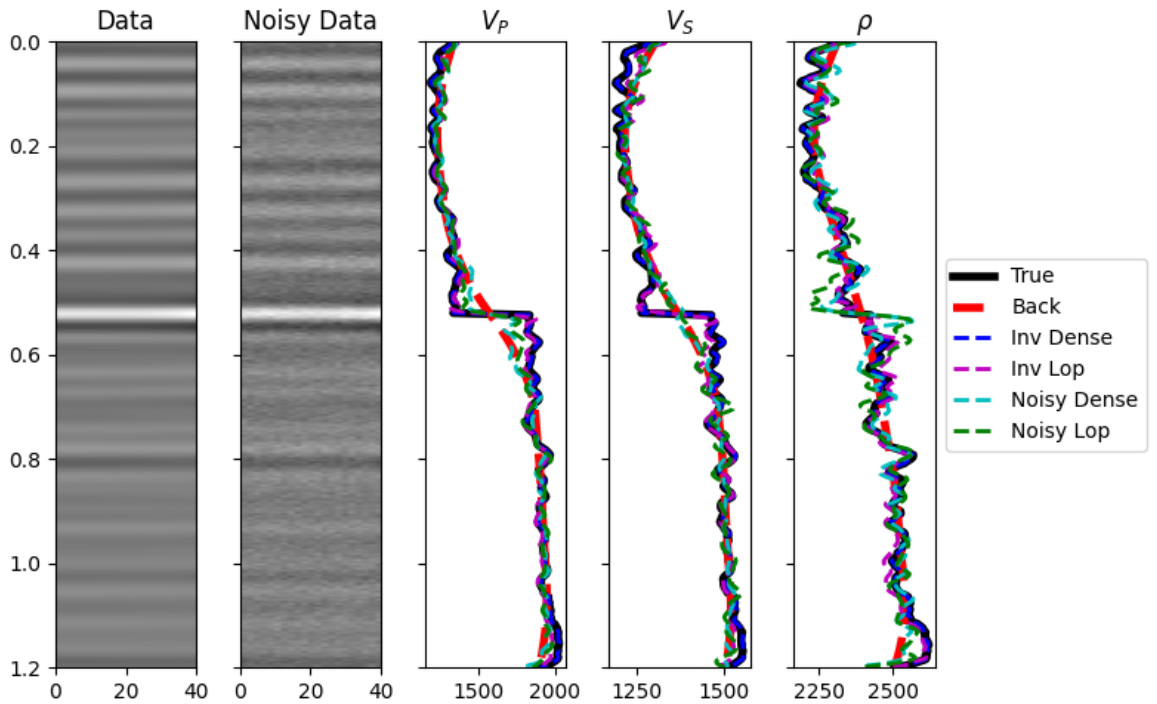
(continues on next page)

(continued from previous page)

```

axs[3].axis("tight")
plt.tight_layout()
plt.subplots_adjust(top=0.85)

```



Finally before moving to the 2d example, we consider the case when both PP and PS data are available. A joint PP-PS

inversion can be easily solved as follows.

```

PSop = pylops.avo.prestack.PrestackLinearModelling(
    2 * wav, theta, vsvp=vsvp, nt0=nt0, linearization="ps"
)
PPPSop = pylops.VStack((PPop, PSop))

# data
dPPPS = PPPSop * m.ravel()
dPPPS = dPPPS.reshape(2, nt0, ntheta)

dPPPSn = dPPPS + np.random.normal(0, 1e-2, dPPPS.shape)

# Invert
minvPPSP, dPPPS_res = pylops.avo.prestack.PrestackInversion(
    dPPPS,
    theta,
    [wav, 2 * wav],
    m0=mback,
    linearization=["fatti", "ps"],
    epsR=5e-1,
    returnres=True,
    **dict(damp=1e-1, iter_lim=100)
)

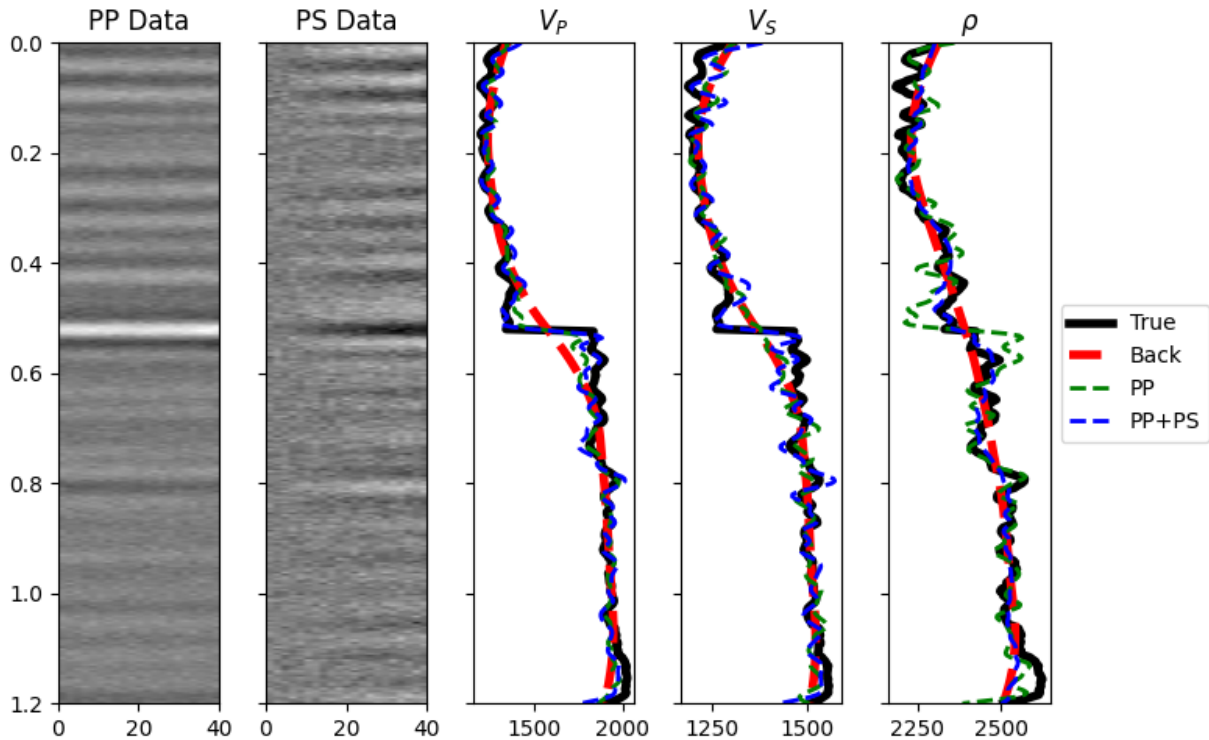
# Data and model
fig, (axd, axdn, axvp, axvs, axrho) = plt.subplots(1, 5, figsize=(8, 5), sharey=True)
axd.imshow(
    dPPPSn[0],
    cmap="gray",
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=-np.abs(dPPPSn[0]).max(),
    vmax=np.abs(dPPPSn[0]).max(),
)
axd.set_title("PP Data")
axd.axis("tight")
axdn.imshow(
    dPPPSn[1],
    cmap="gray",
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=-np.abs(dPPPSn[1]).max(),
    vmax=np.abs(dPPPSn[1]).max(),
)
axdn.set_title("PS 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_noise[:, 0]), t0, "--g", lw=2, label="PP")
axvp.plot(np.exp(minvPPSP[:, 0]), t0, "--b", lw=2, label="PP+PS")
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_noise[:, 1]), t0, "--g", lw=2, label="PP")
axvs.plot(np.exp(minvPPSP[:, 1]), t0, "--b", lw=2, label="PP+PS")
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_noise[:, 2]), t0, "--g", lw=2, label="PP")

```

(continues on next page)

(continued from previous page)

```
axrho.plot(np.exp(minvPPSP[:, 2]), t0, "--b", lw=2, label="PP+PS")
axrho.set_title(r"$\rho$")
axrho.legend(loc="center left", bbox_to_anchor=(1, 0.5))
axd.axis("tight")
plt.tight_layout()
```



We move now to a 2d example. First of all the model is loaded and data generated.

```
# model
inputfile = "../testdata/avo/poststack_model.npz"

model = np.load(inputfile)
x, z = model["x"][:, :6] / 1000.0, model["z"][:300] / 1000.0
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,
```

(continues on next page)

(continued from previous page)

```

    theta,
    vsvp=vsvp,
    nt0=nz,
    spatdims=(nx,),
    linearization="akirich",
    explicit=True,
)

# lop operator
PPop = pylops.avo.prestack.PrestackLinearModelling(
    wav, theta, vsvp=vsvp, nt0=nz, spatdims=(nx,), linearization="akirich"
)

# data
dPP = PPop_dense * m.swapaxes(0, 1).ravel()
dPP = dPP.reshape(ntheta, nz, nx).swapaxes(0, 1)
dPPn = dPP + np.random.normal(0, 5e-2, dPP.shape)

```

Finally we perform the same 4 different inversions as in the post-stack tutorial (see [07. Post-stack inversion](#) for more details).

```

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

# dense inversion with noisy data
minv_dense_noisy = 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 = 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 = 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

```

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):
        axs[ip].imshow(
            m[:, ip], extent=(x[0], x[-1], z[-1], z[0]), vmin=vmin, vmax=vmax,
            cmap=cmap
        )
        axs[ip].set_title("%s - %s" % (param, title))
        axs[ip].axis("tight")
        plt.setp(axs[1].get_yticklabels(), visible=False)
        plt.setp(axs[2].get_yticklabels(), visible=False)

# 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",
    linestyle="--",
)
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",

```

(continues on next page)

(continued from previous page)

```

    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(
    dPPn[:, 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",
    linestyle="--",
)
ax1.set_xlabel("x [km]")
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,

```

(continues on next page)

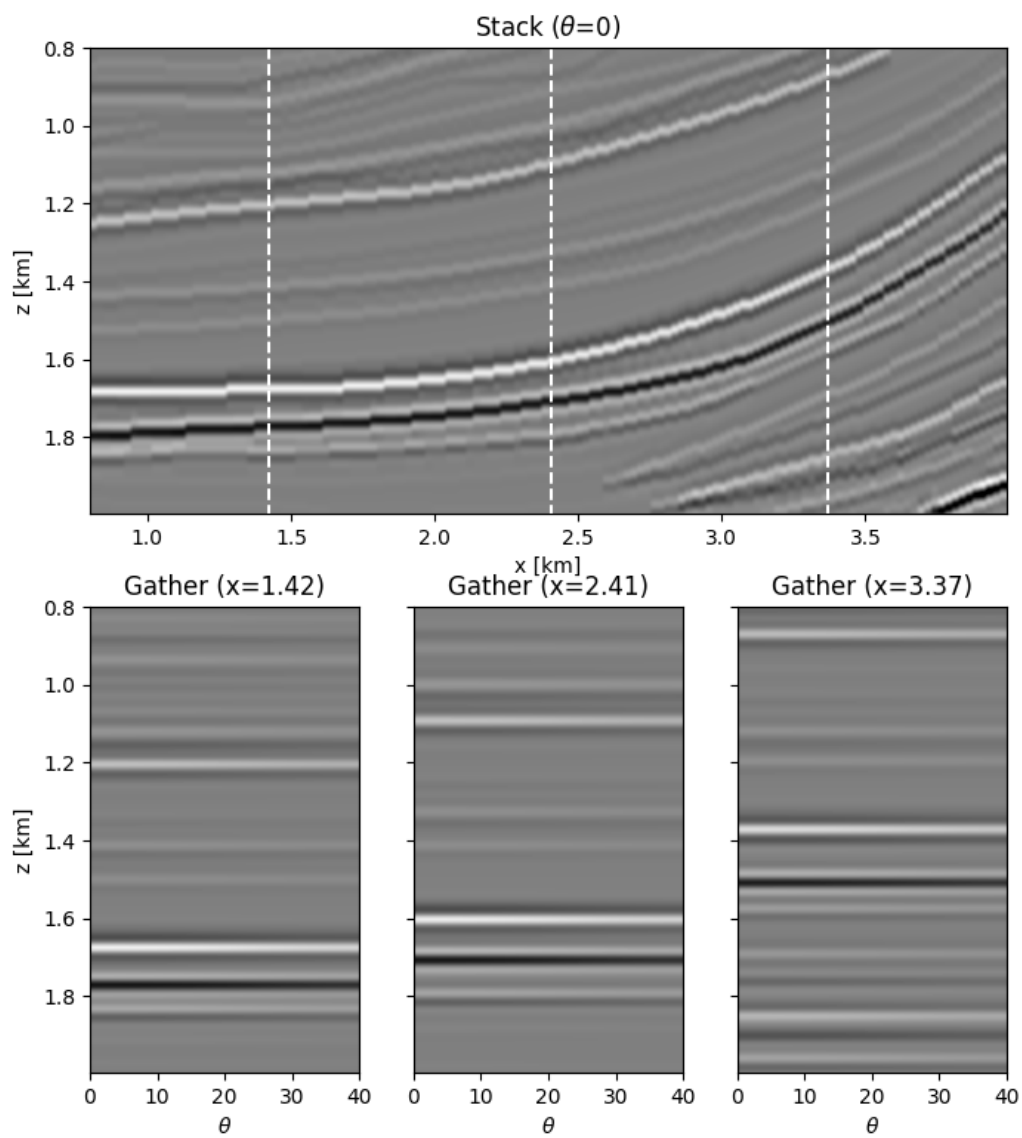
```

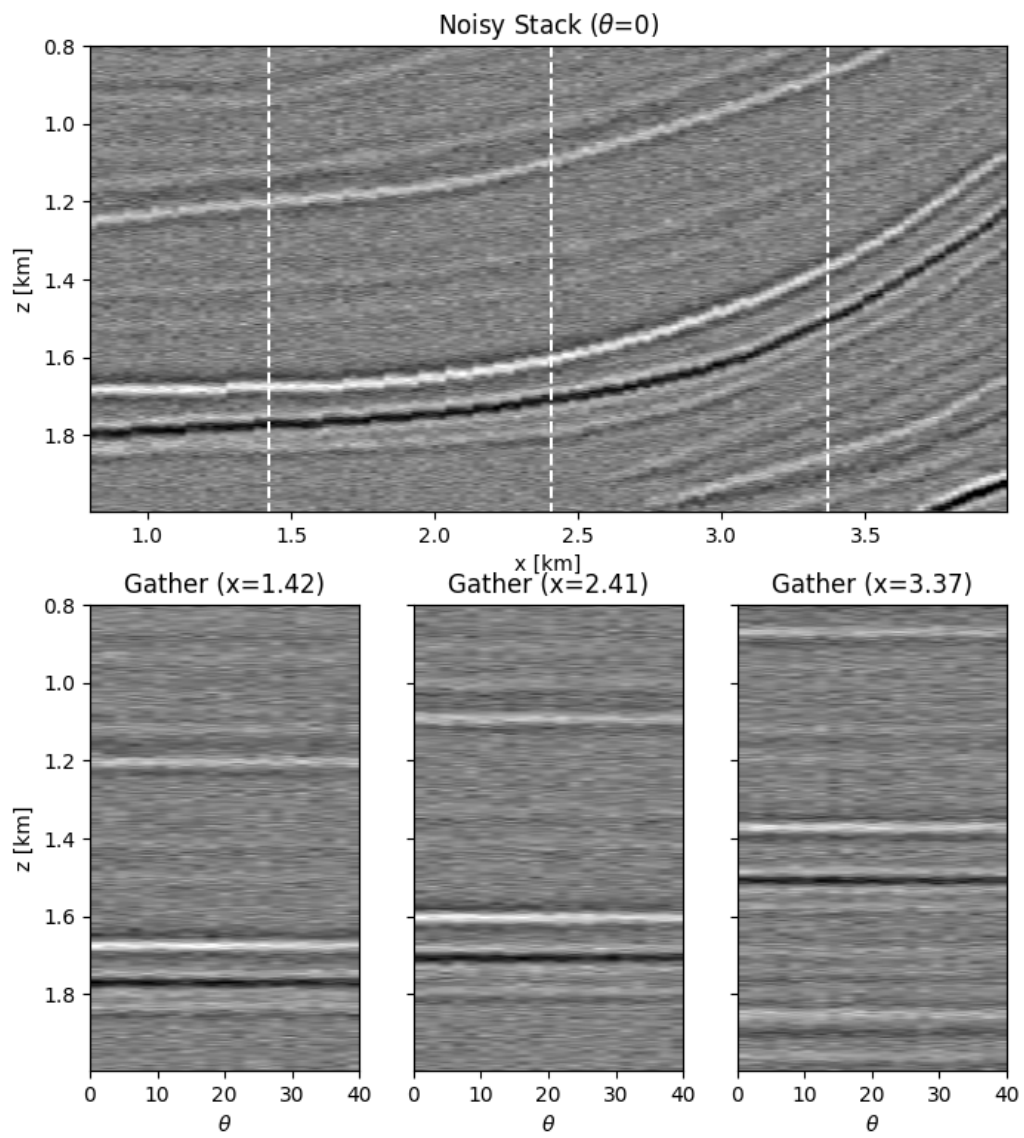
)
ax3.set_title(r"Gather (x=%.2f)" % x[nx // 2])
ax3.set_xlabel(r"$\theta$")
ax3.axis("tight")
ax4.imshow(
    dPPn[:, :, 4 * nx // 5],
    cmap="gray",
    extent=(theta[0], theta[-1], z[-1], z[0]),
    vmin=-0.4,
    vmax=0.4,
)
ax4.set_xlabel(r"$\theta$")
ax4.set_title(r"Gather (x=%.2f)" % x[4 * nx // 5])
ax4.axis("tight")
plt.setp(ax3.get_yticklabels(), visible=False)
plt.setp(ax4.get_yticklabels(), visible=False)

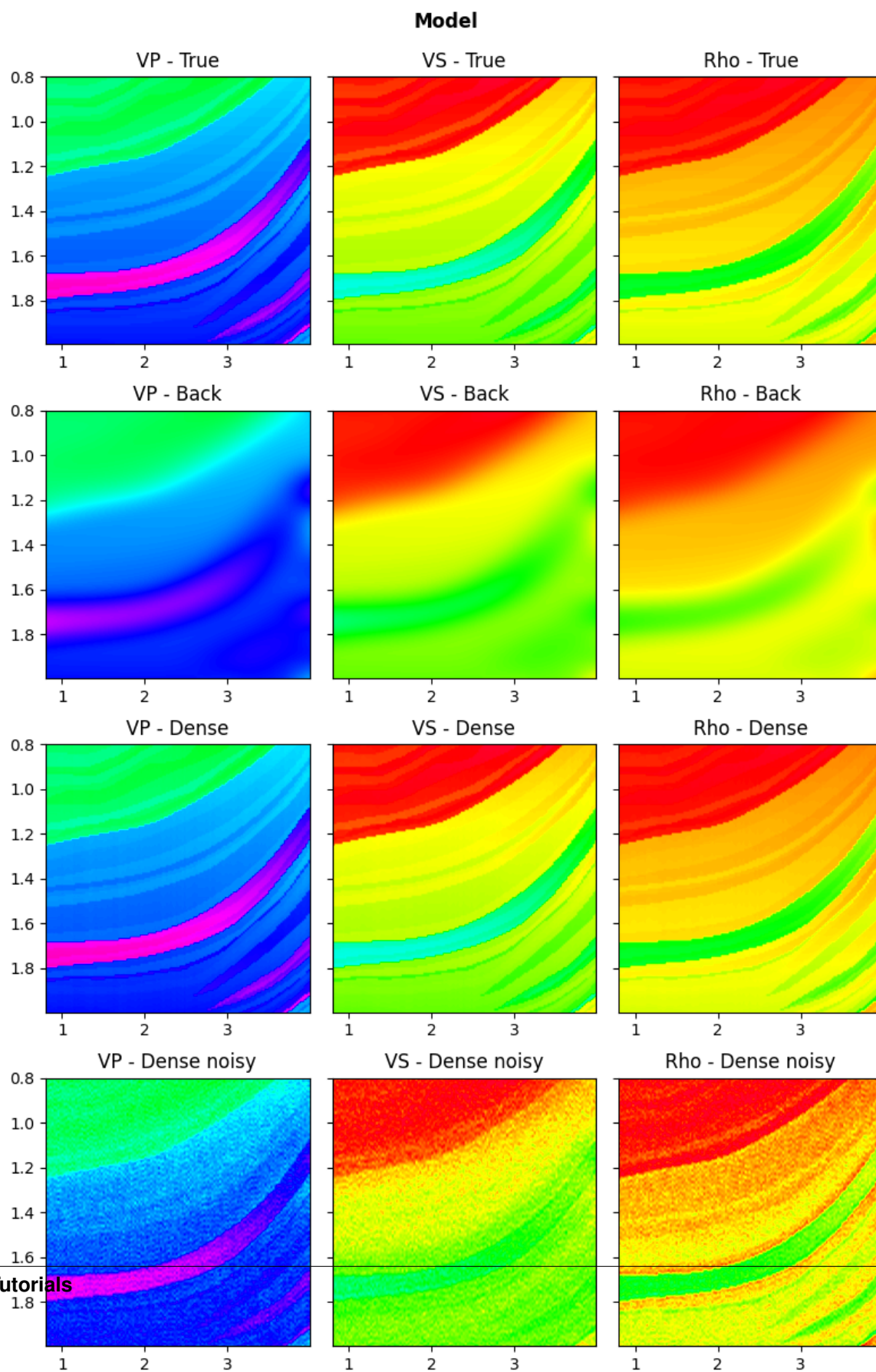
# inverted models
fig, axs = plt.subplots(6, 3, figsize=(8, 19))
fig.suptitle("Model", fontsize=12, fontweight="bold", y=0.95)
plotmodel(axs[0], m, x, z, m.min(), m.max(), title="True")
plotmodel(axs[1], mback, x, z, m.min(), m.max(), title="Back")
plotmodel(axs[2], minv_dense, x, z, m.min(), m.max(), 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)

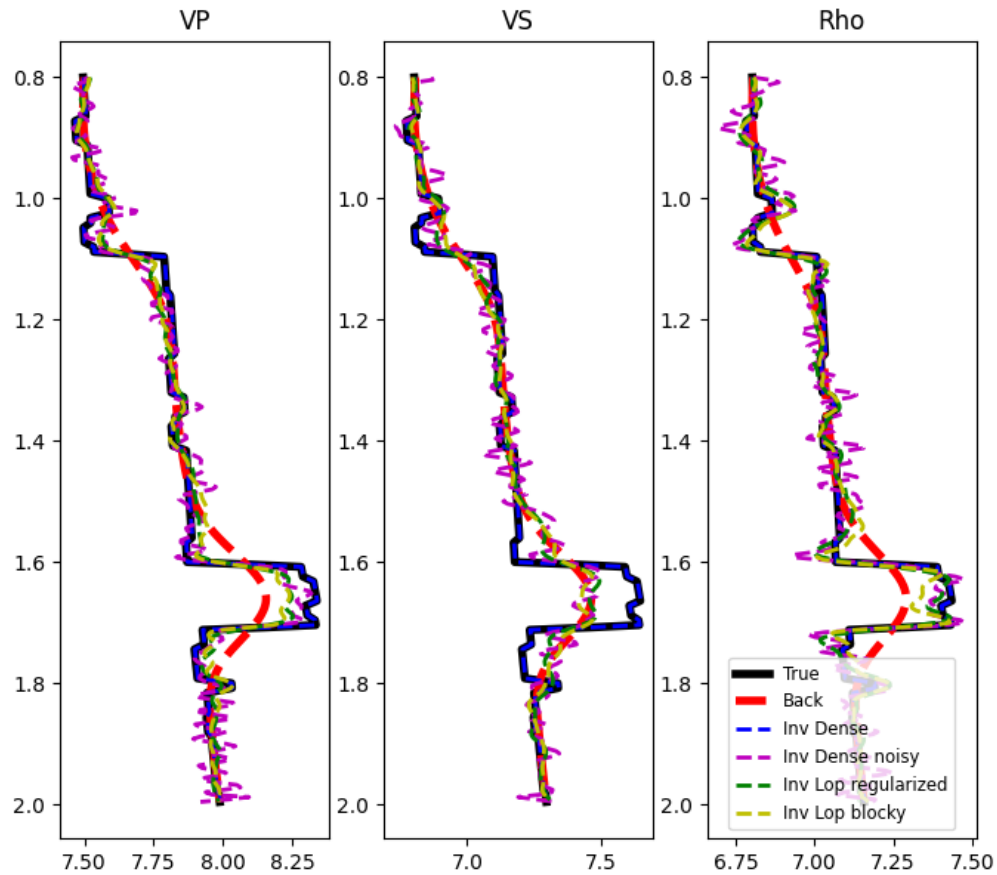
fig, axs = plt.subplots(1, 3, figsize=(8, 7))
for ip, param in enumerate(["VP", "VS", "Rho"]):
    axs[ip].plot(m[:, ip, nx // 2], z, "k", lw=4, label="True")
    axs[ip].plot(mback[:, ip, nx // 2], z, "--r", lw=4, label="Back")
    axs[ip].plot(minv_dense[:, ip, nx // 2], z, "--b", lw=2, label="Inv 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")

```









Out:

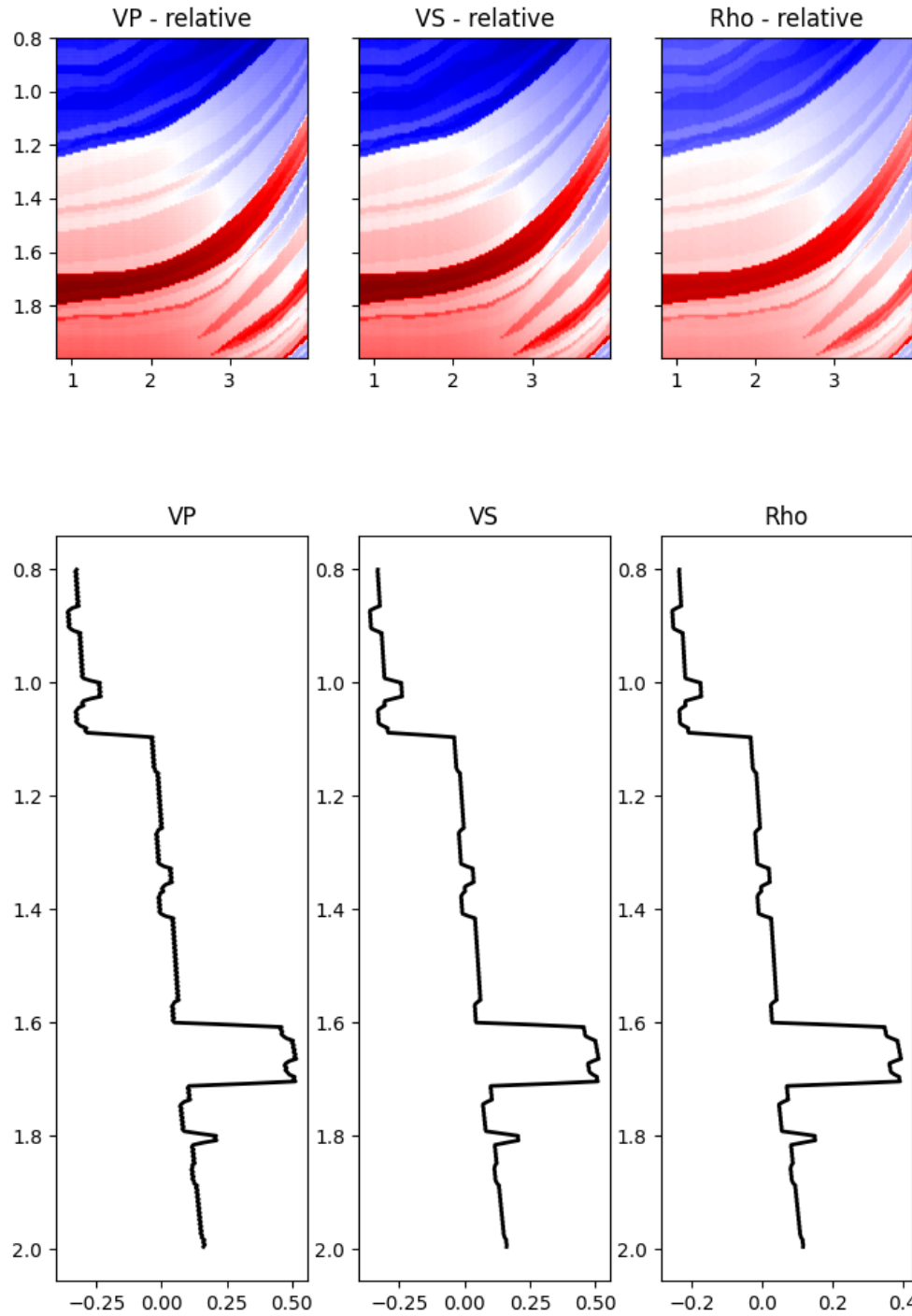
```
<matplotlib.legend.Legend object at 0x7f6aed6fcda0>
```

While the background model `m0` has been provided in all the examples so far, it is worth showing that the module `pylops.avo.prestack.PrestackInversion` can also produce so-called relative elastic parameters (i.e., variations from an average medium property) when the background model `m0` is not available.

```
dminv = pylops.avo.prestack.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()
```



Total running time of the script: (0 minutes 34.333 seconds)

3.5.9 09. Multi-Dimensional Deconvolution

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

```
import warnings

import matplotlib.pyplot as plt
import numpy as np

import pylops
from pylops.utils.seismicevents import hyperbolic2d, makeaxis
from pylops.utils.tapers import taper3d
from pylops.utils.wavelets import ricker

warnings.filterwarnings("ignore")
plt.close("all")

# sphinx_gallery_thumbnail_number = 5
```

Let's start by creating a set of hyperbolic events to be used as our MDC kernel

```
# 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 = [700.0]
amp_m = [1.0]

t0_G = [0.2, 0.5, 0.7]
vrms_G = [800.0, 1200.0, 1500.0]
amp_G = [1.0, 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):
```

(continues on next page)

(continued from previous page)

```

    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, dt=0.004, dr=1.0, dtype="float32"
)

# Create data
d = MDCop * m.ravel()
d = d.reshape(par["ny"], 2 * par["nt"] - 1)

```

Let's display what we have so far: operator, input model, and data

```

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()),

```

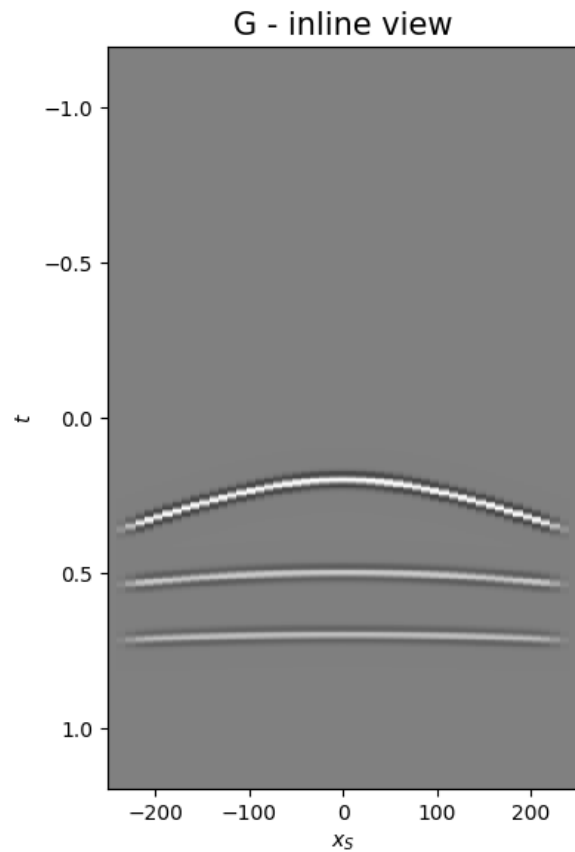
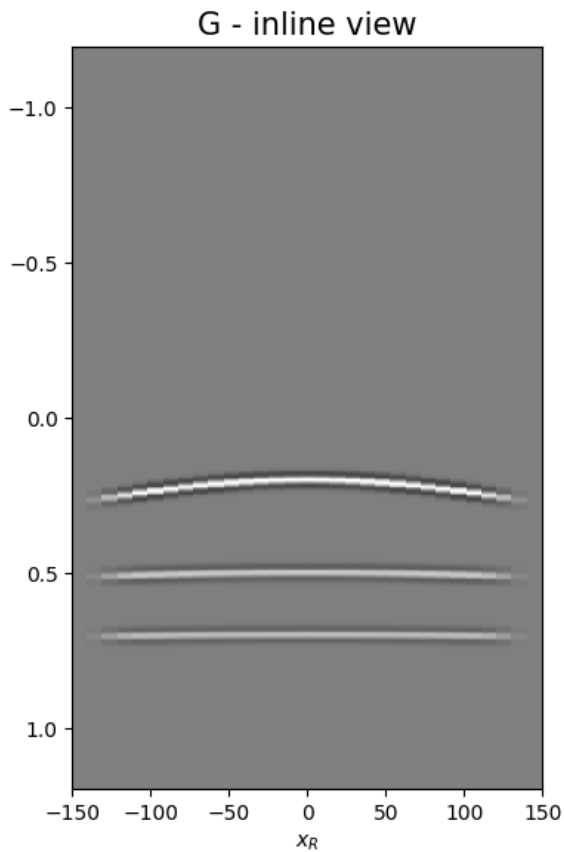
(continues on next page)

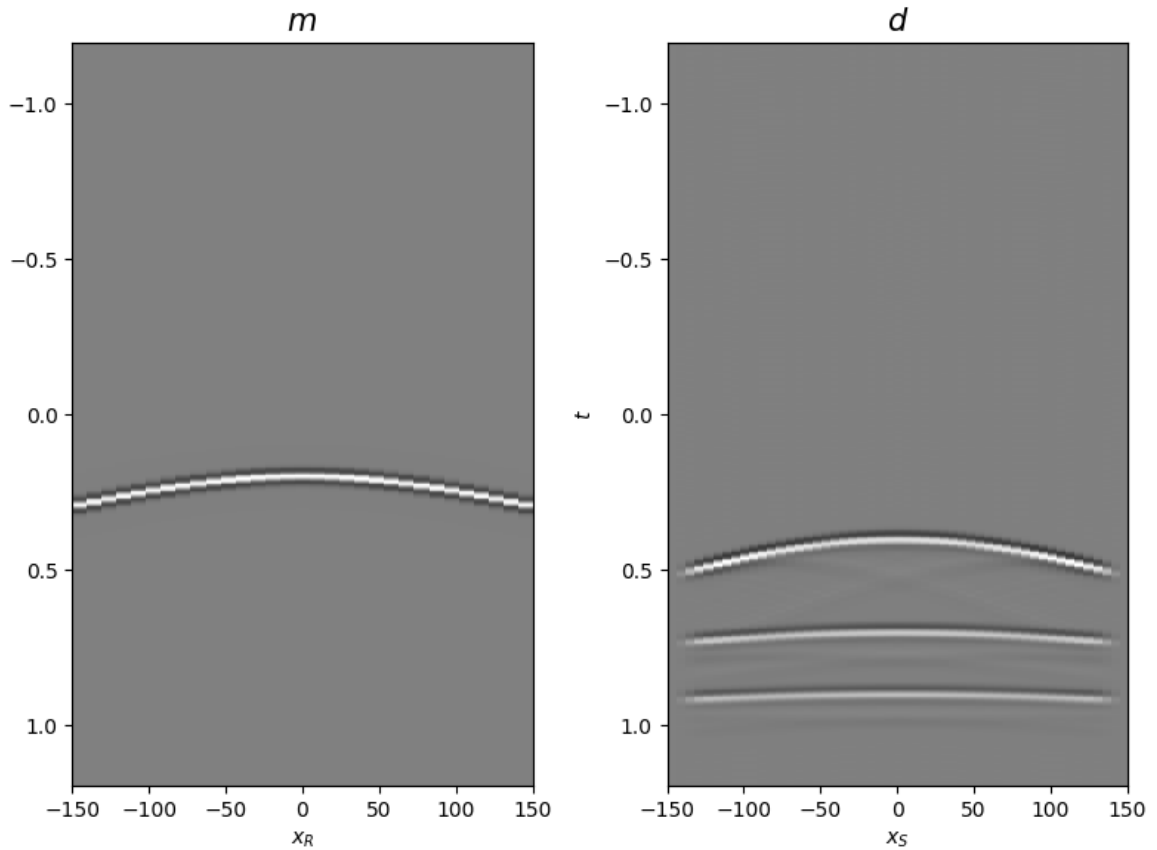
(continued from previous page)

```

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

```
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)

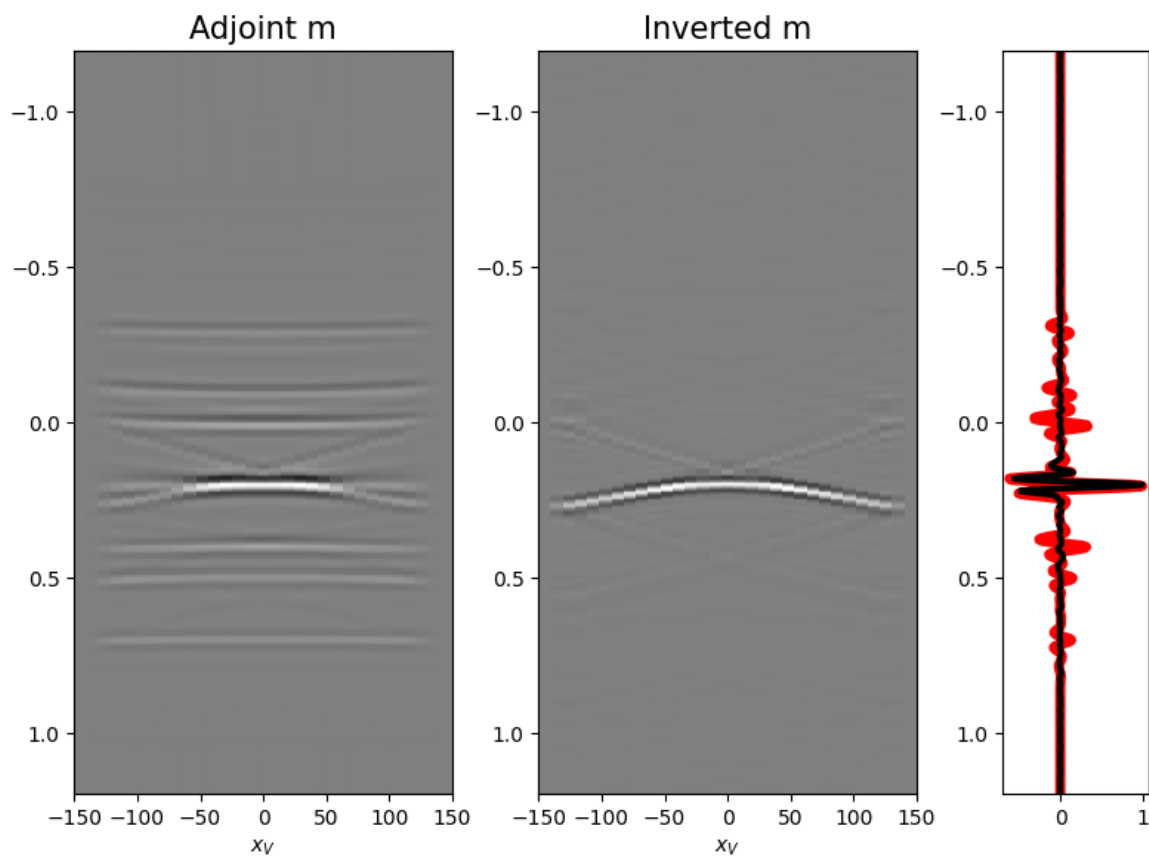
(continued from previous page)

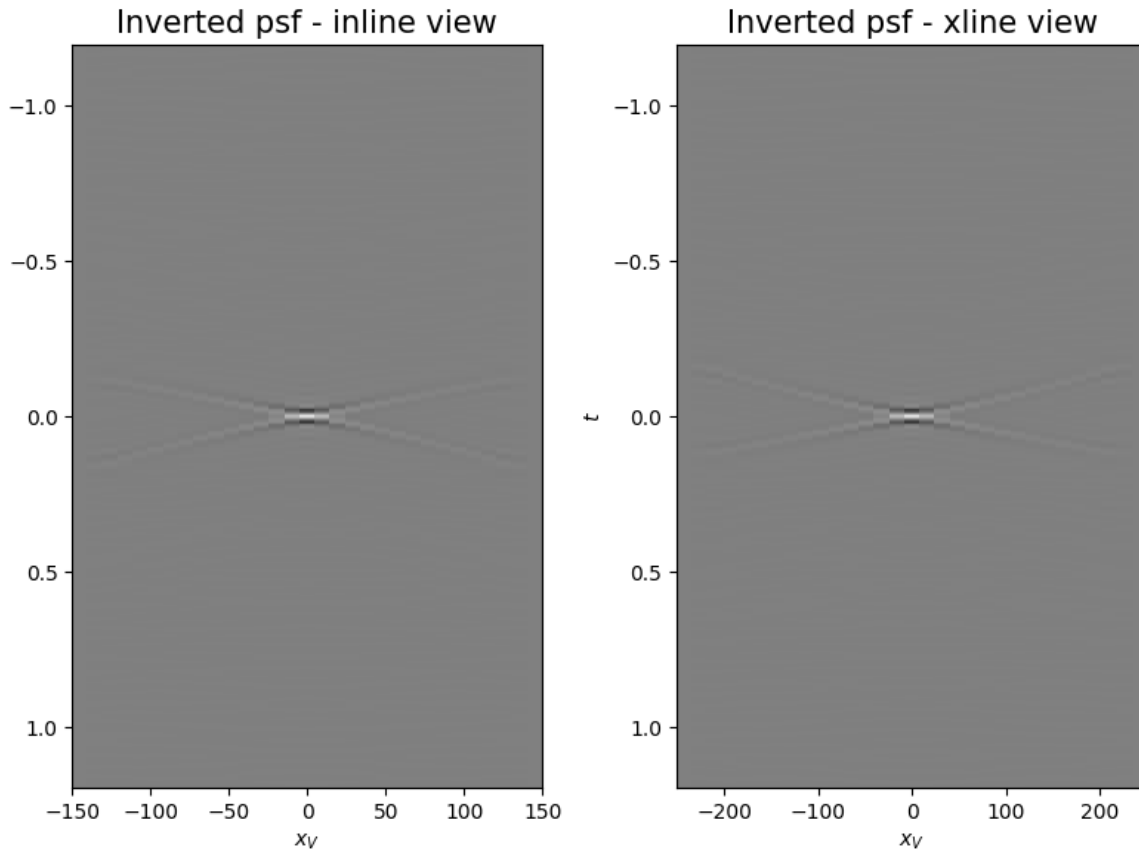
```

    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"$x_V$")
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()),
    extent=(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=(y.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

```
minvprec = pyllops.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)
)

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(
```

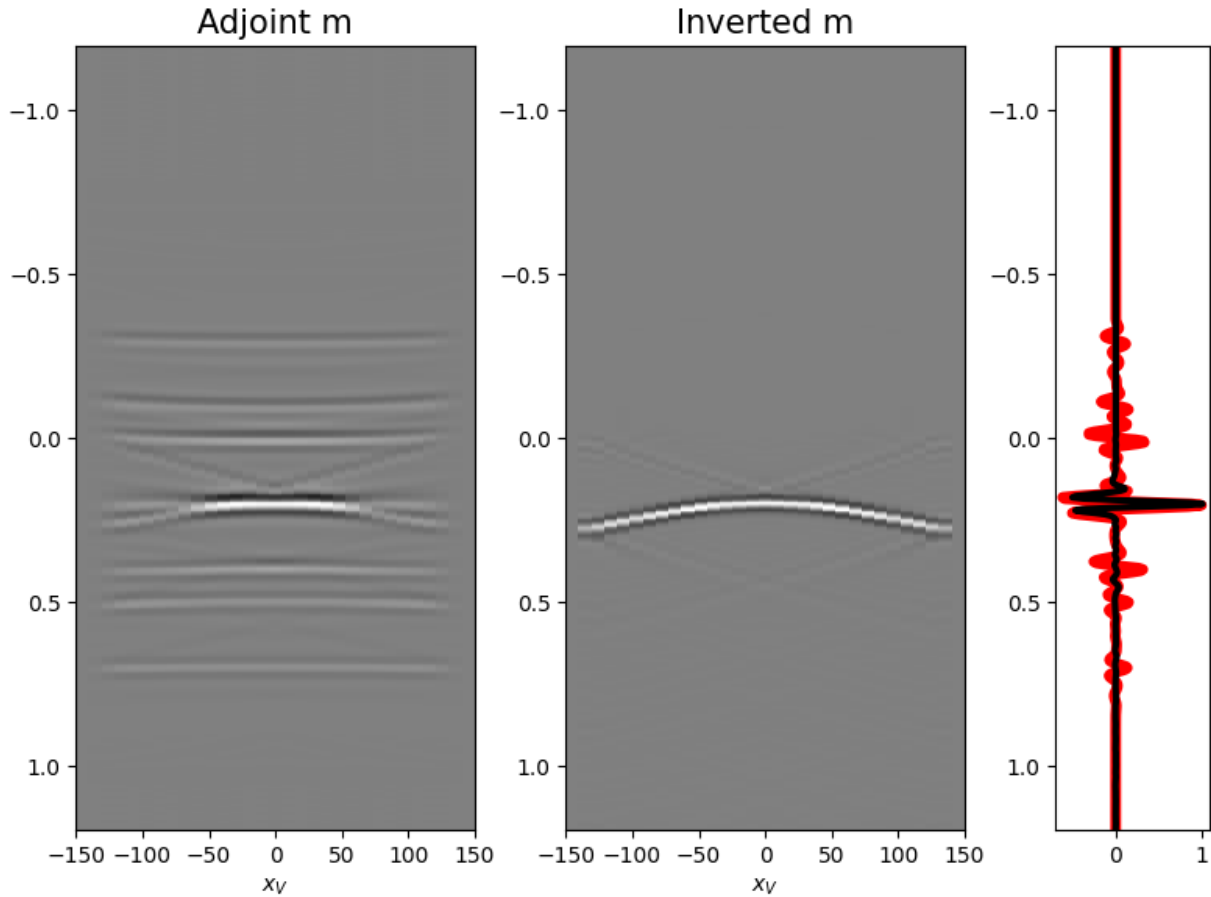
(continues on next page)

(continued from previous page)

```

    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"$x_V$")
axs[1].set_ylabel(r"$t$")
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"$x_V$")
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(
    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()

```



Total running time of the script: (0 minutes 10.482 seconds)

3.5.10 10. Marchenko redatuming by inversion

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

```
# sphinx_gallery_thumbnail_number = 5
# pylint: disable=C0103
import warnings

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

from pylops.waveeqprocessing import Marchenko

warnings.filterwarnings("ignore")
plt.close("all")
```

Let's start by defining some input parameters and loading the test data

```
# Input parameters
inputfile = "../testdata/marchenko/input.npz"
```

(continues on next page)

(continued from previous page)

```

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"]
nr = r.shape[1]
dr = r[0, 1] - r[0, 0]

# Sources
s = inputdata["s"]
ns = 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 = inputdata["R"][:, :, :-100]
R = np.swapaxes(R, 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]
ot, dt, nt = t[0], t[1] - t[0], len(t)

Gsub = np.apply_along_axis(convolve, 0, Gsub, wav, mode="full")
Gsub = Gsub[wav_c:][:nt]
G0sub = np.apply_along_axis(convolve, 0, G0sub, wav, mode="full")
G0sub = G0sub[wav_c:][: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")
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])

```

(continues on next page)

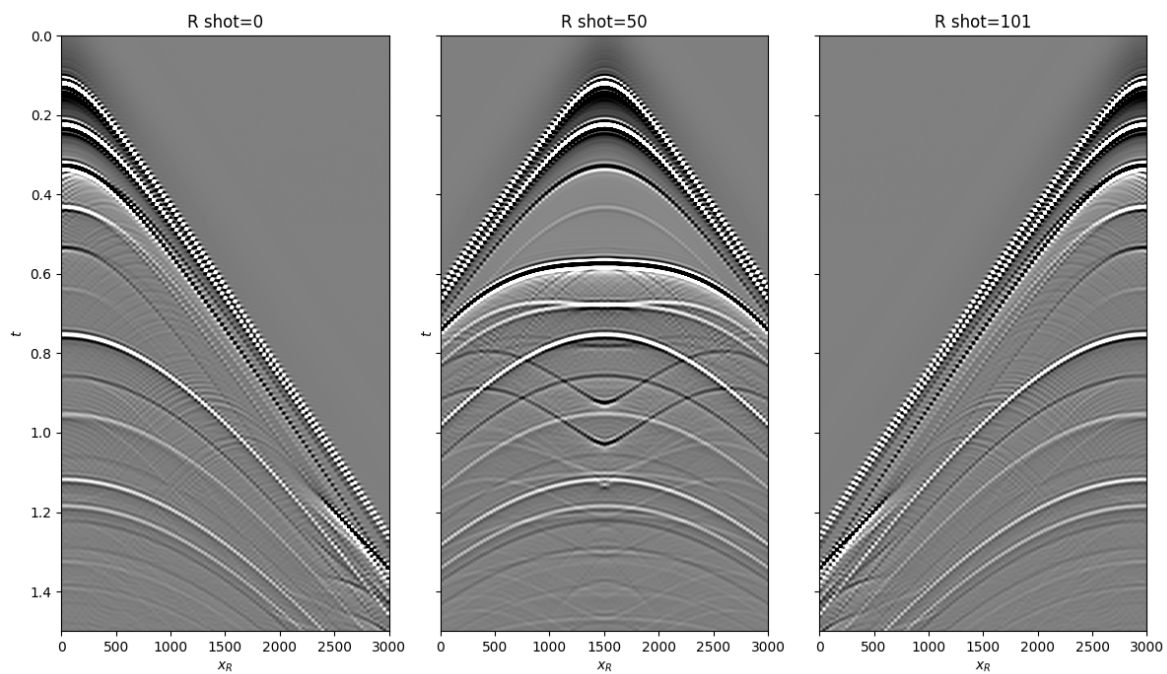
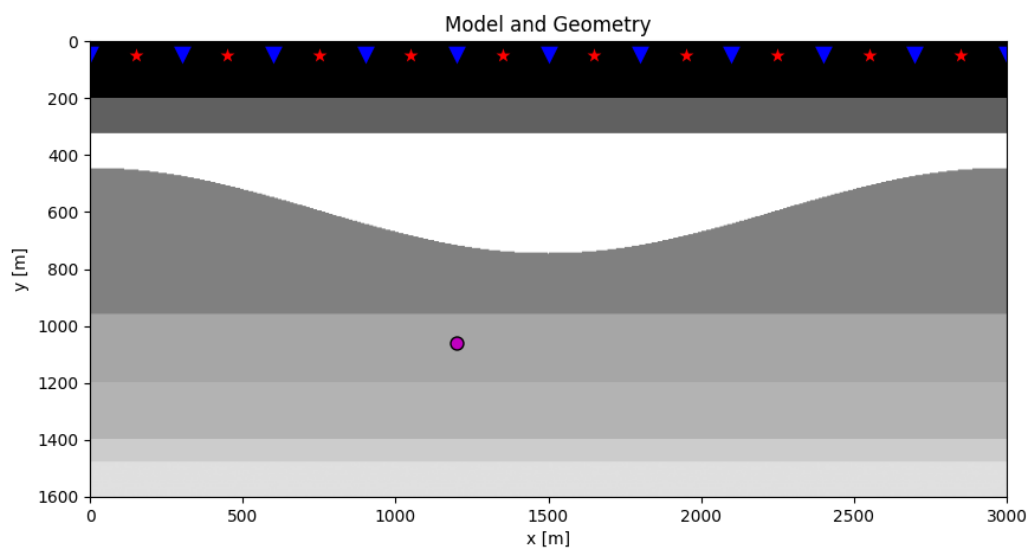
(continued from previous page)

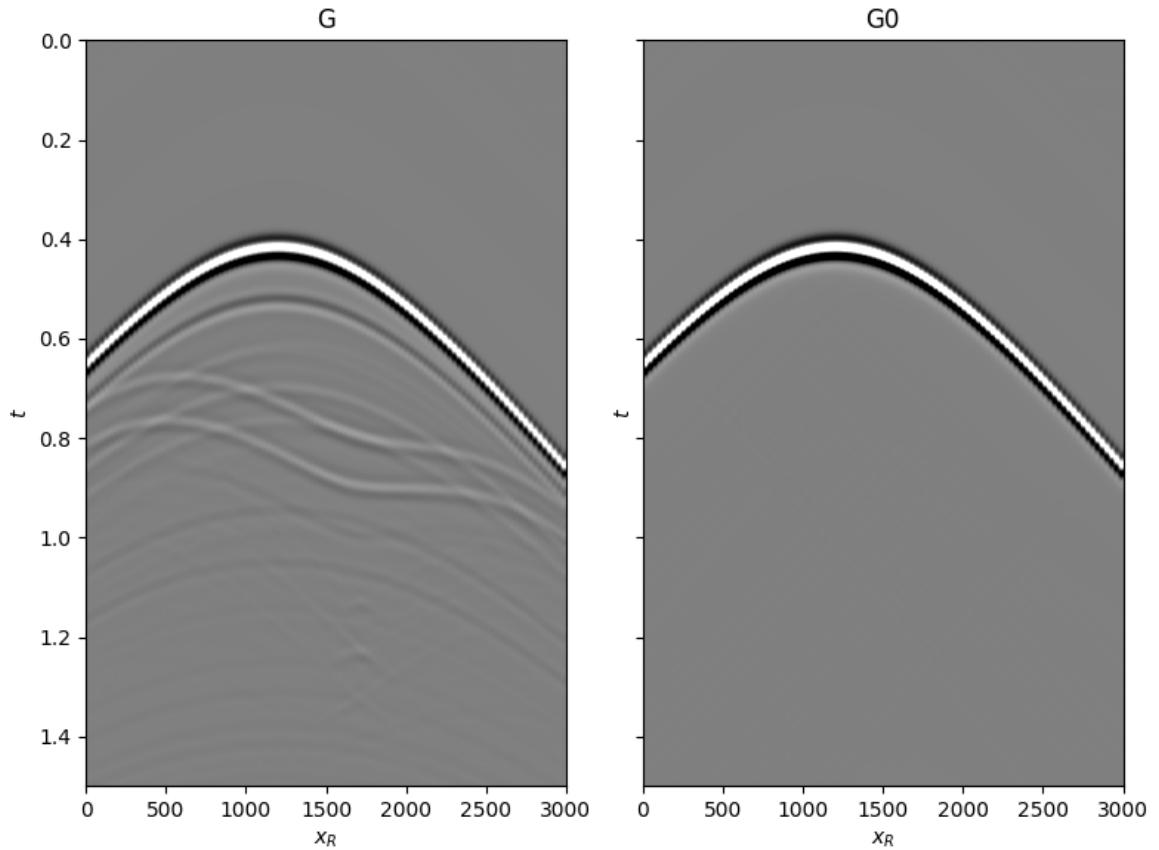
```

)
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].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 `vs`.

```
# 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, nsmooth=nsmooth
)

(
    fl_inv_minus,
    fl_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^H(Opu)=405.16509639614344 - u^H(Op^Hv)=405.1650963961446
Dot test passed, v^H(Opu)=172.06507561051328 - u^H(Op^Hv)=172.06507561051316
```

```
LSQR                Least-squares solution of  Ax = b
The matrix A has 282598 rows and 282598 columns
damp = 0.0000000000000000e+00    calc_var =      0
atol = 1.00e-08                  conlim = 1.00e+08
btol = 1.00e-08                  iter_lim =     10
```

Itn	x[0]	rlnorm	r2norm	Compatible	LS	Norm A	Cond A
0	0.000000e+00	3.134e+07	3.134e+07	1.0e+00	3.3e-08		
1	0.000000e+00	1.374e+07	1.374e+07	4.4e-01	9.3e-01	1.1e+00	1.0e+00
2	0.000000e+00	7.770e+06	7.770e+06	2.5e-01	3.9e-01	1.8e+00	2.2e+00
3	0.000000e+00	5.750e+06	5.750e+06	1.8e-01	3.3e-01	2.1e+00	3.4e+00
4	0.000000e+00	3.930e+06	3.930e+06	1.3e-01	3.4e-01	2.5e+00	5.1e+00
5	0.000000e+00	3.042e+06	3.042e+06	9.7e-02	2.6e-01	2.9e+00	6.8e+00
6	0.000000e+00	2.423e+06	2.423e+06	7.7e-02	2.2e-01	3.3e+00	8.6e+00
7	0.000000e+00	1.675e+06	1.675e+06	5.3e-02	2.5e-01	3.6e+00	1.1e+01
8	0.000000e+00	1.248e+06	1.248e+06	4.0e-02	2.0e-01	3.9e+00	1.3e+01
9	0.000000e+00	1.004e+06	1.004e+06	3.2e-02	1.5e-01	4.2e+00	1.4e+01
10	0.000000e+00	7.762e+05	7.762e+05	2.5e-02	1.8e-01	4.4e+00	1.6e+01

```
LSQR finished
The iteration limit has been reached
```

```
istop =      7    rlnorm = 7.8e+05    anorm = 4.4e+00    arnorm = 6.1e+05
itn    =     10    r2norm = 7.8e+05    acond = 1.6e+01    xnorm  = 3.6e+07
```

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

```
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,
```

(continues on next page)

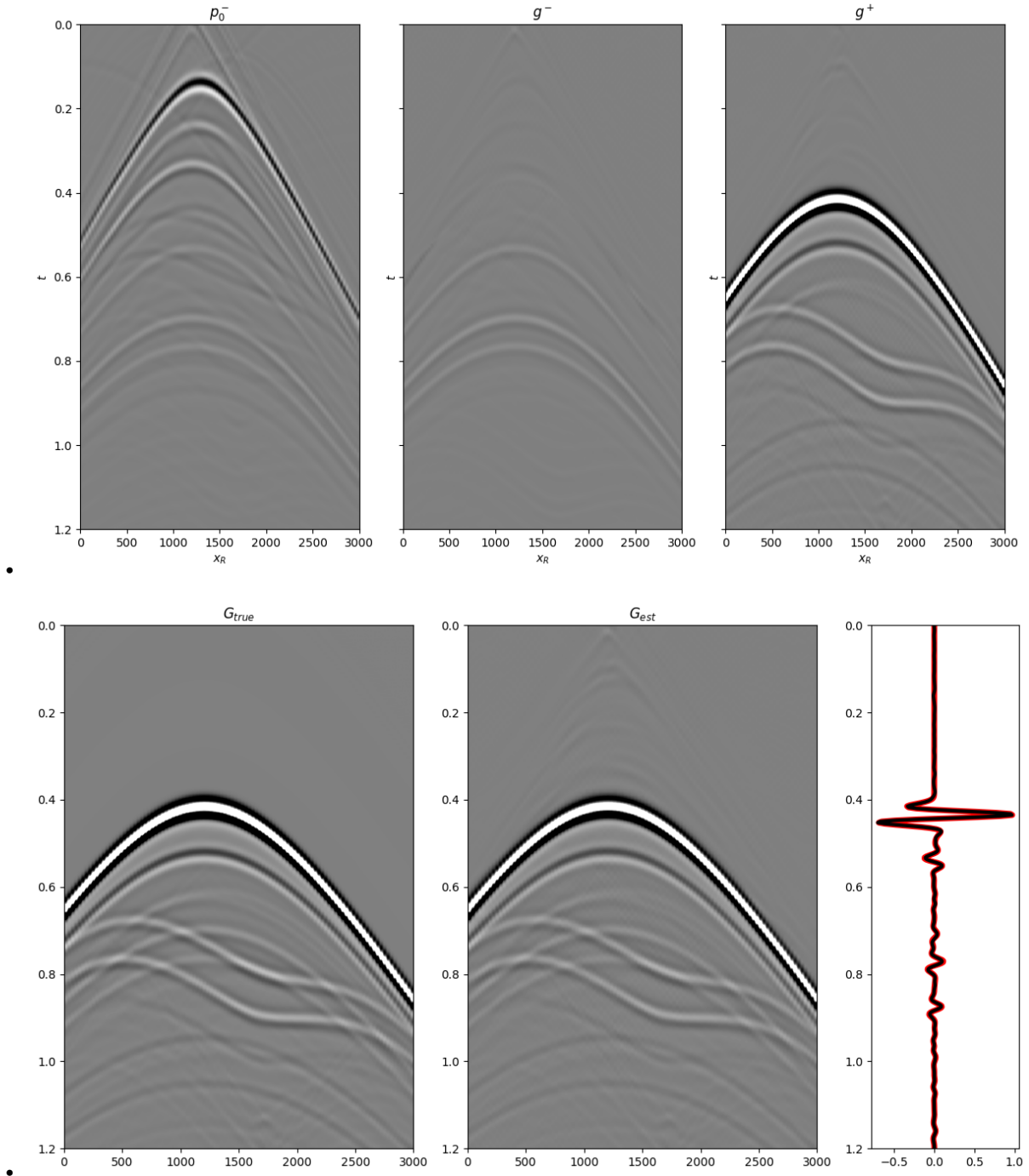
(continued from previous page)

```

    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()

fig = plt.figure(figsize=(12, 7))
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(
    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 8.712 seconds)

3.5.11 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.

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

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.

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

# linear events
v = 1500
t0 = [0.1, 0.2, 0.3]
theta = [0, 0, 0]
amp = [1.0, -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.


```

# 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.ravel()
xadj = xadj.reshape(npx, par["nt"])

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

# filtering
xfilt = np.zeros_like(xadj)
xfilt[npx // 2 - 3 : npx // 2 + 4] = xadj[npx // 2 - 3 : npx // 2 + 4]

yfilt = Rop * xfilt.ravel()
yfilt = yfilt.reshape(par["nx"], par["nt"])

# filtering on sparse transform
xinvfilt = np.zeros_like(xinv)
xinvfilt[npx // 2 - 3 : npx // 2 + 4] = xinv[npx // 2 - 3 : npx // 2 + 4]

yinvfilt = Rop * xinvfilt.ravel()
yinvfilt = yinvfilt.reshape(par["nx"], par["nt"])

```

Finally we visualize our results.

```

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")

```

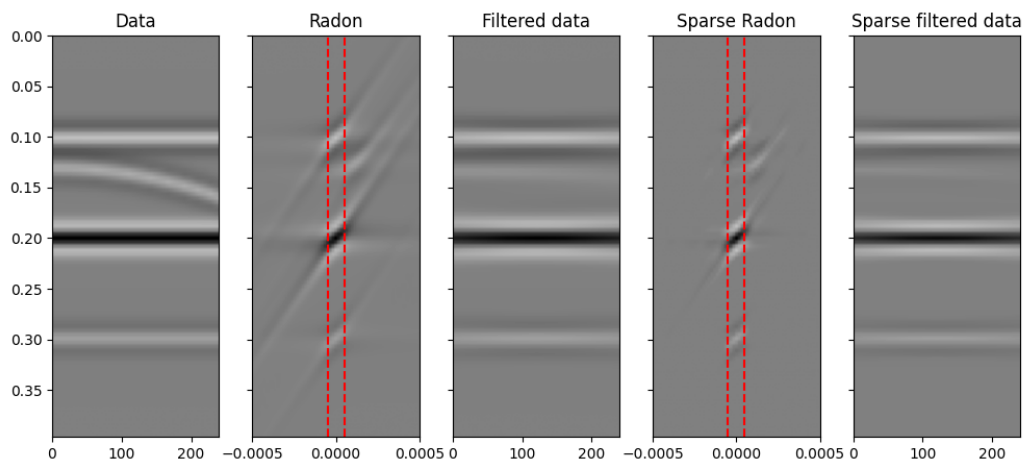
(continues on next page)

(continued from previous page)

```

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[npix // 2 - 3], color="r", linestyle="--")
axs[3].axvline(px[npix // 2 + 3], color="r", linestyle="--")
axs[3].set_title("Sparse Radon")
axs[3].axis("tight")
axs[4].imshow(
    yinvfilt.T,
    cmap="gray",
    vmin=-np.abs(y).max(),
    vmax=np.abs(y).max(),
    extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]),
)
axs[4].set_title("Sparse filtered data")
axs[4].axis("tight")

```



Out:

```
(0.0, 240.0, 0.396, 0.0)
```

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 a more suitable approach for preserving amplitudes could be to apply a parabolic Radon 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 24.632 seconds)

3.5.12 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).

$$\mathbf{y} = \mathbf{R}\mathbf{x}$$

Here $\mathbf{y} = [\mathbf{y}_{R_1}^T, \mathbf{y}_{R_2}^T, \dots, \mathbf{y}_{R_N}^T]^T$ where each vector \mathbf{y}_{R_i} contains all time samples recorded in the seismic data at the specific receiver R_i . Similarly, $\mathbf{x} = [\mathbf{x}_{r_1}^T, \mathbf{x}_{r_2}^T, \dots, \mathbf{x}_{r_M}^T]$, 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).

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

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

np.random.seed(0)
plt.close("all")
```

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

```
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 = [1.0, -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.

```
perc_subsampling = 0.6
nxsub = int(np.round(par["nx"] * perc_subsampling))
```

(continues on next page)

(continued from previous page)

```

iava = np.sort(np.random.permutation(np.arange(par["nx"]))[:nxsub])

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

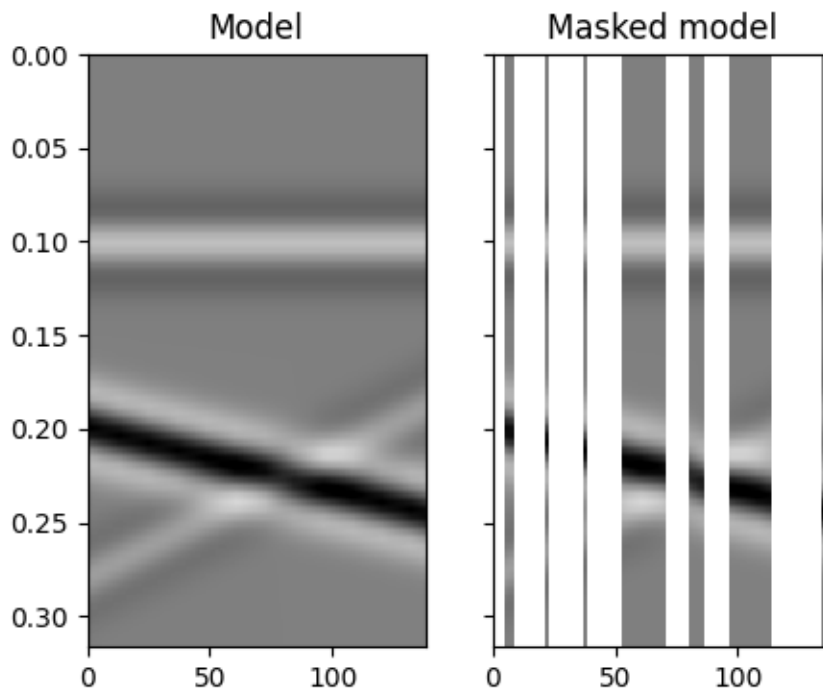
# data
y = Rop * x.ravel()
y = y.reshape(nxsub, par["nt"])

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

# 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

$$J = \|\mathbf{y} - \mathbf{R}\mathbf{x}\|_2 + \epsilon \nabla^2 \|\nabla \mathbf{x}\|_2$$

- sparsity-promoting inversion with `pylops.FFT2` operator used as sparsifying transform

$$J = \|\mathbf{y} - \mathbf{R}\mathbf{F}^H \mathbf{x}\|_2 + \epsilon \|\mathbf{F}^H \mathbf{x}\|_1$$

```
# 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 FFT2
nfft = 2 ** 8
```

(continues on next page)

(continued from previous page)

```

FFTop = pylops.signalprocessing.FFT2D(
    dims=[par["nx"], par["nt"]], nffts=[nfft, nfft], sampling=[par["dx"], par["dt"]]
)
X = FFTop * x.ravel()
X = np.reshape(X, (nfft, nfft))

x11, X11, 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)
)

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(
    x11.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")

fig, axs = plt.subplots(1, 3, figsize=(10, 2))
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],
    )
)

```

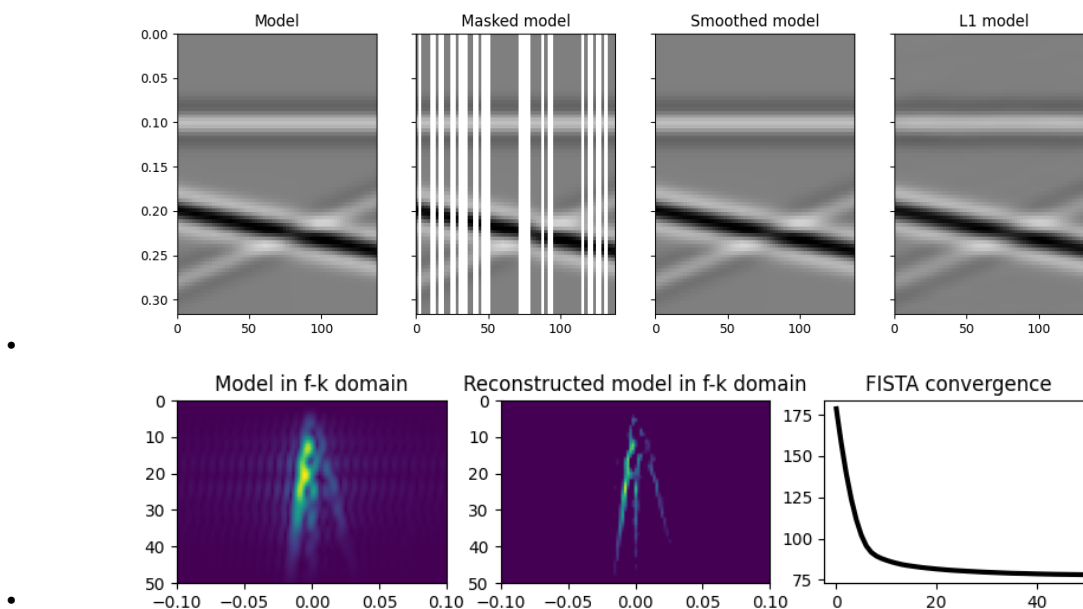
(continues on next page)

(continued from previous page)

```

        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")

```



Out:

```
Text(0.5, 1.0, '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.

```
npx = 40
```

(continues on next page)

(continued from previous page)

```

pxmax = 1e-3
px = np.linspace(-pxmax, pxmax, npix)
Radop = pylops.signalprocessing.Radon2D(taxis, xaxis, px, engine="numba")

RROP = Rop * Radop

# adjoint
Xadj_fromx = Radop.H * x.ravel()
Xadj_fromx = Xadj_fromx.reshape(npix, par["nt"])

Xadj = RROP.H * y.ravel()
Xadj = Xadj.reshape(npix, 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")
axs[0][2].imshow(
    xll.T,
    cmap="gray",
    vmin=-2,
    vmax=2,
    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]),

```

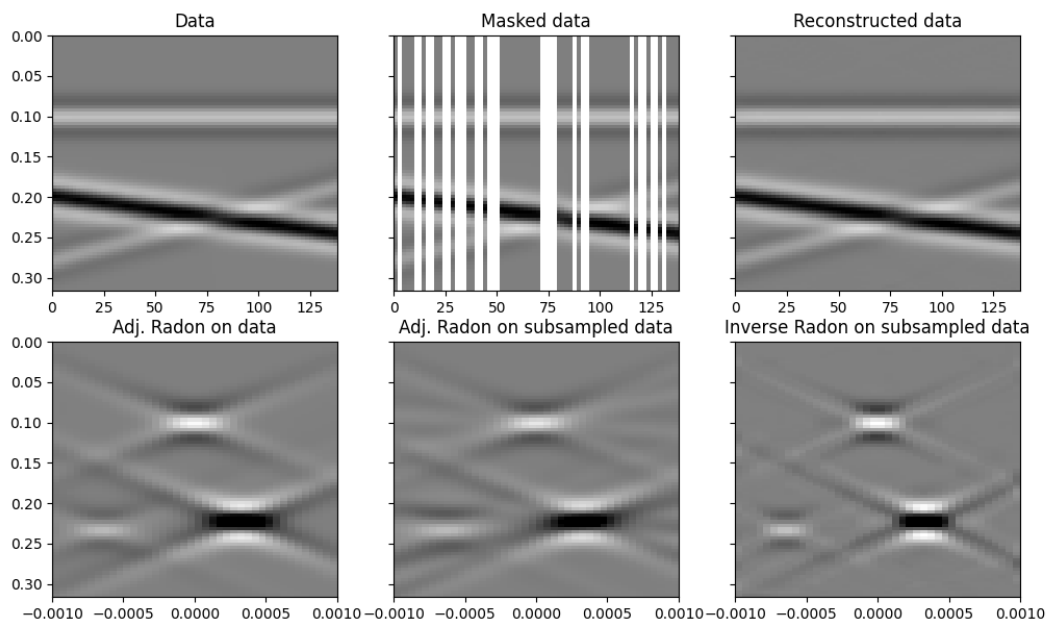
(continues on next page)

(continued from previous page)

```

)
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")

```



Out:

```
(-0.001, 0.001, 0.316, 0.0)
```

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.

```

inputfile = "../testdata/marchenko/input.npz"
inputdata = np.load(inputfile)

x = inputdata["R"][50, :, ::2]
x = x / np.abs(x).max()
taxis, xaxis = inputdata["t"][:, ::2], inputdata["r"][0]

```

(continues on next page)

(continued from previous page)

```

par = {}
par["nx"], par["nt"] = x.shape
par["dx"] = inputdata["r"][0, 1] - inputdata["r"][0, 0]
par["dt"] = inputdata["t"][1] - inputdata["t"][0]

# add wavelet
wav = inputdata["wav"][:, :2]
wav_c = np.argmax(wav)
x = np.apply_along_axis(convolve, 1, x, wav, mode="full")
x = x[:, wav_c:][:, : par["nt"]]

# 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.ravel()
xadj = Rop.H * y.ravel()

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

# apply mask
ymask = Rop.mask(x.ravel())

# sliding windows with radon transform
dx = par["dx"]
nwins = 4
nwin = 27
nover = 3
npx = 31
pxmax = 5e-4
px = np.linspace(-pxmax, pxmax, npix)
dimsd = x.shape
dims = (nwins * npix, 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
)

```

(continues on next page)

(continued from previous page)

```

# adjoint
RSop = Rop * Slidop

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

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

# inverse
x11, X11, _ = 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)
)

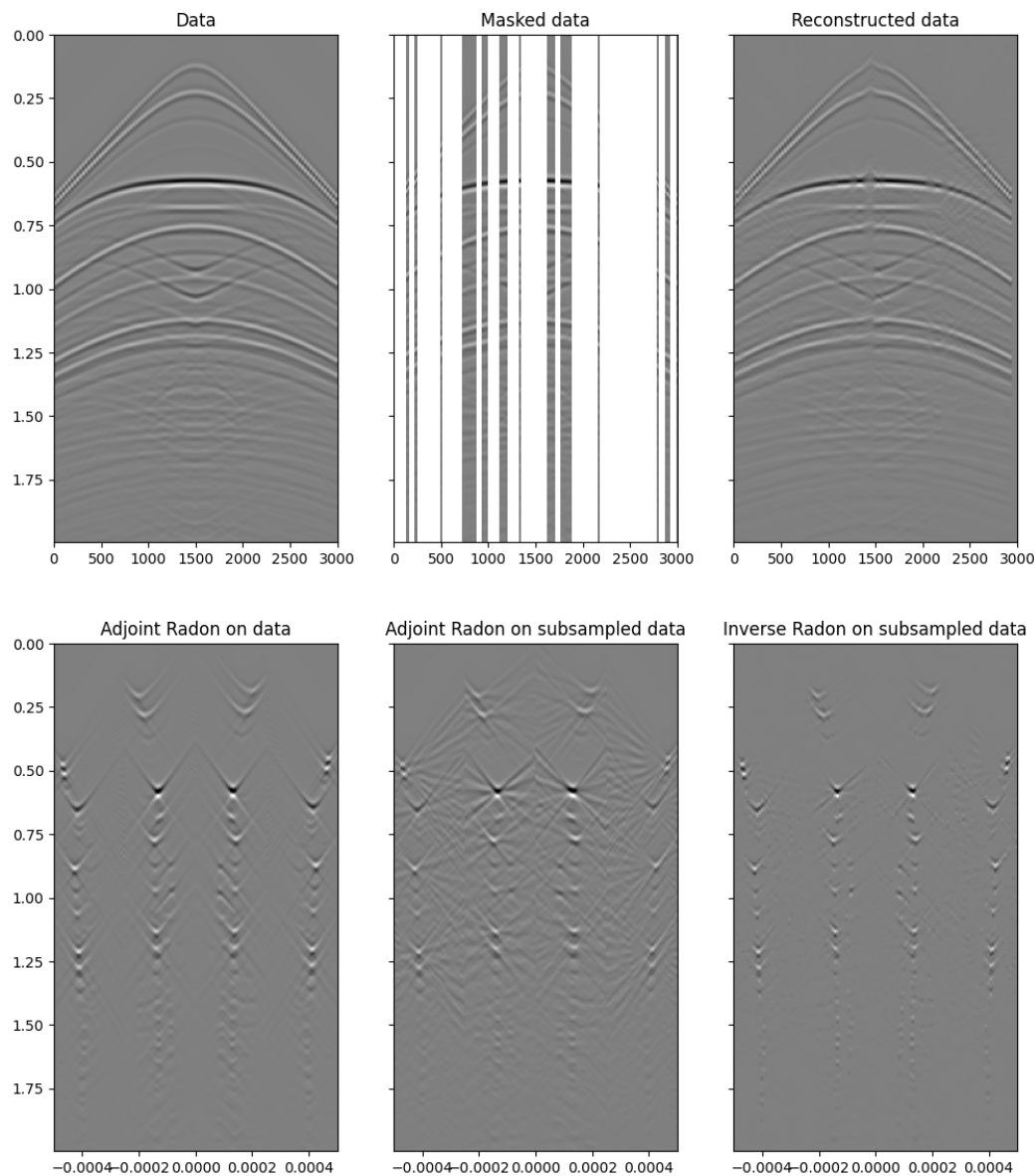
fig, axs = plt.subplots(2, 3, sharey=True, figsize=(12, 14))
axs[0][0].imshow(
    x.T,
    cmap="gray",
    vmin=-0.1,
    vmax=0.1,
    extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]),
)
axs[0][0].set_title("Data")
axs[0][0].axis("tight")
axs[0][1].imshow(
    ymask.T,
    cmap="gray",
    vmin=-0.1,
    vmax=0.1,
    extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]),
)
axs[0][1].set_title("Masked data")
axs[0][1].axis("tight")
axs[0][2].imshow(
    x11.T,
    cmap="gray",
    vmin=-0.1,
    vmax=0.1,
    extent=(xaxis[0], xaxis[-1], taxis[-1], taxis[0]),
)
axs[0][2].set_title("Reconstructed data")
axs[0][2].axis("tight")
axs[1][0].imshow(
    Xadj_fromx.T,
    cmap="gray",
    vmin=-1,
    vmax=1,

```

(continues on next page)

(continued from previous page)

```
        extent=(px[0], px[-1], taxis[-1], taxis[0]),
    )
    axs[1][0].set_title("Adjoint Radon on data")
    axs[1][0].axis("tight")
    axs[1][1].imshow(
        Xadj.T,
        cmap="gray",
        vmin=-0.6,
        vmax=0.6,
        extent=(px[0], px[-1], taxis[-1], taxis[0]),
    )
    axs[1][1].set_title("Adjoint Radon on subsampled data")
    axs[1][1].axis("tight")
    axs[1][2].imshow(
        Xl1.T,
        cmap="gray",
        vmin=-0.03,
        vmax=0.03,
        extent=(px[0], px[-1], taxis[-1], taxis[0]),
    )
    axs[1][2].set_title("Inverse Radon on subsampled data")
    axs[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 7.263 seconds)

3.5.13 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*.

```
import matplotlib.pyplot as plt

# sphinx_gallery_thumbnail_number = 3
import numpy as np
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"]
nr = r.shape[1]
dr = r[0, 1] - r[0, 0]

# Sources
s = inputdata["s"]

# Model
rho = inputdata["rho"]

# Axes
t = inputdata["t"]
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))
ax1 = plt.subplot2grid((1, 5), (0, 0), colspan=4)
ax2 = plt.subplot2grid((1, 5), (0, 4))
ax1.imshow(rho, cmap="gray", extent=(x[0], x[-1], z[-1], z[0]))
ax1.scatter(r[0, ::5], r[1, ::5], marker="v", s=150, c="b", edgecolors="k")
ax1.scatter(s[0], s[1], marker="x", s=250, c="r", edgecolors="k")
ax1.axis("tight")
ax1.set_xlabel("x [m]")
```

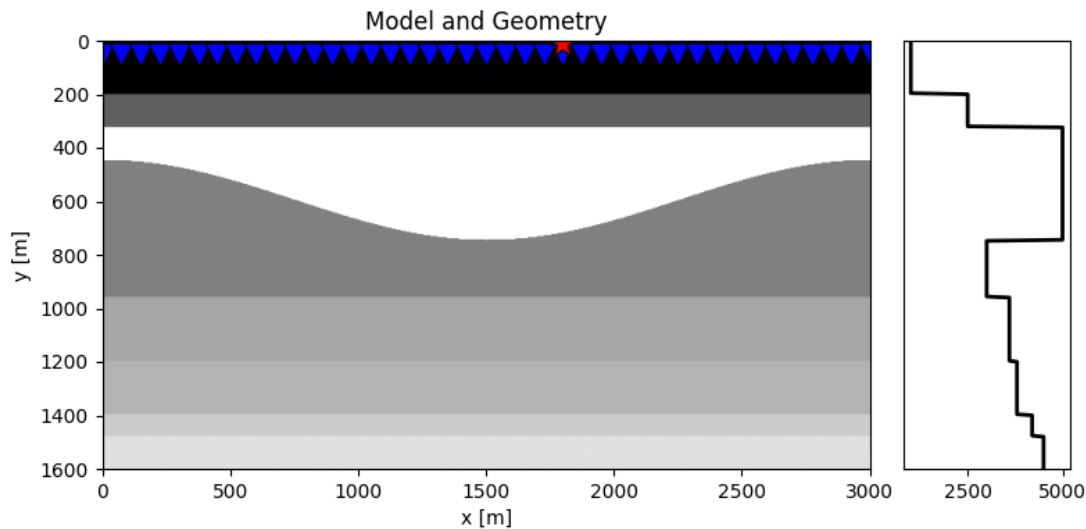
(continues on next page)

(continued from previous page)

```

ax1.set_ylabel("y [m]")
ax1.set_title("Model and Geometry")
ax1.set_xlim(x[0], x[-1])
ax1.set_ylim(z[-1], z[0])
ax2.plot(rho[:, len(x) // 2], z, "k", lw=2)
ax2.set_ylim(z[-1], z[0])
ax2.set_yticks([])

```



Out:

```
[ ]
```

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 traveltimes of the direct arrival.

```

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(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")

```

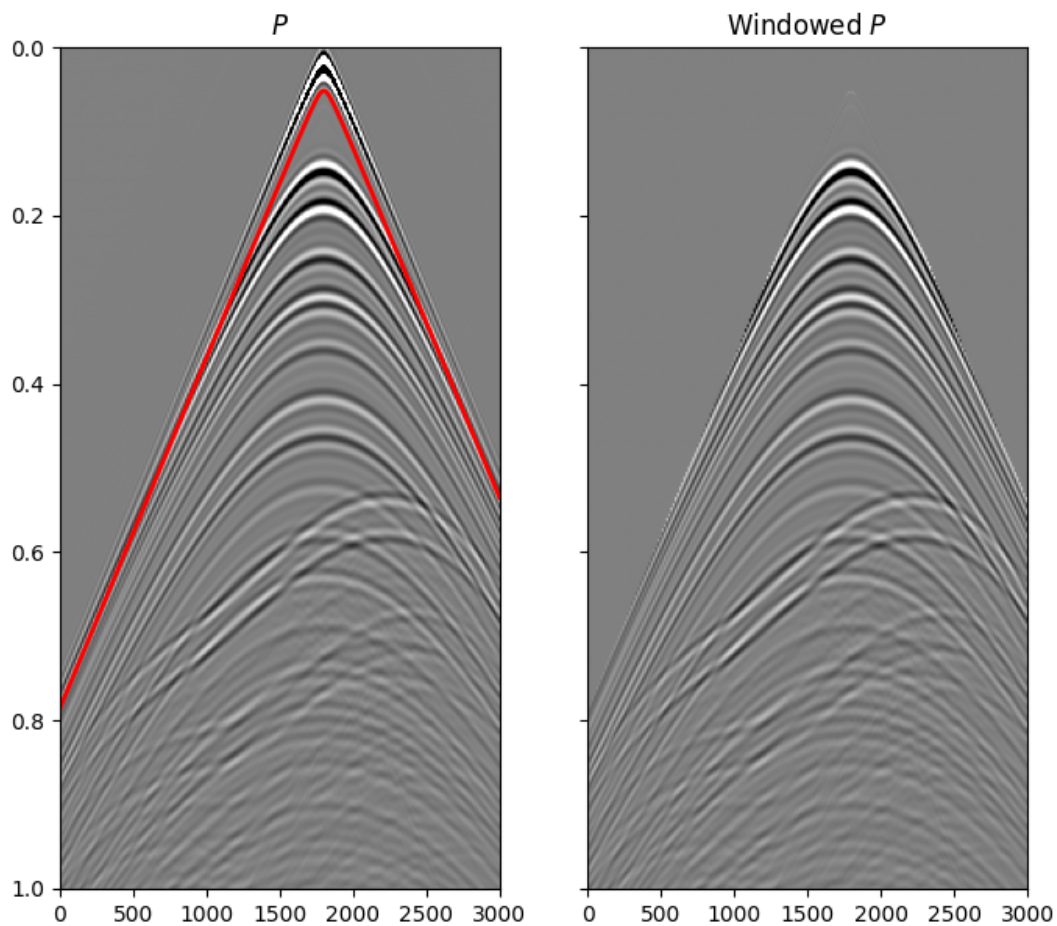
(continues on next page)

(continued from previous page)

```

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

```

pup, pdown = pyllops.waveeqprocessing.Deghosting(
    p.T,

```

(continues on next page)

(continued from previous page)

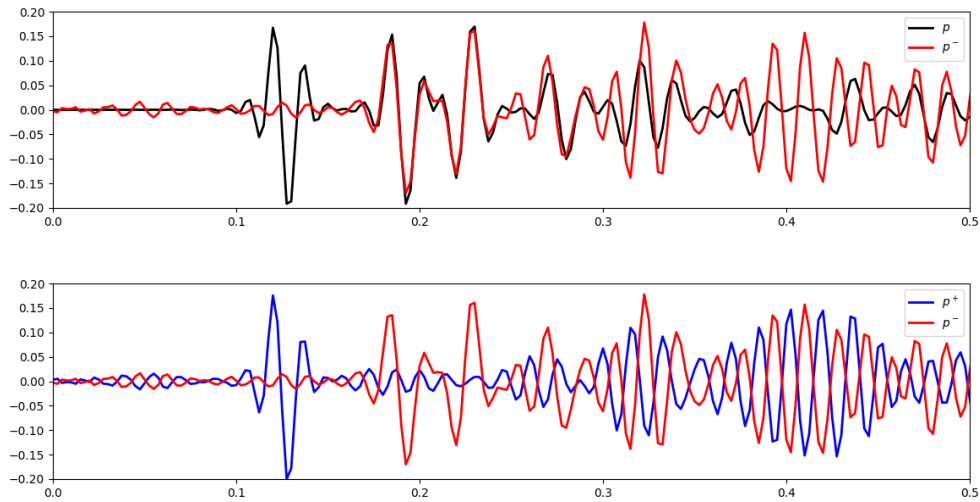
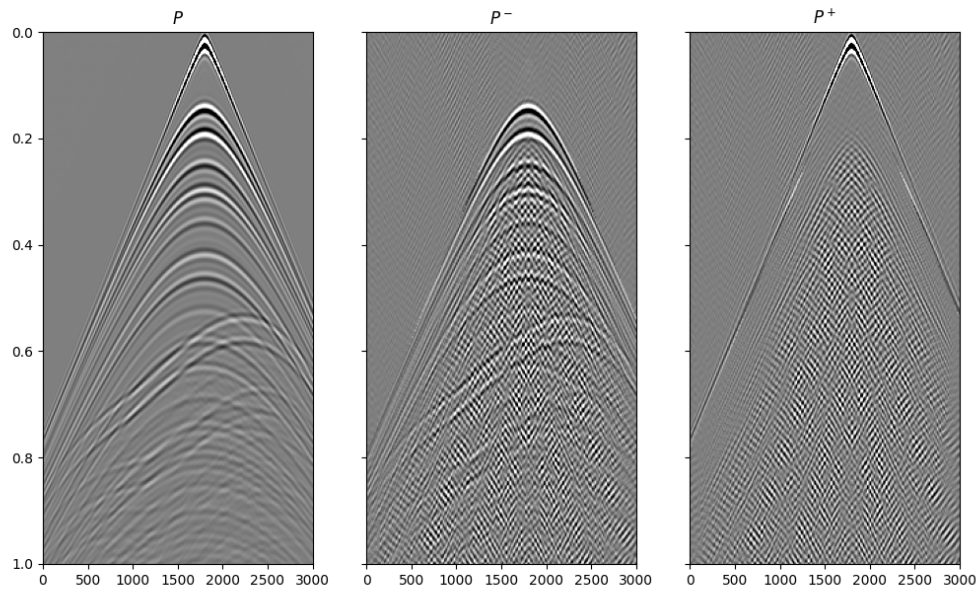
```

    nt,
    nr,
    dt,
    dr,
    vel_sep,
    r[1, 0] + dz,
    win=win,
    npad=5,
    ntaper=11,
    solver=lsqr,
    dottest=False,
    dtype="complex128",
    **dict(damp=1e-10, iter_lim=60)
)

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()

```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aed334438>
```

To see more examples head over to the following notebook: [notebook1](#).

Total running time of the script: (0 minutes 6.865 seconds)

3.5.14 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.waveeqprocessing.UpDownComposition2D` 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.

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

import pylops
from pylops.utils.seismicevents import hyperbolic2d, makeaxis
from pylops.utils.wavelets import ricker

np.random.seed(0)
plt.close("all")
```

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

```
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([1400.0, 1500.0, 2000.0])
amp = np.array([1.0, -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

```
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, FFTop.f2, indexing="ij")
k = F / vel_sep
Kz = np.sqrt((k ** 2 - Kx ** 2).astype(np.complex128))
Kz[np.isnan(Kz)] = 0
OBL = rho_sep * (np.abs(F) / Kz)
OBL[Kz == 0] = 0
```

(continues on next page)

(continued from previous page)

```

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 * 100.0,
    ntaper=ntaper,
    dtype="complex128",
)

# wavefield modelling
d = UPop * np.concatenate((p_plus.ravel(), p_minus.ravel())).ravel()
d = np.real(d.reshape(2 * par["nx"], par["nt"]))
p, vz = d[:, par["nx"]], d[par["nx"] :]

# obliquity scaled vz
VZ = FFTop * vz.ravel()
VZ = VZ.reshape(nfft, nfft)

VZ_obl = OBL * VZ
vz_obl = FFTop.H * VZ_obl.ravel()
vz_obl = np.real(vz_obl.reshape(par["nx"], par["nt"]))

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")

```

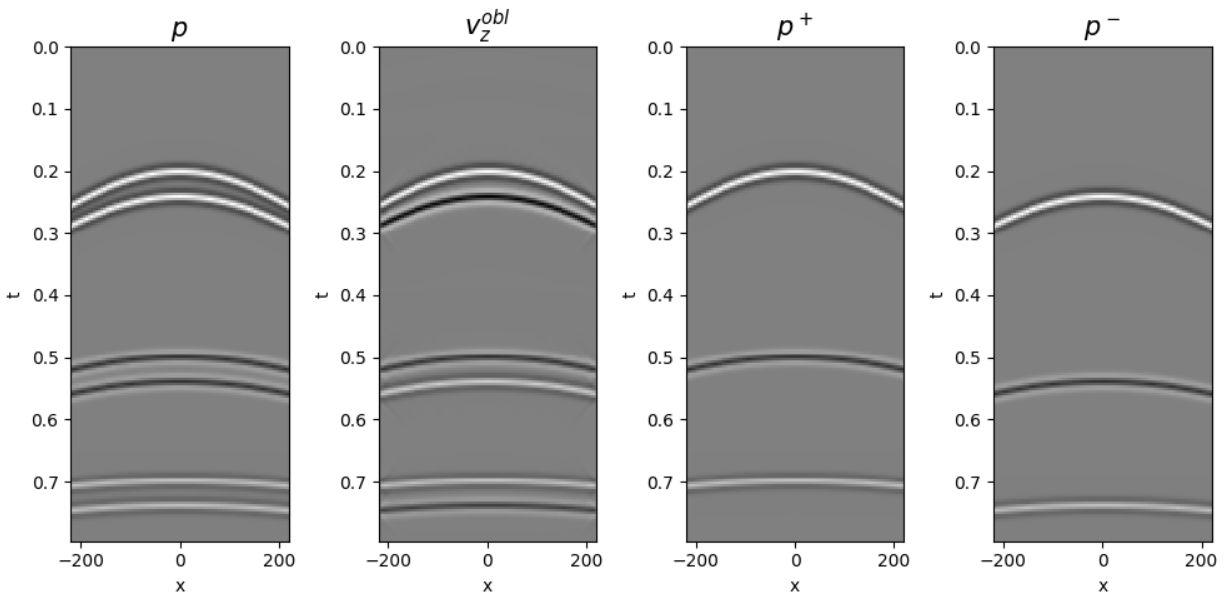
(continues on next page)

(continued from previous page)

```

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",
    interpolation="nearest",
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
    vmin=-1,
    vmax=1,
)
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

```

pup_sep, pdown_sep = pyllops.waveeqprocessing.WavefieldDecomposition(
    p,
    vz,
    par["nt"],
    par["nx"],
)

```

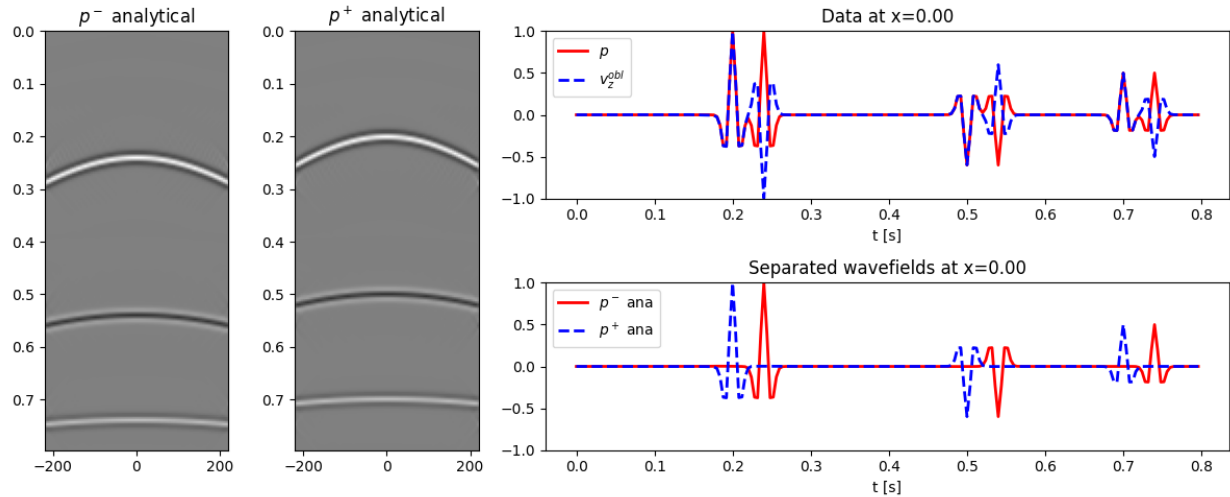
(continues on next page)

(continued from previous page)

```

    par["dt"],
    par["dx"],
    rho_sep,
    vel_sep,
    nffts=(nfft, nfft),
    kind="analytical",
    critical=critical * 100,
    ntaper=ntaper,
    dtype="complex128",
)
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)
axs0.imshow(
    pup_sep.T, cmap="gray", vmin=-1, vmax=1, extent=(x.min(), x.max(), t.max(), t.
    ↪min())
)
axs0.set_title(r"$p^-$ analytical")
axs0.axis("tight")
axs1.imshow(
    pdown_sep.T,
    cmap="gray",
    vmin=-1,
    vmax=1,
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs1.set_title(r"$p^+$ analytical")
axs1.axis("tight")
axs2.plot(t, p[par["nx"] // 2], "r", lw=2, label=r"$p$")
axs2.plot(t, vz_obl[par["nx"] // 2], "--b", lw=2, label=r"$v_z^{\text{obl}}$")
axs2.set_ylim(-1, 1)
axs2.set_title("Data at x=%.2f" % x[par["nx"] // 2])
axs2.set_xlabel("t [s]")
axs2.legend()
axs3.plot(t, pup_sep[par["nx"] // 2], "r", lw=2, label=r"$p^-$ ana")
axs3.plot(t, pdown_sep[par["nx"] // 2], "--b", lw=2, label=r"$p^+$ ana")
axs3.set_title("Separated wavefields at x=%.2f" % x[par["nx"] // 2])
axs3.set_xlabel("t [s]")
axs3.set_ylim(-1, 1)
axs3.legend()
plt.tight_layout()

```



We repeat the same exercise but this time we invert the composition operator `pylops.waveeqprocessing.UpDownComposition2D`

```
pup_inv, pdown_inv = pylops.waveeqprocessing.WavefieldDecomposition(
    p,
    vz,
    par["nt"],
    par["nx"],
    par["dt"],
    par["dx"],
    rho_sep,
    vel_sep,
    nffts=(nfft, nfft),
    kind="inverse",
    critical=critical * 100,
    ntaper=ntaper,
    dtype="complex128",
    **dict(damp=1e-10, iter_lim=20)
)

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)
axs0.imshow(
    pup_inv.T, cmap="gray", vmin=-1, vmax=1, extent=(x.min(), x.max(), t.max(), t.
    ↪min())
)
axs0.set_title(r"$p^-$ inverse")
axs0.axis("tight")
axs1.imshow(
    pdown_inv.T,
    cmap="gray",
    vmin=-1,
    vmax=1,
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs1.set_title(r"$p^+$ inverse")
```

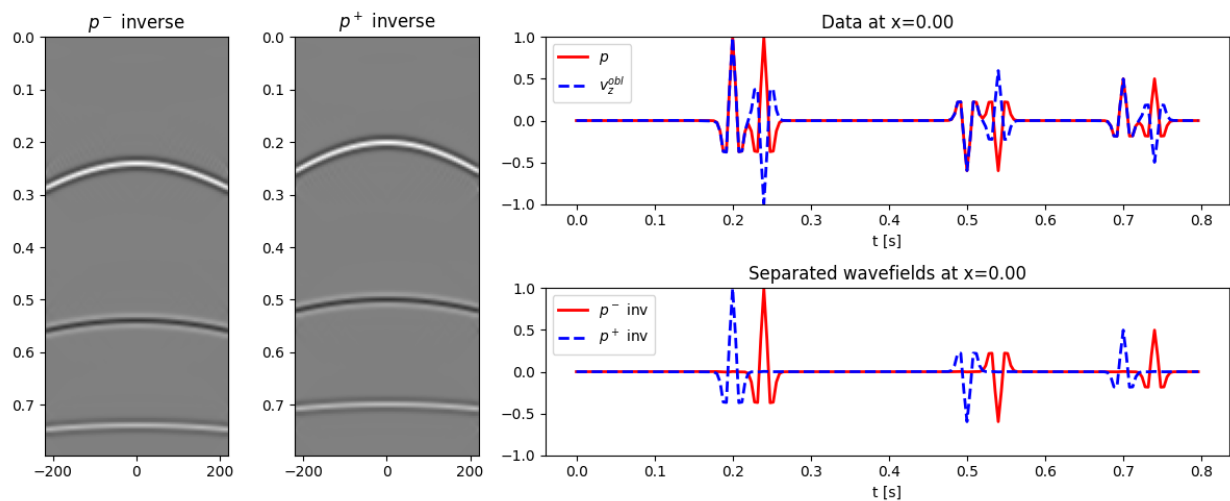
(continues on next page)

(continued from previous page)

```

axs1.axis("tight")
axs2.plot(t, p[par["nx"] // 2], "r", lw=2, label=r"$p$")
axs2.plot(t, vz_obl[par["nx"] // 2], "--b", lw=2, label=r"$v_z^{obl}$")
axs2.set_ylim(-1, 1)
axs2.set_title("Data at x=%.2f" % x[par["nx"] // 2])
axs2.set_xlabel("t [s]")
axs2.legend()
axs3.plot(t, pup_inv[par["nx"] // 2], "r", lw=2, label=r"$p^- inv$")
axs3.plot(t, pdown_inv[par["nx"] // 2], "--b", lw=2, label=r"$p^+ inv$")
axs3.set_title("Separated wavefields at x=%.2f" % x[par["nx"] // 2])
axs3.set_xlabel("t [s]")
axs3.set_ylim(-1, 1)
axs3.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

```

PtoVop = pylops.waveeqprocessing.PressureToVelocity(
    par["nt"],
    par["nx"],
    par["dt"],
    par["dx"],
    rho_sep,
    vel_sep,
    nffts=(nfft, nfft),
    critical=critical * 100.0,
    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(

```

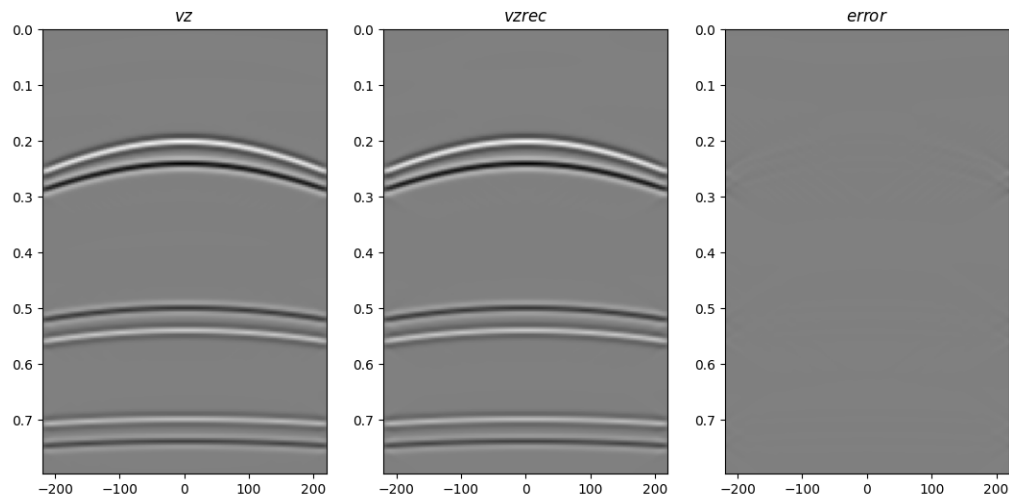
(continues on next page)

(continued from previous page)

```

    vz.T,
    cmap="gray",
    vmin=-1e-6,
    vmax=1e-6,
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[0].set_title(r"$vz$")
axs[0].axis("tight")
axs[1].imshow(
    vz_rec.T, cmap="gray", vmin=-1e-6, vmax=1e-6, extent=(x.min(), x.max(), t[-1],
↪t[0])
)
axs[1].set_title(r"$vz \text{ rec}$")
axs[1].axis("tight")
axs[2].imshow(
    vz.T - vz_rec.T,
    cmap="gray",
    vmin=-1e-6,
    vmax=1e-6,
    extent=(x.min(), x.max(), t[-1], t[0]),
)
axs[2].set_title(r"$error$")
axs[2].axis("tight")

```



Out:

```
(-220.0, 220.0, 0.796, 0.0)
```

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 7.841 seconds)

3.5.15 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:

$$d(\mathbf{x}_r, \mathbf{x}_s, t) = w(t) * \int_V G(\mathbf{x}, \mathbf{x}_s, t) G(\mathbf{x}_r, \mathbf{x}, t) m(\mathbf{x}) d\mathbf{x}$$

where $m(\mathbf{x})$ is the reflectivity at every location in the subsurface, $G(\mathbf{x}, \mathbf{x}_s, t)$ and $G(\mathbf{x}_r, \mathbf{x}, t)$ 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.

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

```
# Velocity Model
nx, nz = 81, 60
dx, dz = 4, 4
x, z = np.arange(nx) * dx, np.arange(nz) * dz
v0 = 1000 # initial velocity
kv = 0.0 # gradient
vel = np.outer(np.ones(nx), v0 + kv * z)

# Reflectivity Model
refl = np.zeros((nx, nz))
refl[:, 30] = -1
refl[:, 50] = 0.5

# Receivers
nr = 11
rx = np.linspace(10 * dx, (nx - 10) * dx, nr)
rz = 20 * np.ones(nr)
recs = np.vstack((rx, rz))
dr = recs[0, 1] - recs[0, 0]

# Sources
ns = 10
sx = np.linspace(dx * 10, (nx - 10) * dx, ns)
sz = 10 * np.ones(ns)
sources = np.vstack((sx, sz))
```

(continues on next page)

(continued from previous page)

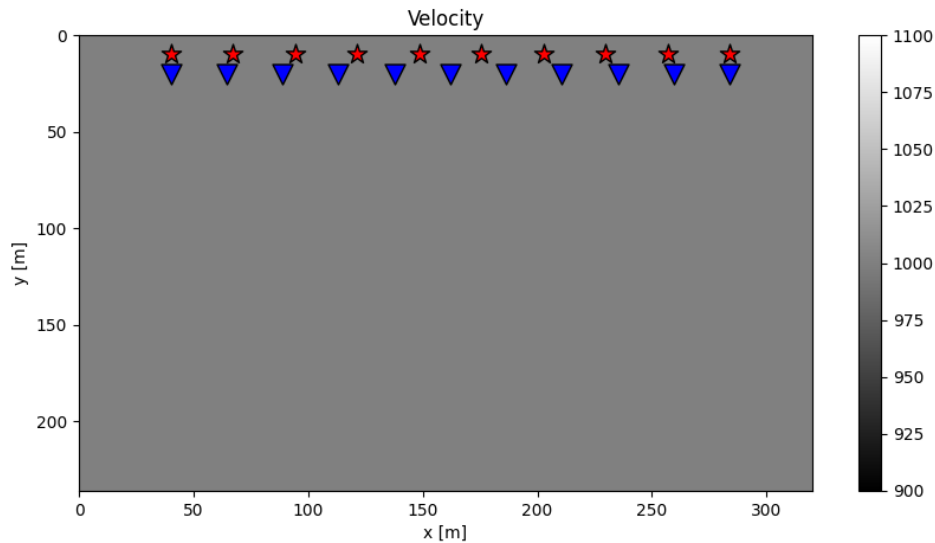
```

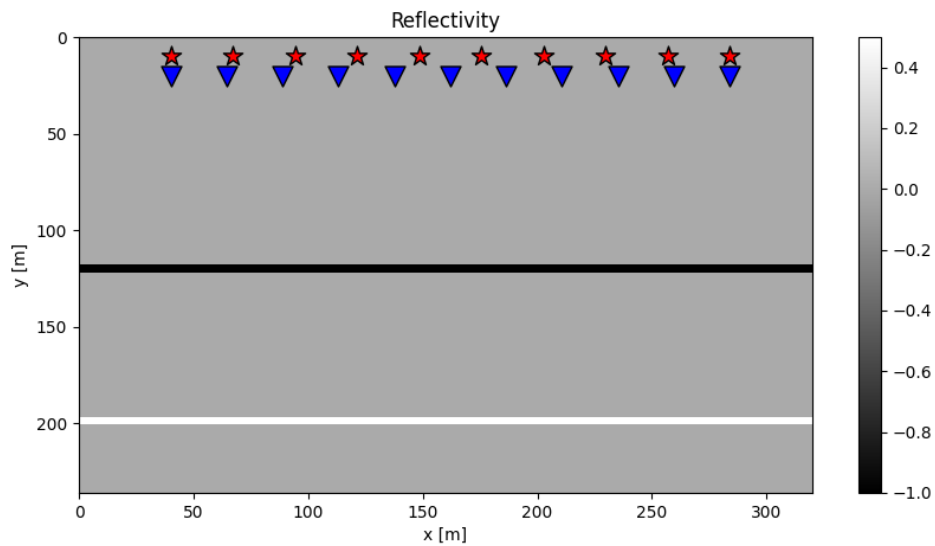
ds = sources[0, 1] - sources[0, 0]

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])

```





Out:

```
(0.0, 320.0)
```

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).

```
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)

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()
```

(continues on next page)

(continued from previous page)

```

dinv = dinv.reshape(ns, nr, nt)

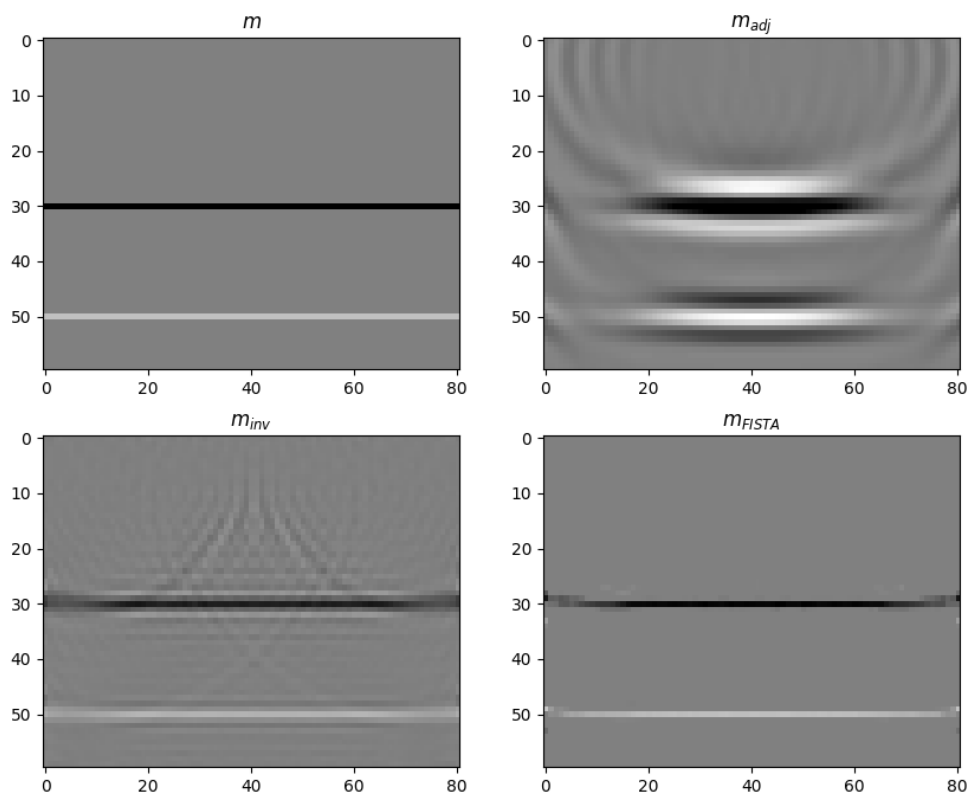
dinv_sparse = lsm.Demop * minv_sparse.ravel()
dinv_sparse = dinv_sparse.reshape(ns, nr, nt)

# 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"$m_{FISTA}$")

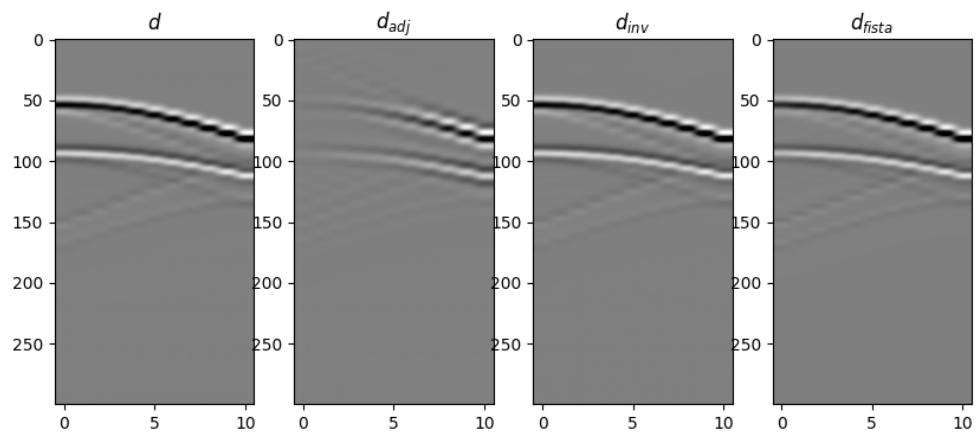
fig, axs = plt.subplots(1, 4, figsize=(10, 4))
axs[0].imshow(d[0, :, :300].T, cmap="gray", vmin=-d.max(), vmax=d.max())
axs[0].set_title(r"$d$")
axs[0].axis("tight")
axs[1].imshow(dadj[0, :, :300].T, cmap="gray", vmin=-dadj.max(), vmax=dadj.max())
axs[1].set_title(r"$d_{adj}$")
axs[1].axis("tight")
axs[2].imshow(dinv[0, :, :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")

fig, axs = plt.subplots(1, 4, figsize=(10, 4))
axs[0].imshow(d[ns // 2, :, :300].T, cmap="gray", vmin=-d.max(), vmax=d.max())
axs[0].set_title(r"$d$")
axs[0].axis("tight")
axs[1].imshow(dadj[ns // 2, :, :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[ns // 2, :, :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[ns // 2, :, :300].T, cmap="gray", vmin=-d.max(), vmax=d.
↪max())
axs[3].set_title(r"$d_{fista}$")
axs[3].axis("tight")

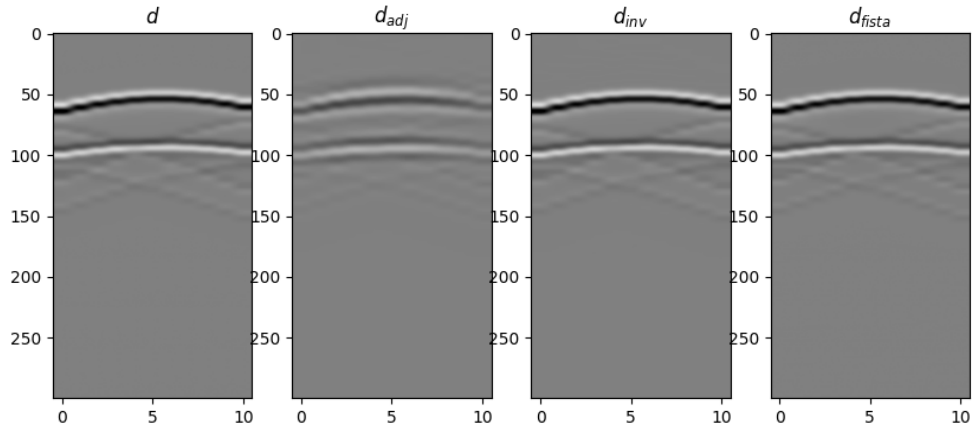
```



•



•



Out:

```
(-0.5, 10.5, 299.5, -0.5)
```

This was just a short teaser, for a more advanced set of examples of 2D and 3D traveltime-based LSM head over to [this notebook](#).

Total running time of the script: (0 minutes 4.635 seconds)

3.5.16 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.

```
import matplotlib.pyplot as plt

# sphinx_gallery_thumbnail_number = 2
import numpy as np
from numba import jit

import pylops

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:

$$t(r, \theta; x) = \tan(90 - \theta)x + \frac{r}{\sin(\theta)}$$

where θ 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.

```
@jit(nopython=True)
def radoncurve(x, r, theta):
    return (
```

(continues on next page)

(continued from previous page)

```

        (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").T
x = x / x.max()
nx, ny = x.shape

ntheta = 150
theta = np.linspace(0.0, 180.0, ntheta, endpoint=False)

Rlop = pylops.signalprocessing.Radon2D(
    np.arange(ny),
    np.arange(nx),
    theta,
    kind=radoncurve,
    centeredh=True,
    interp=False,
    engine="numba",
    dtype="float64",
)

y = Rlop.H * x.ravel()
y = y.reshape(ntheta, ny)

```

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

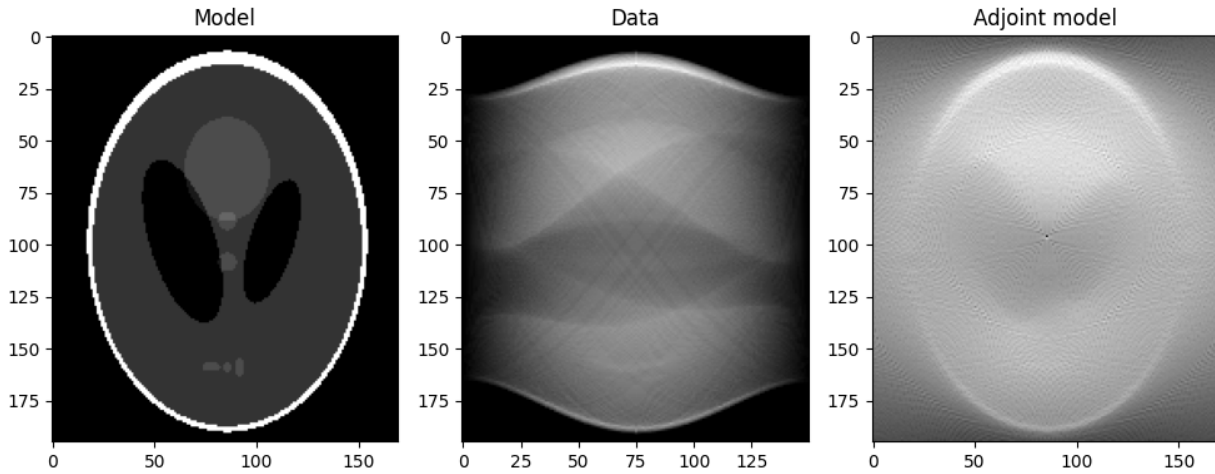
This is the first step of a common reconstruction technique, named filtered back-projection, which simply applies a correction filter in the frequency domain to the adjoint model.

```

xrec = Rlop * y.ravel()
xrec = xrec.reshape(nx, ny)

fig, axs = plt.subplots(1, 3, figsize=(10, 4))
axs[0].imshow(x.T, vmin=0, vmax=1, cmap="gray")
axs[0].set_title("Model")
axs[0].axis("tight")
axs[1].imshow(y.T, cmap="gray")
axs[1].set_title("Data")
axs[1].axis("tight")
axs[2].imshow(xrec.T, cmap="gray")
axs[2].set_title("Adjoint model")
axs[2].axis("tight")
fig.tight_layout()

```

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.

```
Dop = [
    pylops.FirstDerivative(
        ny * nx, dims=(nx, ny), dir=0, edge=True, kind="backward", dtype=np.float64
    ),
    pylops.FirstDerivative(
        ny * nx, dims=(nx, ny), dir=1, edge=True, kind="backward", dtype=np.float64
    ),
]
D2op = pylops.Laplacian(dims=(nx, ny), edge=True, dtype=np.float64)

# L2
xinv_sm = pylops.optimization.leastsquares.RegularizedInversion(
    R Lop.H, [D2op], y.ravel(), epsRs=[1e1], **dict(iter_lim=20)
)
xinv_sm = np.real(xinv_sm.reshape(nx, ny))

# TV
mu = 1.5
lamda = [1.0, 1.0]
niter = 3
niterinner = 4

xinv, niter = pylops.optimization.sparsity.SplitBregman(
    R Lop.H,
    Dop,
    y.ravel(),
    niter,
    niterinner,
    mu=mu,
    epsRLls=lamda,
    tol=1e-4,
    tau=1.0,
    show=False,
    **dict(iter_lim=20, damp=1e-2)
)
xinv = np.real(xinv.reshape(nx, ny))
```

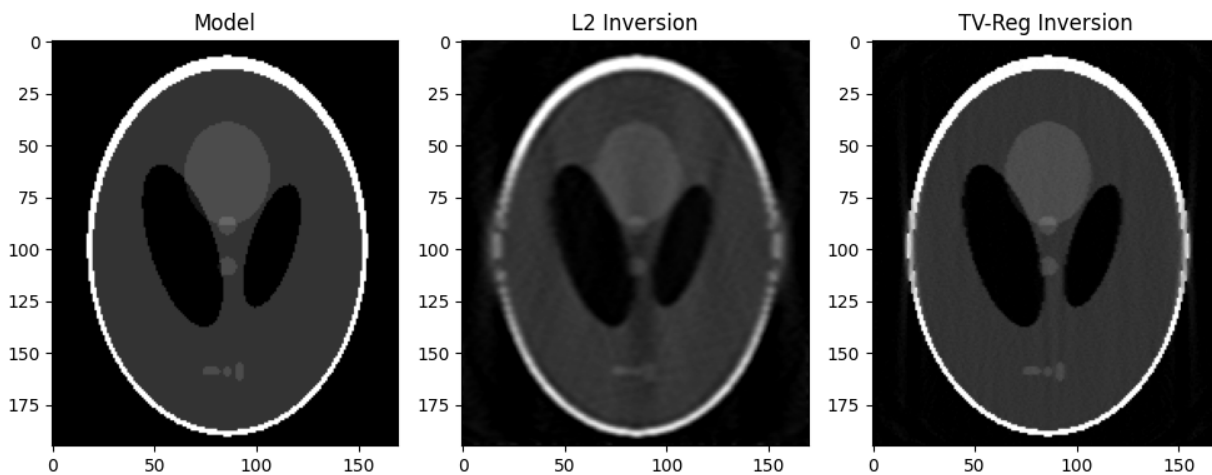
(continues on next page)

(continued from previous page)

```

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

```



Total running time of the script: (0 minutes 13.089 seconds)

3.5.17 17. Real/Complex Inversion

In this tutorial we will discuss two equivalent approaches to the solution of inverse problems with real-valued model vector and complex-valued data vector. In other words, we consider a modelling operator $\mathbf{A} : \mathbb{F}^m \rightarrow \mathbb{C}^n$ (which could be the case for example for the real FFT).

Mathematically speaking, this problem can be solved equivalently by inverting the complex-valued problem:

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

or the real-valued augmented system

$$\text{ReIm} \begin{bmatrix} \mathbf{y} \\ \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{A} \\ \mathbf{A} \end{bmatrix} \mathbf{x}$$

Whilst we already know how to solve the first problem, let's see how we can solve the second one by taking advantage of the real method of the `pylops.LinearOperator` object. We will also wrap our linear operator into a `pylops.MemoizeOperator` which remembers the last N model and data vectors and by-passes the computation of the forward and/or adjoint pass whenever the same pair reappears. This is very useful in our case when we want to compute the real and the imag components of

```

import matplotlib.pyplot as plt
import numpy as np

```

(continues on next page)

(continued from previous page)

```
import pylops

plt.close("all")
np.random.seed(0)
```

To start we create the forward problem

```
n = 5
x = np.arange(n) + 1.0

# make A
Ar = np.random.normal(0, 1, (n, n))
Ai = np.random.normal(0, 1, (n, n))
A = Ar + 1j * Ai
Aop = pylops.MatrixMult(A, dtype=np.complex128)
y = Aop @ x
```

Let's check we can solve this problem using the first formulation

```
Alop = Aop.toreal(forw=False, adj=True)
xinv = Alop.div(y)

print("xinv=%s\n" % xinv)
```

Out:

```
xinv=[1. 2. 3. 4. 5.]
```

Let's now see how we formulate the second problem

```
Amop = pylops.MemoizeOperator(Aop, max_neval=10)
Arop = Amop.toreal()
Aiop = Amop.toimag()

Alop = pylops.VStack([Arop, Aiop])
y1 = np.concatenate([np.real(y), np.imag(y)])
xinv1 = np.real(Alop.div(y1))

print("xinv1=%s\n" % xinv1)
```

Out:

```
xinv1=[1. 2. 3. 4. 5.]
```

Total running time of the script: (0 minutes 0.010 seconds)

3.6 Gallery

Below is a gallery of examples which use PyLops operators and utilities.

3.6.1 1D Smoothing

This example shows how to use the `pylops.Smoothing1D` operator to smooth an input signal along a given axis.

Derivative (or roughening) operators are generally used *regularization* in inverse problems. Smoothing has the opposite effect of roughening and it can be employed as *preconditioning* in inverse problems.

A smoothing operator is a simple compact filter on length n_{smooth} and each elements is equal to $1/n_{smooth}$.

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

import pylops

plt.close("all")
```

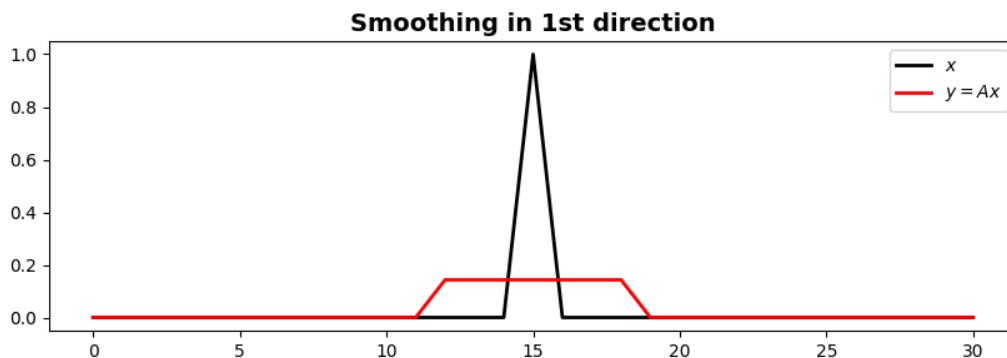
Define the input parameters: number of samples of input signal (N) and length of the smoothing filter regression coefficients (n_{smooth}). In this first case the input signal is one at the center and zero elsewhere.

```
N = 31
nsmooth = 7
x = np.zeros(N)
x[int(N / 2)] = 1

Sop = pylops.Smoothing1D(nsmooth=nsmooth, dims=[N], dtype="float32")

y = Sop * x
xadj = Sop.H * y

fig, ax = plt.subplots(1, 1, figsize=(10, 3))
ax.plot(x, "k", lw=2, label=r"$x$")
ax.plot(y, "r", lw=2, label=r"$y=Ax$")
ax.set_title("Smoothing in 1st direction", fontsize=14, fontweight="bold")
ax.legend()
```



Out:

```
<matplotlib.legend.Legend object at 0x7f6af6639518>
```

Let's repeat the same exercise with a random signal as input. After applying smoothing, we will also try to invert it.

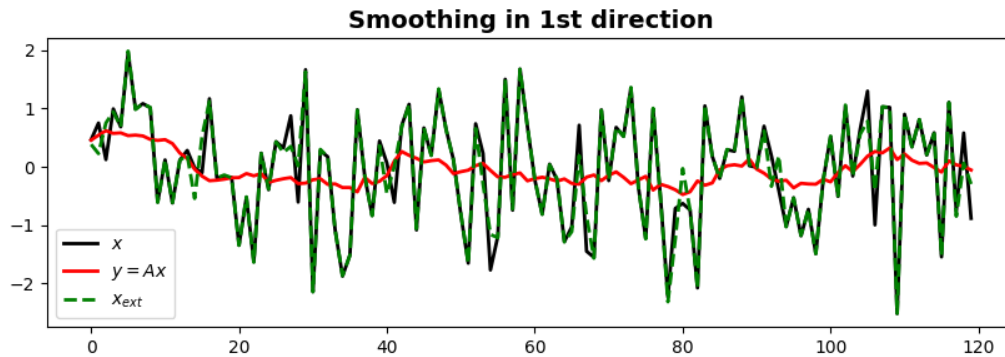
```
N = 120
nsmooth = 13
x = np.random.normal(0, 1, N)
Sop = pylops.Smoothing1D(nsmooth=13, dims=(N), dtype="float32")

y = Sop * x
xest = Sop / y
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(1, 1, figsize=(10, 3))
ax.plot(x, "k", lw=2, label=r"$x$")
ax.plot(y, "r", lw=2, label=r"$y=Ax$")
ax.plot(xest, "--g", lw=2, label=r"$x_{\text{ext}}$")
ax.set_title("Smoothing in 1st direction", fontsize=14, fontweight="bold")
ax.legend()
```



Out:

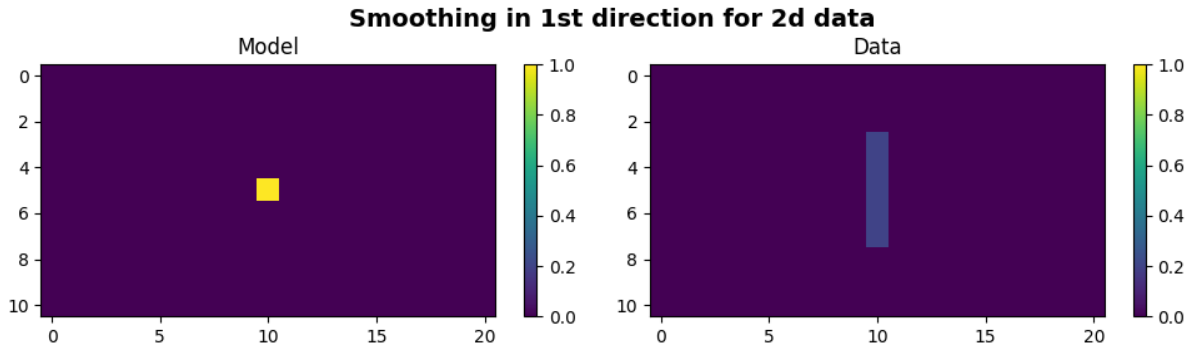
```
<matplotlib.legend.Legend object at 0x7f6af64d61d0>
```

Finally we show that the same operator can be applied to multi-dimensional data along a chosen axis.

```
A = np.zeros((11, 21))
A[5, 10] = 1

Sop = pylops.Smoothing1D(nsmooth=5, dims=(11, 21), dir=0, dtype="float64")
B = np.reshape(Sop * np.ndarray.flatten(A), (11, 21))

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle(
    "Smoothing in 1st direction for 2d data", fontsize=14, fontweight="bold", y=0.95
)
im = axs[0].imshow(A, interpolation="nearest", vmin=0, vmax=1)
axs[0].axis("tight")
axs[0].set_title("Model")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(B, interpolation="nearest", vmin=0, vmax=1)
axs[1].axis("tight")
axs[1].set_title("Data")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)
```



Total running time of the script: (0 minutes 0.921 seconds)

3.6.2 1D, 2D and 3D Sliding

This example shows how to use the `pylops.signalprocessing.Sliding1D`, `pylops.signalprocessing.Sliding2D` and `pylops.signalprocessing.Sliding3D` operators to perform repeated transforms over small strides of a 1-, 2- or 3-dimensional array.

For the 1-d case, the transform that we apply in this example is the `pylops.signalprocessing.FFT`.

For the 2- and 3-d cases, the transform that we apply in this example is the `pylops.signalprocessing.Radon2D` (and `pylops.signalprocessing.Radon3D`) but this operator has been design to allow a variety of transforms as long as they operate with signals that are 2 or 3-dimensional in nature, respectively.

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

import pylops

plt.close("all")
```

Let's start by creating a 1-dimensional array of size n_t and create a sliding operator to compute its transformed representation.

```
nwins = 4
nwin = 26
nover = 3
nop = 64
dim = (nop + 2) // 2 * nwins
dimd = nwin * nwins - 3 * nover

t = np.arange(dimd) * 0.004
data = np.sin(2 * np.pi * 20 * t)

Op = pylops.signalprocessing.FFT(nwin, nfft=nop, real=True)
Slid = pylops.signalprocessing.Sliding1D(
    Op.H, dim, dimd, nwin, nover, tapertype=None, design=False
)

x = Slid.H * data.ravel()
```

We now create a similar operator but we also add a taper to the overlapping parts of the patches and use it to reconstruct the original signal. This is done by simply using the adjoint of the `pylops.signalprocessing.Sliding1D` operator. Note that for non- orthogonal operators, this must be replaced by an inverse.

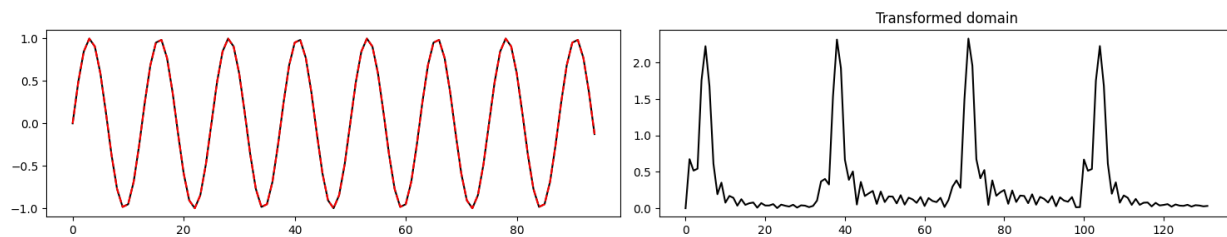
```

Slid = pylops.signalprocessing.Sliding1D(
    Op.H, dim, dimd, nwin, nover, tapertype="cosine", design=False
)

reconstructed_data = Slid * x.ravel()

fig, axs = plt.subplots(1, 2, figsize=(15, 3))
axs[0].plot(data, "k", label="Data")
axs[0].plot(reconstructed_data, "--r", label="Rec Data")
axs[1].set_title("Original domain")
axs[1].plot(np.abs(x), "k")
axs[1].set_title("Transformed domain")
plt.tight_layout()

```



Out:

```

/home/docs/checkouts/readthedocs.org/user_builds/pylops/envs/v1.18.2/lib/python3.6/
site-packages/numpy/core/_asarray.py:83: ComplexWarning: Casting complex values to
real discards the imaginary part
    return array(a, dtype, copy=False, order=order)

```

We now create a 2-dimensional array of size $n_x \times n_t$ composed of 3 parabolic events

```

par = {"ox": -140, "dx": 2, "nx": 140, "ot": 0, "dt": 0.004, "nt": 200, "f0": 20}

v = 1500
t0 = [0.2, 0.4, 0.5]
px = [0, 0, 0]
pxx = [1e-5, 5e-6, 1e-20]
amp = [1.0, -2, 0.5]

# Create axis
t, t2, x, y = pylops.utils.seismicevents.makeaxis(par)

# Create wavelet
wav = pylops.utils.wavelets.ricker(t[:41], f0=par["f0"])[0]

# Generate model
_, data = pylops.utils.seismicevents.parabolic2d(x, t, t0, px, pxx, amp, wav)

```

We want to divide this 2-dimensional data into small overlapping patches in the spatial direction and apply the adjoint of the `pylops.signalprocessing.Radon2D` operator to each patch. This is done by simply using the adjoint of the `pylops.signalprocessing.Sliding2D` operator

```

nwins = 5
winsize = 36
overlap = 10
npx = 61

```

(continues on next page)

(continued from previous page)

```

px = np.linspace(-5e-3, 5e-3, npx)

dimsd = data.shape
dims = (nwins * npx, par["nt"])

# Sliding window transform without taper
Op = pylops.signalprocessing.Radon2D(
    t,
    np.linspace(-par["dx"] * winsize // 2, par["dx"] * winsize // 2, winsize),
    px,
    centeredh=True,
    kind="linear",
    engine="numba",
)
Slid = pylops.signalprocessing.Sliding2D(
    Op, dims, dimsd, winsize, overlap, tapertype=None
)

radon = Slid.H * data.ravel()
radon = radon.reshape(dims)

```

We now create a similar operator but we also add a taper to the overlapping parts of the patches.

```

Slid = pylops.signalprocessing.Sliding2D(
    Op, dims, dimsd, winsize, overlap, tapertype="cosine"
)

reconstructed_data = Slid * radon.ravel()
reconstructed_data = reconstructed_data.reshape(dimsd)

```

We will see that our reconstructed signal presents some small artifacts. This is because we have not inverted our operator but simply applied the adjoint to estimate the representation of the input data in the Radon domain. We can do better if we use the inverse instead.

```

radoninv = pylops.LinearOperator(Slid, explicit=False).div(data.ravel(), niter=10)
reconstructed_datainv = Slid * radoninv.ravel()

radoninv = radoninv.reshape(dims)
reconstructed_datainv = reconstructed_datainv.reshape(dimsd)

```

Let's finally visualize all the intermediate results as well as our final data reconstruction after inverting the *pylops.signalprocessing.Sliding2D* operator.

```

fig, axs = plt.subplots(2, 3, sharey=True, figsize=(12, 10))
im = axs[0][0].imshow(data.T, cmap="gray")
axs[0][0].set_title("Original data")
plt.colorbar(im, ax=axs[0][0])
axs[0][0].axis("tight")
im = axs[0][1].imshow(radon.T, cmap="gray")
axs[0][1].set_title("Adjoint Radon")
plt.colorbar(im, ax=axs[0][1])
axs[0][1].axis("tight")
im = axs[0][2].imshow(reconstructed_data.T, cmap="gray")
axs[0][2].set_title("Reconstruction from adjoint")
plt.colorbar(im, ax=axs[0][2])
axs[0][2].axis("tight")

```

(continues on next page)

(continued from previous page)

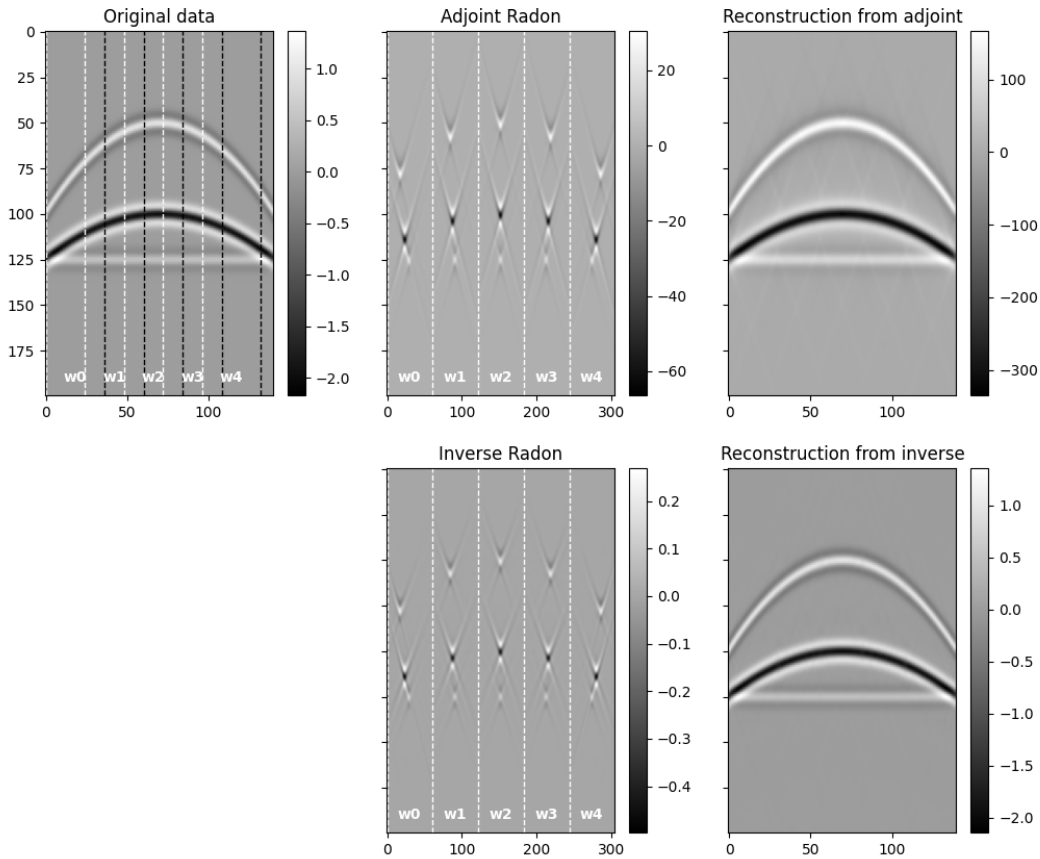
```

axs[1][0].axis("off")
im = axs[1][1].imshow(radoninv.T, cmap="gray")
axs[1][1].set_title("Inverse Radon")
plt.colorbar(im, ax=axs[1][1])
axs[1][1].axis("tight")
im = axs[1][2].imshow(reconstructed_datainv.T, cmap="gray")
axs[1][2].set_title("Reconstruction from inverse")
plt.colorbar(im, ax=axs[1][2])
axs[1][2].axis("tight")

for i in range(0, 114, 24):
    axs[0][0].axvline(i, color="w", lw=1, ls="--")
    axs[0][0].axvline(i + winsize, color="k", lw=1, ls="--")
    axs[0][0].text(
        i + winsize // 2,
        par["nt"] - 10,
        "w" + str(i // 24),
        ha="center",
        va="center",
        weight="bold",
        color="w",
    )

for i in range(0, 305, 61):
    axs[0][1].axvline(i, color="w", lw=1, ls="--")
    axs[0][1].text(
        i + npix // 2,
        par["nt"] - 10,
        "w" + str(i // 61),
        ha="center",
        va="center",
        weight="bold",
        color="w",
    )
    axs[1][1].axvline(i, color="w", lw=1, ls="--")
    axs[1][1].text(
        i + npix // 2,
        par["nt"] - 10,
        "w" + str(i // 61),
        ha="center",
        va="center",
        weight="bold",
        color="w",
    )

```



We notice two things, i) provided small enough patches and a transform that can explain data *locally*, we have been able to reconstruct our original data almost to perfection. ii) inverse is better than adjoint as expected as the adjoint does not only introduce small artifacts but also does not respect the original amplitudes of the data.

An appropriate transform alongside with a sliding window approach will result in a very good approach for interpolation (or *regularization*) of irregularly sampled seismic data.

Finally, we do the same for a 3-dimensional array of size $n_y \times n_x \times n_t$ composed of 3 hyperbolic events

```
par = {
    "oy": -15,
    "dy": 2,
    "ny": 14,
    "ox": -18,
    "dx": 2,
    "nx": 18,
    "ot": 0,
    "dt": 0.004,
    "nt": 50,
    "f0": 30,
}
```

(continues on next page)

(continued from previous page)

```

vrms = [200, 200]
t0 = [0.05, 0.1]
amp = [1.0, -2]

# Create axis
t, t2, x, y = pyllops.utils.seismicevents.makeaxis(par)

# Create wavelet
wav = pyllops.utils.wavelets.ricker(t[:41], f0=par["f0"])[0]

# Generate model
_, data = pyllops.utils.seismicevents.hyperbolic3d(x, y, t, t0, vrms, vrms, amp, wav)

# Sliding window plan
nwins = (4, 5)
winsize = (5, 6)
overlap = (2, 3)
npx = 21
px = np.linspace(-5e-3, 5e-3, npx)

dimsd = data.shape
dims = (nwins[0] * npx, nwins[1] * npx, par["nt"])

# Sliding window transform without taper
Op = pyllops.signalprocessing.Radon3D(
    t,
    np.linspace(-par["dy"] * winsize[0] // 2, par["dy"] * winsize[0] // 2,
↪winsize[0]),
    np.linspace(-par["dx"] * winsize[1] // 2, par["dx"] * winsize[1] // 2,
↪winsize[1]),
    px,
    px,
    centeredh=True,
    kind="linear",
    engine="numba",
)
Slid = pyllops.signalprocessing.Sliding3D(
    Op, dims, dimsd, winsize, overlap, (npx, npx), tapertype=None
)

radon = Slid.H * data.ravel()
radon = radon.reshape(nwins[0], nwins[1], npx, npx, par["nt"])

Slid = pyllops.signalprocessing.Sliding3D(
    Op, dims, dimsd, winsize, overlap, (npx, npx), tapertype="cosine", design=False
)

reconstructed_data = Slid * radon.ravel()
reconstructed_data = reconstructed_data.reshape(dimsd)

radoninv = pyllops.LinearOperator(Slid, explicit=False).div(data.ravel(), niter=10)
reconstructed_datainv = Slid * radoninv.ravel()

radoninv = radoninv.reshape(nwins[0], nwins[1], npx, npx, par["nt"])
reconstructed_datainv = reconstructed_datainv.reshape(dimsd)

fig, axs = plt.subplots(2, 3, sharey=True, figsize=(12, 7))

```

(continues on next page)

(continued from previous page)

```

im = axs[0][0].imshow(data[par["ny"] // 2].T, cmap="gray", vmin=-2, vmax=2)
axs[0][0].set_title("Original data")
plt.colorbar(im, ax=axs[0][0])
axs[0][0].axis("tight")
im = axs[0][1].imshow(
    radon[nwins[0] // 2, :, :, npx // 2].reshape(nwins[1] * npx, par["nt"]).T,
    cmap="gray",
    vmin=-25,
    vmax=25,
)
axs[0][1].set_title("Adjoint Radon")
plt.colorbar(im, ax=axs[0][1])
axs[0][1].axis("tight")
im = axs[0][2].imshow(
    reconstructed_data[par["ny"] // 2].T, cmap="gray", vmin=-1000, vmax=1000
)
axs[0][2].set_title("Reconstruction from adjoint")
plt.colorbar(im, ax=axs[0][2])
axs[0][2].axis("tight")
axs[1][0].axis("off")
im = axs[1][1].imshow(
    radoninv[nwins[0] // 2, :, :, npx // 2].reshape(nwins[1] * npx, par["nt"]).T,
    cmap="gray",
    vmin=-0.025,
    vmax=0.025,
)
axs[1][1].set_title("Inverse Radon")
plt.colorbar(im, ax=axs[1][1])
axs[1][1].axis("tight")
im = axs[1][2].imshow(
    reconstructed_datainv[par["ny"] // 2].T, cmap="gray", vmin=-2, vmax=2
)
axs[1][2].set_title("Reconstruction from inverse")
plt.colorbar(im, ax=axs[1][2])
axs[1][2].axis("tight")

fig, axs = plt.subplots(2, 3, figsize=(12, 7))
im = axs[0][0].imshow(data[:, :, 25], cmap="gray", vmin=-2, vmax=2)
axs[0][0].set_title("Original data")
plt.colorbar(im, ax=axs[0][0])
axs[0][0].axis("tight")
im = axs[0][1].imshow(
    radon[nwins[0] // 2, :, :, 25].reshape(nwins[1] * npx, npx).T,
    cmap="gray",
    vmin=-25,
    vmax=25,
)
axs[0][1].set_title("Adjoint Radon")
plt.colorbar(im, ax=axs[0][1])
axs[0][1].axis("tight")
im = axs[0][2].imshow(reconstructed_data[:, :, 25], cmap="gray", vmin=-1000,
    ↪vmax=1000)
axs[0][2].set_title("Reconstruction from adjoint")
plt.colorbar(im, ax=axs[0][2])
axs[0][2].axis("tight")
axs[1][0].axis("off")

```

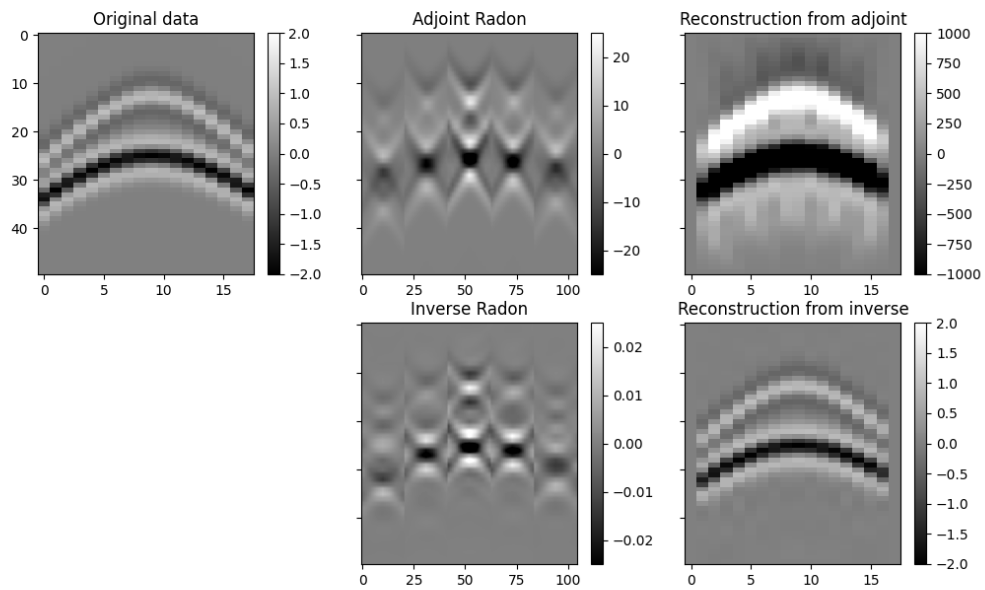
(continues on next page)

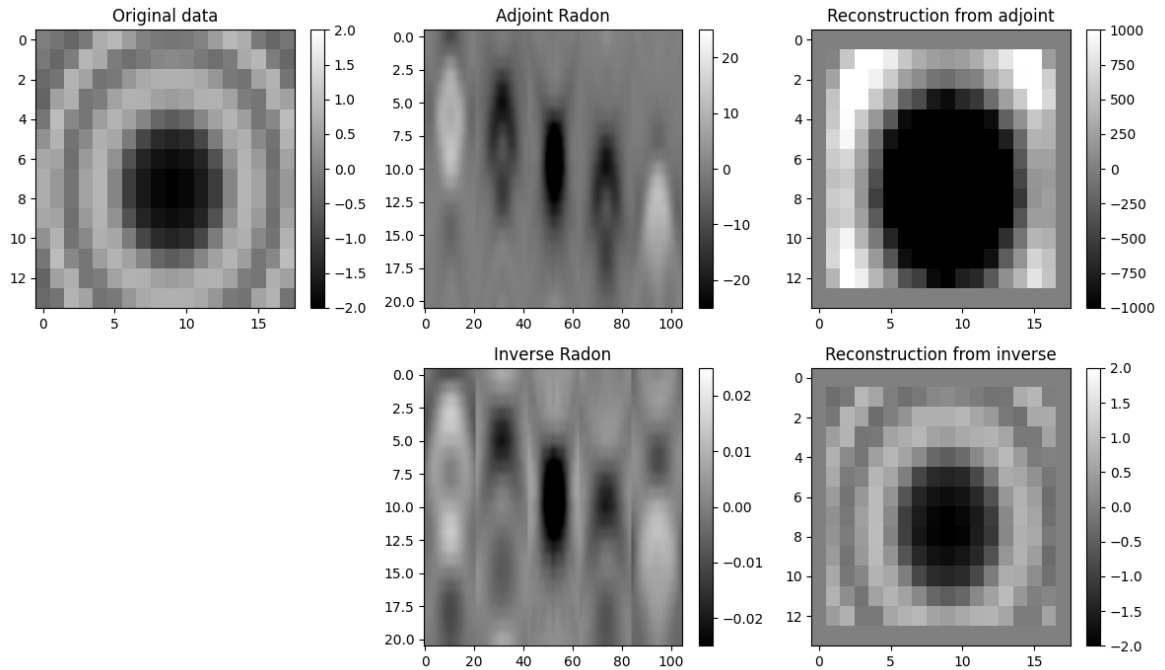
(continued from previous page)

```

im = axs[1][1].imshow(
    radoninv[nwins[0] // 2, :, :, :, 25].reshape(nwins[1] * npx, npx).T,
    cmap="gray",
    vmin=-0.025,
    vmax=0.025,
)
axs[1][1].set_title("Inverse Radon")
plt.colorbar(im, ax=axs[1][1])
axs[1][1].axis("tight")
im = axs[1][2].imshow(reconstructed_datainv[:, :, 25], cmap="gray", vmin=-2, vmax=2)
axs[1][2].set_title("Reconstruction from inverse")
plt.colorbar(im, ax=axs[1][2])
axs[1][2].axis("tight")
plt.tight_layout()

```





Total running time of the script: (0 minutes 9.379 seconds)

3.6.3 2D Patching

This example shows how to use the `pylops.signalprocessing.Patch2D` operator to perform repeated transforms over small patches of a 2-dimensional array. The transform that we apply in this example is the `pylops.signalprocessing.FFT2D` but this operator has been design to allow a variety of transforms as long as they operate with signals that are 2-dimensional in nature, respectively.

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

import pylops

plt.close("all")
```

Let's start by creating an 2-dimensional array of size $n_x \times n_t$ composed of 3 parabolic events

```
par = {"ox": -140, "dx": 2, "nx": 140, "ot": 0, "dt": 0.004, "nt": 200, "f0": 20}

v = 1500
t0 = [0.2, 0.4, 0.5]
px = [0, 0, 0]
pxx = [1e-5, 5e-6, 1e-20]
amp = [1.0, -2, 0.5]

# Create axis
t, t2, x, y = pylops.utils.seismicevents.makeaxis(par)

# Create wavelet
wav = pylops.utils.wavelets.ricker(t[:41], f0=par["f0"])[0]
```

(continues on next page)

(continued from previous page)

```
# Generate model
_, data = pylops.utils.seismicevents.parabolic2d(x, t, t0, px, pxx, amp, wav)
```

We want to divide this 2-dimensional data into small overlapping patches in the spatial direction and apply the adjoint of the `pylops.signalprocessing.FFT2D` operator to each patch. This is done by simply using the adjoint of the `pylops.signalprocessing.Patch2D` operator. Note that for non-orthogonal operators, this must be replaced by an inverse.

```
nwins = (13, 6)
nwin = (20, 34)
nop = (128, 128)
nover = (10, 4)

dimsd = data.shape
dims = (nwins[0] * nop[0], nwins[1] * nop[1])

# Sliding window transform without taper
Op = pylops.signalprocessing.FFT2D(nwin, nffts=nop)
Slid = pylops.signalprocessing.Patch2D(
    Op.H, dims, dimsd, nwin, nover, nop, tapertype=None, design=False
)
fftdata = Slid.H * data.ravel()
```

We now create a similar operator but we also add a taper to the overlapping parts of the patches. We then apply the forward to restore the original signal.

```
Slid = pylops.signalprocessing.Patch2D(
    Op.H, dims, dimsd, nwin, nover, nop, tapertype="hanning", design=False
)

reconstructed_data = Slid * fftdata.ravel()
reconstructed_data = np.real(reconstructed_data.reshape(dimsd))
```

Finally we re-arrange the transformed patches so that we can also display them

```
fftdatareshaped = np.zeros((nop[0] * nwins[0], nop[1] * nwins[1]), dtype=fftdata.
    ↳dtype)

iwin = 1
for ix in range(nwins[0]):
    for it in range(nwins[1]):
        fftdatareshaped[
            ix * nop[0] : (ix + 1) * nop[0], it * nop[1] : (it + 1) * nop[1]
        ] = np.fft.fftshift(
            fftdata[nop[0] * nop[1] * (iwin - 1) : nop[0] * nop[1] * iwin].
    ↳reshape(nop)
        )
        iwin += 1
```

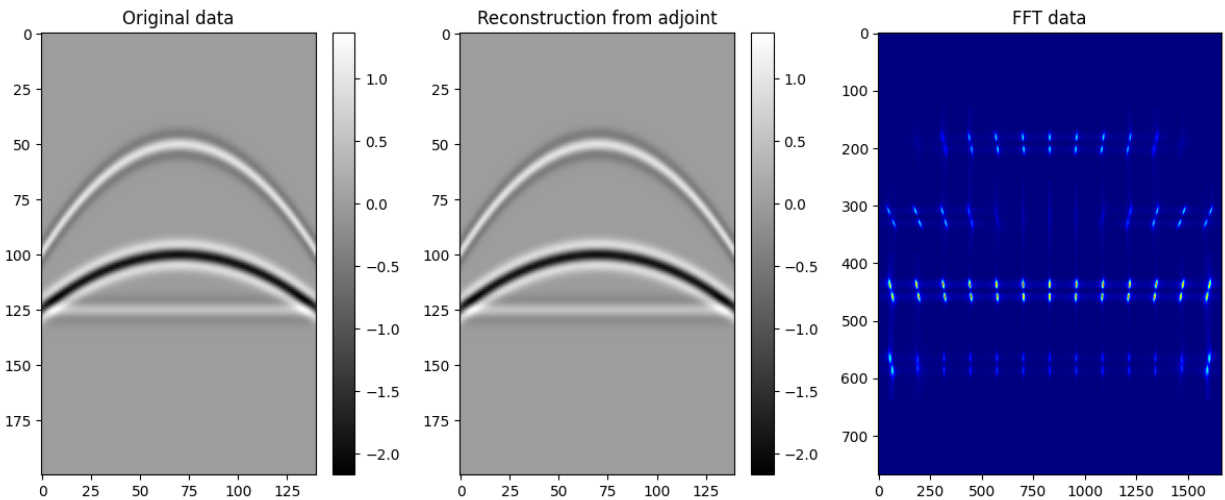
Let's finally visualize all the intermediate results as well as our final data reconstruction after inverting the `pylops.signalprocessing.Sliding2D` operator.

```
fig, axs = plt.subplots(1, 3, figsize=(12, 5))
im = axs[0].imshow(data.T, cmap="gray")
axs[0].set_title("Original data")
```

(continues on next page)

(continued from previous page)

```
plt.colorbar(im, ax=axes[0])
axes[0].axis("tight")
im = axes[1].imshow(reconstructed_data.T, cmap="gray")
axes[1].set_title("Reconstruction from adjoint")
plt.colorbar(im, ax=axes[1])
axes[1].axis("tight")
axes[2].imshow(np.abs(fftdatareshaped).T, cmap="jet")
axes[2].set_title("FFT data")
axes[2].axis("tight")
plt.tight_layout()
```



Total running time of the script: (0 minutes 0.683 seconds)

3.6.4 2D Smoothing

This example shows how to use the `pylops.Smoothing2D` operator to smooth a multi-dimensional input signal along two given axes.

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

import pylops

plt.close("all")
```

Define the input parameters: number of samples of input signal (N and M) and length of the smoothing filter regression coefficients ($n_{smooth,1}$ and $n_{smooth,2}$). In this first case the input signal is one at the center and zero elsewhere.

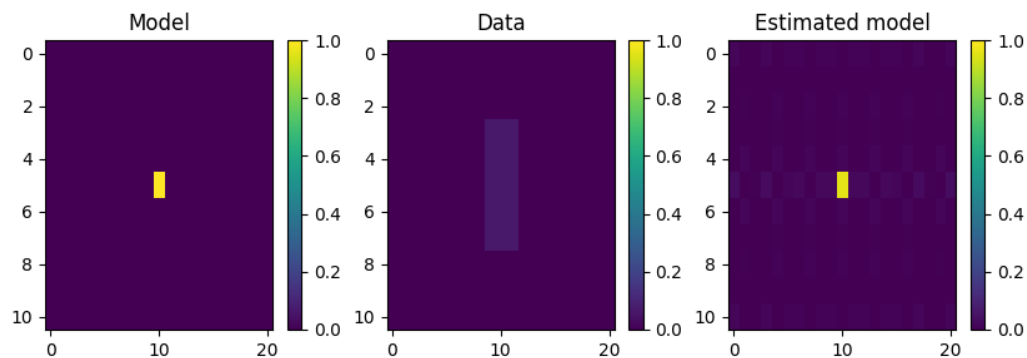
```
N, M = 11, 21
nsmooth1, nsmooth2 = 5, 3
A = np.zeros((N, M))
A[5, 10] = 1

Sop = pylops.Smoothing2D(nsmooth=[nsmooth1, nsmooth2], dims=[N, M], dtype="float64")
B = Sop * A.ravel()
B = np.reshape(B, (N, M))
```


After applying smoothing, we will also try to invert it.

```
Aest = Sop / B.ravel()
Aest = np.reshape(Aest, (N, M))

fig, axs = plt.subplots(1, 3, figsize=(10, 3))
im = axs[0].imshow(A, interpolation="nearest", vmin=0, vmax=1)
axs[0].axis("tight")
axs[0].set_title("Model")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(B, interpolation="nearest", vmin=0, vmax=1)
axs[1].axis("tight")
axs[1].set_title("Data")
plt.colorbar(im, ax=axs[1])
im = axs[2].imshow(Aest, interpolation="nearest", vmin=0, vmax=1)
axs[2].axis("tight")
axs[2].set_title("Estimated model")
plt.colorbar(im, ax=axs[2])
```



Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f6aee32f940>
```

Total running time of the script: (0 minutes 0.349 seconds)

3.6.5 AVO modelling

This example shows how to create pre-stack angle gathers using the `pylops.avo.avo.AVOLinearModelling` operator.

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

import pylops
from pylops.utils.wavelets import ricker

plt.close("all")
np.random.seed(0)
```

Let's start by creating the input elastic property profiles

```

nt0 = 501
dt0 = 0.004
ntheta = 21

t0 = np.arange(nt0) * dt0
thetamin, thetamax = 0, 40
theta = np.linspace(thetamin, thetamax, ntheta)

# Elastic property profiles
vp = 1200 + np.arange(nt0) + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 80,
↪nt0))
vs = 600 + vp / 2 + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 20, nt0))
rho = 1000 + vp + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 30, nt0))
vp[201:] += 500
vs[201:] += 200
rho[201:] += 100

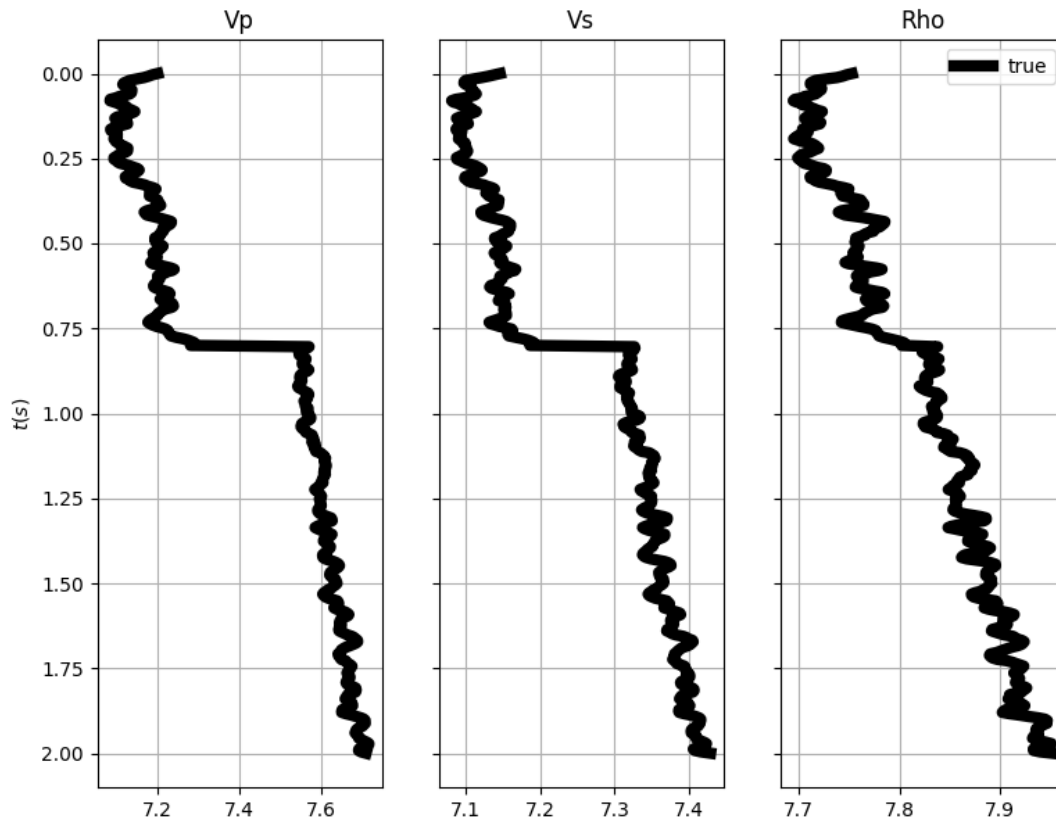
# Wavelet
ntwav = 41
wavoff = 10
wav, twav, wavc = ricker(t0[: ntwav // 2 + 1], 20)
wav_phase = np.hstack((wav[wavoff:], np.zeros(wavoff)))

# vs/vp profile
vsvp = 0.5
vsvp_z = np.linspace(0.4, 0.6, nt0)

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

fig, axs = plt.subplots(1, 3, figsize=(9, 7), sharey=True)
axs[0].plot(m[:, 0], t0, "k", lw=6)
axs[0].set_title("Vp")
axs[0].set_ylabel(r"$t(s)$")
axs[0].invert_yaxis()
axs[0].grid()
axs[1].plot(m[:, 1], t0, "k", lw=6)
axs[1].set_title("Vs")
axs[1].invert_yaxis()
axs[1].grid()
axs[2].plot(m[:, 2], t0, "k", lw=6, label="true")
axs[2].set_title("Rho")
axs[2].invert_yaxis()
axs[2].grid()
axs[2].legend()

```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aedbbbe0>
```

We create now the operators to model the AVO responses for a set of elastic profiles

```
# constant vsvp
PPop_const = pylops.avo.avo.AVONLinearModelling(
    theta, vsvp=vsvp, nt0=nt0, linearization="akirich", dtype=np.float64
)

# depth-variant vsvp
PPop_variant = pylops.avo.avo.AVONLinearModelling(
    theta, vsvp=vsvp_z, linearization="akirich", dtype=np.float64
)
```

We can then apply those operators to the elastic model and create some synthetic reflection responses

```
dPP_const = PPop_const * m.ravel()
dPP_const = dPP_const.reshape(nt0, ntheta)

dPP_variant = PPop_variant * m.ravel()
dPP_variant = dPP_variant.reshape(nt0, ntheta)
```

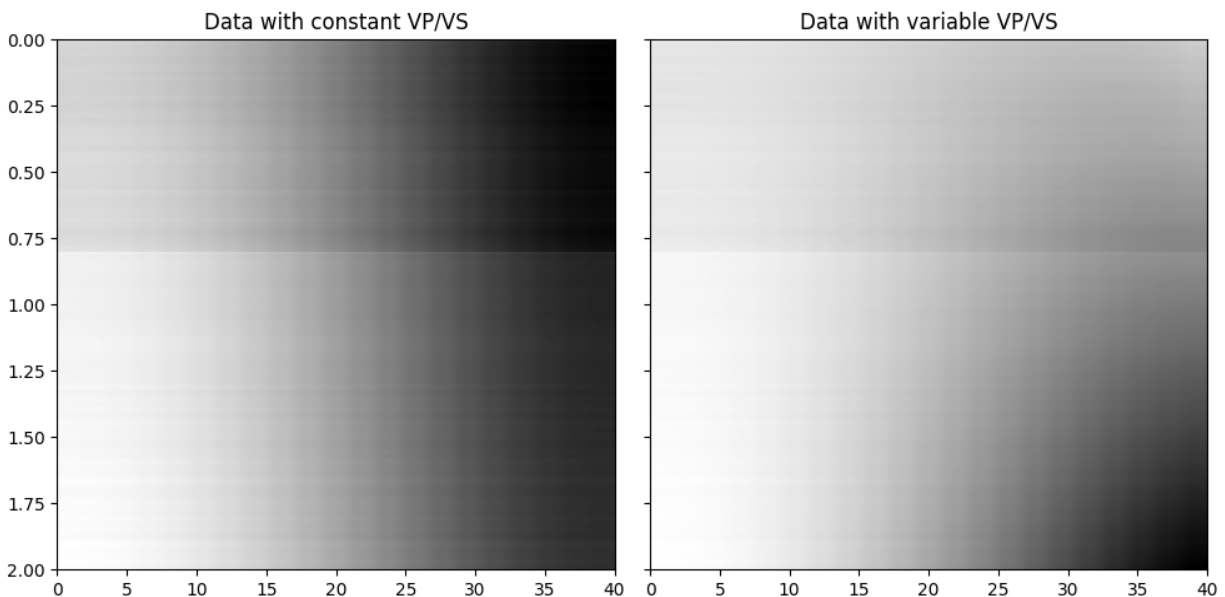
(continues on next page)

(continued from previous page)

```

fig, axs = plt.subplots(1, 2, figsize=(10, 5), sharey=True)
axs[0].imshow(
    dPP_const,
    cmap="gray",
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=dPP_const.min(),
    vmax=dPP_const.max(),
)
axs[0].set_title("Data with constant VP/VS")
axs[0].axis("tight")
axs[1].imshow(
    dPP_variant,
    cmap="gray",
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=dPP_variant.min(),
    vmax=dPP_variant.max(),
)
axs[1].set_title("Data with variable VP/VS")
axs[1].axis("tight")
plt.tight_layout()

```



Finally we can also model the PS response by simply changing the linearization choice as follows

```

PSop = pyllops.avo.avo.AVOLinearModelling(
    theta, vsvp=vsvp, nt0=nt0, linearization="ps", dtype=np.float64
)

```

We can then apply those operators to the elastic model and create some synthetic reflection responses

```

dPS = PSop * m.ravel()
dPS = dPS.reshape(nt0, ntheta)

fig, axs = plt.subplots(1, 2, figsize=(10, 5), sharey=True)
axs[0].imshow(
    dPP_const,

```

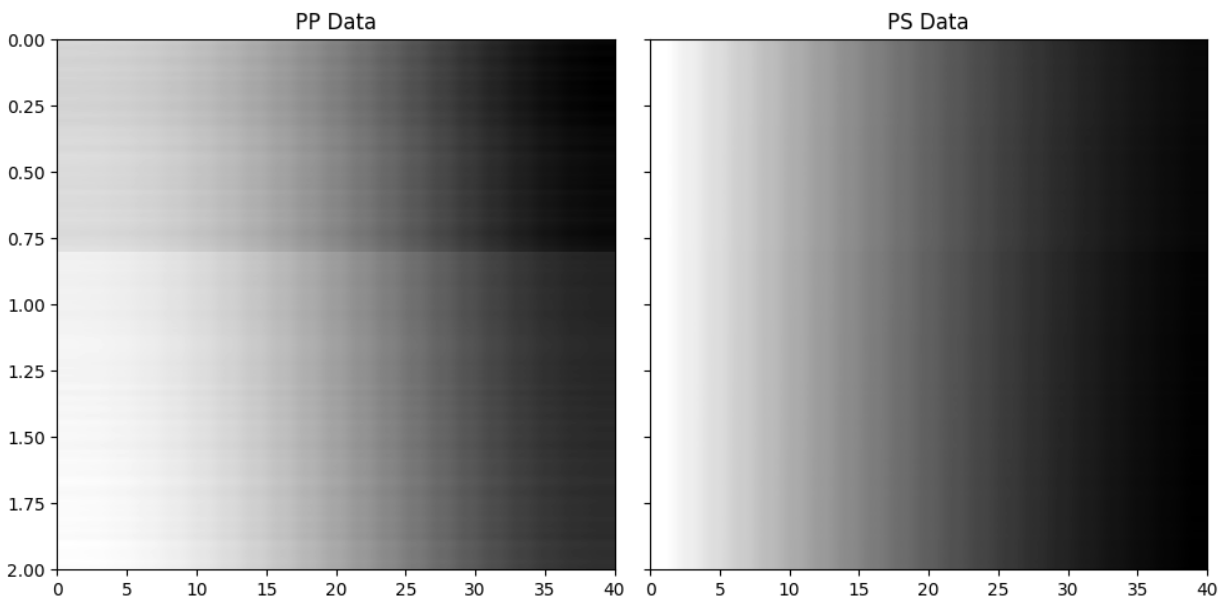
(continues on next page)

(continued from previous page)

```

cmap="gray",
extent=(theta[0], theta[-1], t0[-1], t0[0]),
vmin=dPP_const.min(),
vmax=dPP_const.max(),
)
axs[0].set_title("PP Data")
axs[0].axis("tight")
axs[1].imshow(
    dPS,
    cmap="gray",
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=dPS.min(),
    vmax=dPS.max(),
)
axs[1].set_title("PS Data")
axs[1].axis("tight")
plt.tight_layout()

```



Total running time of the script: (0 minutes 0.980 seconds)

3.6.6 Bilinear Interpolation

This example shows how to use the `pylops.signalprocessing.Bilinear` operator to perform bilinear interpolation to a 2-dimensional input vector.

```

import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt
import numpy as np
from scipy import misc

import pylops

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

```

First of all, we create a 2-dimensional input vector containing an image from the `scipy.misc` family.

```
x = misc.face()[::5, ::5, 0]
nz, nx = x.shape
```

We can now define a set of available samples in the first and second direction of the array and apply bilinear interpolation.

```
nsamples = 2000
iava = np.vstack(
    (np.random.uniform(0, nz - 1, nsamples), np.random.uniform(0, nx - 1, nsamples))
)

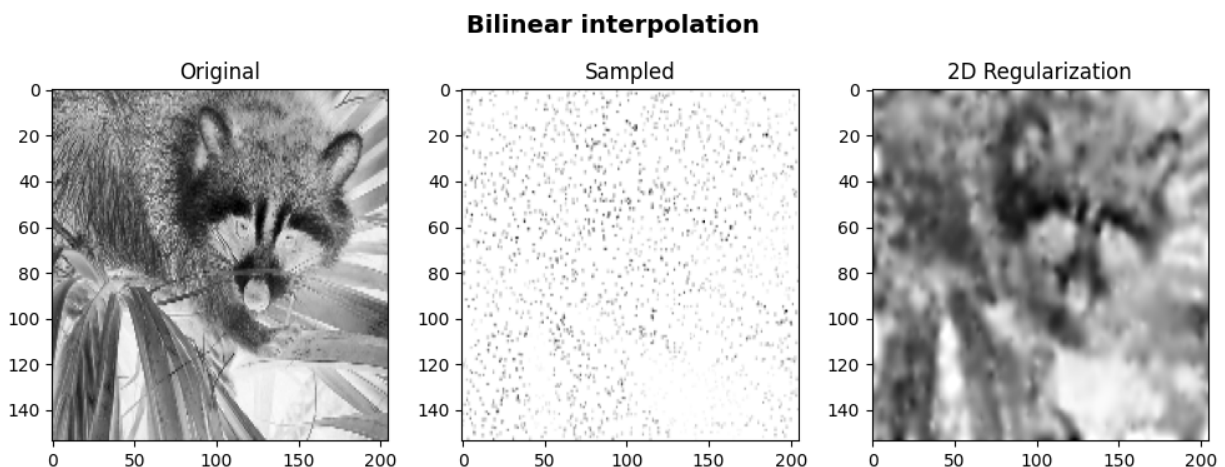
Bop = pylops.signalprocessing.Bilinear(iava, (nz, nx))
y = Bop * x.ravel()
```

At this point we try to reconstruct the input signal imposing a smooth solution by means of a regularization term that minimizes the Laplacian of the solution.

```
D2op = pylops.Laplacian((nz, nx), weights=(1, 1), dtype="float64")

xadj = Bop.H * y
xinv = pylops.optimization.leastsquares.NormalEquationsInversion(
    Bop, [D2op], y, epsRs=[np.sqrt(0.1)], returninfo=False, **dict(maxiter=100)
)
xadj = xadj.reshape(nz, nx)
xinv = xinv.reshape(nz, nx)

fig, axs = plt.subplots(1, 3, figsize=(10, 4))
fig.suptitle("Bilinear interpolation", fontsize=14, fontweight="bold", y=0.95)
axs[0].imshow(x, cmap="gray_r", vmin=0, vmax=250)
axs[0].axis("tight")
axs[0].set_title("Original")
axs[1].imshow(xadj, cmap="gray_r", vmin=0, vmax=250)
axs[1].axis("tight")
axs[1].set_title("Sampled")
axs[2].imshow(xinv, cmap="gray_r", vmin=0, vmax=250)
axs[2].axis("tight")
axs[2].set_title("2D Regularization")
plt.tight_layout()
plt.subplots_adjust(top=0.8)
```



Total running time of the script: (0 minutes 1.212 seconds)

3.6.7 CGLS and LSQR Solvers

This example shows how to use the `pylops.optimization.leastsquares.cgls` and `pylops.optimization.leastsquares.lsqr` PyLops solvers to minimize the following cost function:

$$J = \|y - Ax\|_2^2 + \epsilon \|x\|_2^2$$

Note that the LSQR solver behaves in the same way as the `scipy.sparse.linalg.lsqr` solver. However, our solver is also able to operate on `cupy` arrays and perform computations on a GPU.

```
import warnings

import matplotlib.pyplot as plt
import numpy as np

import pylops

plt.close("all")
warnings.filterwarnings("ignore")
```

Let's define a matrix **A** of size (N and M) and fill the matrix with random numbers

```
N, M = 20, 10
A = np.random.normal(0, 1, (N, M))
Aop = pylops.MatrixMult(A, dtype="float64")

x = np.ones(M)
```

We can now use the `cgls` solver to invert this matrix

```
y = Aop * x
xest, istop, nit, r1norm, r2norm, cost_cgls = pylops.optimization.solver.cgls(
    Aop, y, x0=np.zeros_like(x), niter=10, tol=1e-10, show=True
)

print("x= %s" % x)
print("cgls solution xest= %s" % xest)
```

Out:

```
CGLS
-----
The Operator Op has 20 rows and 10 cols
damp = 0.000000e+00      tol = 1.000000e-10      niter = 10
-----
```

Itn	x[0]	r1norm	r2norm
1	9.1362e-01	3.5210e+00	3.5210e+00
2	1.1328e+00	1.9174e+00	1.9174e+00
3	1.1030e+00	7.9210e-01	7.9210e-01
4	1.0366e+00	3.9919e-01	3.9919e-01
5	1.0086e+00	1.4627e-01	1.4627e-01
6	1.0069e+00	8.0987e-02	8.0987e-02
7	9.9981e-01	3.8979e-02	3.8979e-02
8	9.9936e-01	1.9302e-02	1.9302e-02

(continues on next page)

(continued from previous page)

```

    9      1.0006e+00      3.0820e-03      3.0820e-03
   10      1.0000e+00      3.6146e-15      3.6146e-15

```

```
Iterations = 10      Total time (s) = 0.00
```

```
-----
x= [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
cgls solution xest= [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

And the lsqr solver to invert this matrix

```

y = Aop * x
(
    xest,
    istop,
    itn,
    rlnorm,
    r2norm,
    anorm,
    acond,
    arnorm,
    xnorm,
    var,
    cost_lsqr,
) = pylops.optimization.solver.lsqr(Aop, y, x0=np.zeros_like(x), niter=10, show=True)

print("x= %s" % x)
print("lsqr solution xest= %s" % xest)

```

Out:

```

LSQR
-----
The Operator Op has 20 rows and 10 cols
damp = 0.0000000000000000e+00      calc_var =      1
atol = 1.00e-08                    conlim = 1.00e+08
btol = 1.00e-08                    niter =      10
-----

  Itn      x[0]      rlnorm      r2norm  Compatible   LS      Norm A   Cond A
  ---  ---
  0  0.00000e+00  1.650e+01  1.650e+01   1.0e+00  3.4e-01
  1  9.13620e-01  3.521e+00  3.521e+00   2.1e-01  1.4e-01  5.7e+00  1.0e+00
  2  1.13279e+00  1.917e+00  1.917e+00   1.2e-01  8.8e-02  7.3e+00  2.1e+00
  3  1.10304e+00  7.921e-01  7.921e-01   4.8e-02  3.9e-02  9.0e+00  3.5e+00
  4  1.03663e+00  3.992e-01  3.992e-01   2.4e-02  1.3e-02  1.1e+01  4.7e+00
  5  1.00858e+00  1.463e-01  1.463e-01   8.9e-03  5.1e-03  1.1e+01  6.2e+00
  6  1.00687e+00  8.099e-02  8.099e-02   4.9e-03  3.3e-03  1.2e+01  7.4e+00
  7  9.99808e-01  3.898e-02  3.898e-02   2.4e-03  1.5e-03  1.3e+01  8.8e+00
  8  9.99356e-01  1.930e-02  1.930e-02   1.2e-03  7.2e-04  1.4e+01  1.0e+01
  9  1.00062e+00  3.082e-03  3.082e-03   1.9e-04  8.3e-05  1.4e+01  1.2e+01
 10  1.00000e+00  4.480e-15  4.480e-15   2.7e-16  3.1e-16  1.4e+01  1.3e+01

LSQR finished, Opx - b is small enough, given atol, btol

istop =      1      rlnorm = 4.5e-15      anorm = 1.4e+01      arnorm = 2.8e-14
itn     =     10      r2norm = 4.5e-15      acond = 1.3e+01      xnorm  = 3.2e+00
Total time (s) = 0.00

```

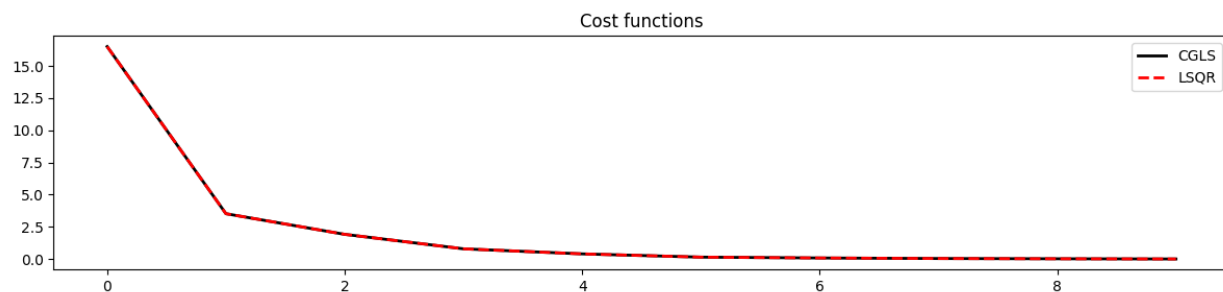
(continues on next page)

(continued from previous page)

```
-----
x= [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
lsqr solution xest= [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Finally we show that the L2 norm of the residual of the two solvers decays in the same way, as LSQR is algebraically equivalent to CG on the normal equations and CGLS

```
plt.figure(figsize=(12, 3))
plt.plot(cost_cgls, "k", lw=2, label="CGLS")
plt.plot(cost_lsqr, "--r", lw=2, label="LSQR")
plt.title("Cost functions")
plt.legend()
plt.tight_layout()
```



Note that while we used a dense matrix here, any other linear operator can be fed to `cgls` and `lsqr` as is the case for any other PyLops solver.

Total running time of the script: (0 minutes 0.188 seconds)

3.6.8 Causal Integration

This example shows how to use the `pylops.CausalIntegration` operator to integrate an input signal (in forward mode) and to apply a smooth, regularized derivative (in inverse mode). This is a very interesting by-product of this operator which may result very useful when the data to which you want to apply a numerical derivative is noisy.

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

import pylops

plt.close("all")
```

Let's start with a 1D example. Define the input parameters: number of samples of input signal (`nt`), sampling step (`dt`) as well as the input signal which will be equal to $x(t) = \sin(t)$:

```
nt = 81
dt = 0.3
t = np.arange(nt) * dt
x = np.sin(t)
```

We can now create our causal integration operator and apply it to the input signal. We can also compute the analytical integral $y(t) = \int \sin(t) dt = -\cos(t)$ and compare the results. We can also invert the integration operator and by

remembering that this is equivalent to a first order derivative, we will compare our inverted model with the result obtained by simply applying the `pylops.FirstDerivative` forward operator to the same data.

Note that, as explained in details in `pylops.CausalIntegration`, integration has no unique solution, as any constant c can be added to the integrated signal y , for example if $x(t) = t^2$ the $y(t) = \int t^2 dt = \frac{t^3}{3} + c$. We thus subtract first sample from the analytical integral to obtain the same result as the numerical one.

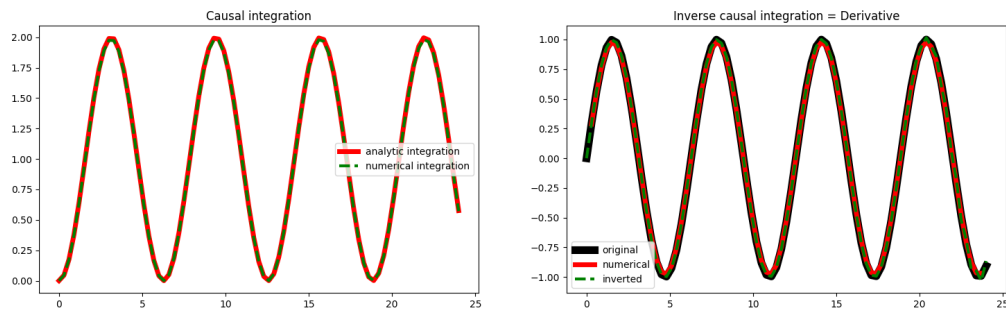
```
Cop = pylops.CausalIntegration(nt, sampling=dt, halfcurrent=True)

yana = -np.cos(t) + np.cos(t[0])
y = Cop * x
xinv = Cop / y

# Numerical derivative
Dop = pylops.FirstDerivative(nt, sampling=dt)
xder = Dop * y

# Visualize data and inversion
fig, axs = plt.subplots(1, 2, figsize=(18, 5))
axs[0].plot(t, yana, "r", lw=5, label="analytic integration")
axs[0].plot(t, y, "--g", lw=3, label="numerical integration")
axs[0].legend()
axs[0].set_title("Causal integration")

axs[1].plot(t, x, "k", lw=8, label="original")
axs[1].plot(t[1:-1], xder[1:-1], "r", lw=5, label="numerical")
axs[1].plot(t, xinv, "--g", lw=3, label="inverted")
axs[1].legend()
axs[1].set_title("Inverse causal integration = Derivative")
```



Out:

```
Text(0.5, 1.0, 'Inverse causal integration = Derivative')
```

As expected we obtain the same result. Let's see what happens if we now add some random noise to our data.

```
# Add noise
yn = y + np.random.normal(0, 4e-1, y.shape)

# Numerical derivative
Dop = pylops.FirstDerivative(nt, sampling=dt)
xder = Dop * yn

# Regularized derivative
Rop = pylops.SecondDerivative(nt)
```

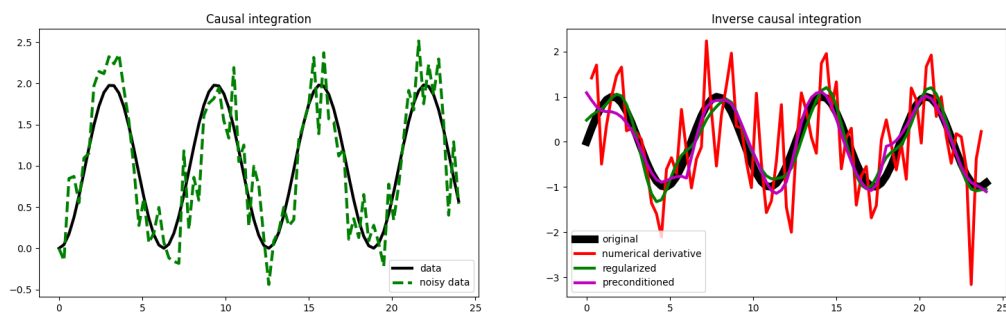
(continues on next page)

(continued from previous page)

```
xreg = pylops.RegularizedInversion(
    Cop, [Rop], yn, epsRs=[1e0], **dict(iter_lim=100, atol=1e-5)
)

# Preconditioned derivative
Sop = pylops.Smoothing1D(41, nt)
xp = pylops.PreconditionedInversion(Cop, Sop, yn, **dict(iter_lim=10, atol=1e-3))

# Visualize data and inversion
fig, axs = plt.subplots(1, 2, figsize=(18, 5))
axs[0].plot(t, y, "k", lw=3, label="data")
axs[0].plot(t, yn, "--g", lw=3, label="noisy data")
axs[0].legend()
axs[0].set_title("Causal integration")
axs[1].plot(t, x, "k", lw=8, label="original")
axs[1].plot(t[1:-1], xder[1:-1], "r", lw=3, label="numerical derivative")
axs[1].plot(t, xreg, "g", lw=3, label="regularized")
axs[1].plot(t, xp, "m", lw=3, label="preconditioned")
axs[1].legend()
axs[1].set_title("Inverse causal integration")
```



Out:

```
Text(0.5, 1.0, 'Inverse causal integration')
```

We can see here the great advantage of framing our numerical derivative as an inverse problem, and more specifically as the inverse of the causal integration operator.

Let's conclude with a 2d example where again the integration/derivative will be performed along the first axis

```
nt, nx = 41, 11
dt = 0.3
ot = 0
t = np.arange(nt) * dt + ot
x = np.outer(np.sin(t), np.ones(nx))

Cop = pylops.CausalIntegration(
    nt * nx, dims=(nt, nx), sampling=dt, dir=0, halfcurrent=True
)

y = Cop * x.ravel()
y = y.reshape(nt, nx)
yn = y + np.random.normal(0, 4e-1, y.shape)
```

(continues on next page)

(continued from previous page)

```

# Numerical derivative
Dop = pylops.FirstDerivative(nt * nx, dims=(nt, nx), dir=0, sampling=dt)
xder = Dop * yn.ravel()
xder = xder.reshape(nt, nx)

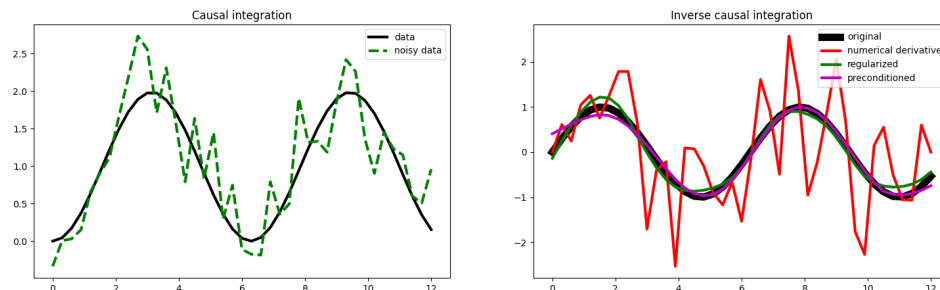
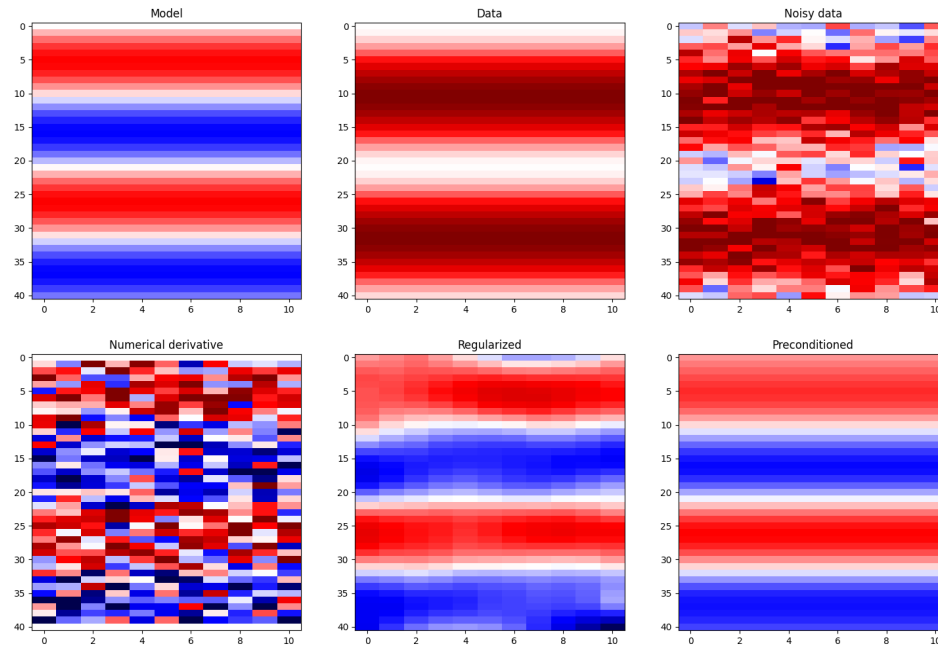
# Regularized derivative
Rop = pylops.Laplacian(dims=(nt, nx))
xreg = pylops.RegularizedInversion(
    Cop, [Rop], yn.ravel(), epsRs=[1e0], **dict(iter_lim=100, atol=1e-5)
)
xreg = xreg.reshape(nt, nx)

# Preconditioned derivative
Sop = pylops.Smoothing2D((11, 21), dims=(nt, nx))
xp = pylops.PreconditionedInversion(
    Cop, Sop, yn.ravel(), **dict(iter_lim=10, atol=1e-2)
)
xp = xp.reshape(nt, nx)

# Visualize data and inversion
vmax = 2 * np.max(np.abs(x))
fig, axs = plt.subplots(2, 3, figsize=(18, 12))
axs[0][0].imshow(x, cmap="seismic", vmin=-vmax, vmax=vmax)
axs[0][0].set_title("Model")
axs[0][0].axis("tight")
axs[0][1].imshow(y, cmap="seismic", vmin=-vmax, vmax=vmax)
axs[0][1].set_title("Data")
axs[0][1].axis("tight")
axs[0][2].imshow(yn, cmap="seismic", vmin=-vmax, vmax=vmax)
axs[0][2].set_title("Noisy data")
axs[0][2].axis("tight")
axs[1][0].imshow(xder, cmap="seismic", vmin=-vmax, vmax=vmax)
axs[1][0].set_title("Numerical derivative")
axs[1][0].axis("tight")
axs[1][1].imshow(xreg, cmap="seismic", vmin=-vmax, vmax=vmax)
axs[1][1].set_title("Regularized")
axs[1][1].axis("tight")
axs[1][2].imshow(xp, cmap="seismic", vmin=-vmax, vmax=vmax)
axs[1][2].set_title("Preconditioned")
axs[1][2].axis("tight")

# Visualize data and inversion at a chosen xlocation
fig, axs = plt.subplots(1, 2, figsize=(18, 5))
axs[0].plot(t, y[:, nx // 2], "k", lw=3, label="data")
axs[0].plot(t, yn[:, nx // 2], "--g", lw=3, label="noisy data")
axs[0].legend()
axs[0].set_title("Causal integration")
axs[1].plot(t, x[:, nx // 2], "k", lw=8, label="original")
axs[1].plot(t, xder[:, nx // 2], "r", lw=3, label="numerical derivative")
axs[1].plot(t, xreg[:, nx // 2], "g", lw=3, label="regularized")
axs[1].plot(t, xp[:, nx // 2], "m", lw=3, label="preconditioned")
axs[1].legend()
axs[1].set_title("Inverse causal integration")

```



Out:

```
Text(0.5, 1.0, 'Inverse causal integration')
```

Total running time of the script: (0 minutes 1.545 seconds)

3.6.9 Chirp Radon Transform

This example shows how to use the `pylops.signalprocessing.ChirpRadon2D` and `pylops.signalprocessing.ChirpRadon3D` operators to apply the linear Radon Transform to 2-dimensional or 3-dimensional signals, respectively.

When working with the linear Radon transform, this is a faster implementation compared to in `pylops.signalprocessing.Radon2D` and `pylops.signalprocessing.Radon3D` and should be preferred. This method provides also an analytical inverse.

Note that the forward and adjoint definitions in these two pairs of operators are swapped.

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

import pylops

plt.close("all")
```

Let's start by creating a empty 2d matrix of size $n_x \times n_t$ with a single linear event.

```
par = {
    "ot": 0,
    "dt": 0.004,
    "nt": 51,
    "ox": -250,
    "dx": 10,
    "nx": 51,
    "oy": -250,
    "dy": 10,
    "ny": 51,
    "f0": 40,
}
theta = [
    0,
]
t0 = [
    0.1,
]
amp = [
    1.0,
]

# Create axes
t, t2, x, y = pylops.utils.seismicevents.makeaxis(par)
dt, dx, dy = par["dt"], par["dx"], par["dy"]

# Create wavelet
wav, _, wav_c = pylops.utils.wavelets.ricker(t[:41], f0=par["f0"])

# Generate data
_, d = pylops.utils.seismicevents.linear2d(x, t, 1500.0, t0, theta, amp, wav)
```

We can now define our operators and apply the forward, adjoint and inverse steps.

```
npix, pxmax = par["nx"], 5e-4
px = np.linspace(-pxmax, pxmax, npix)

R2Op = pylops.signalprocessing.ChirpRadon2D(t, x, pxmax * dx / dt, dtype="float64")
dL_chirp = R2Op * d.ravel()
dadj_chirp = R2Op.H * dL_chirp
dinv_chirp = R2Op.inverse(dL_chirp)

dL_chirp = dL_chirp.reshape(par["nx"], par["nt"])
dadj_chirp = dadj_chirp.reshape(par["nx"], par["nt"])
dinv_chirp = dinv_chirp.reshape(par["nx"], par["nt"])

fig, axs = plt.subplots(1, 4, figsize=(12, 4))
axs[0].imshow(d.T, vmin=-1, vmax=1, cmap="seismic_r", extent=(x[0], x[-1], t[-1],
↪t[0]))
```

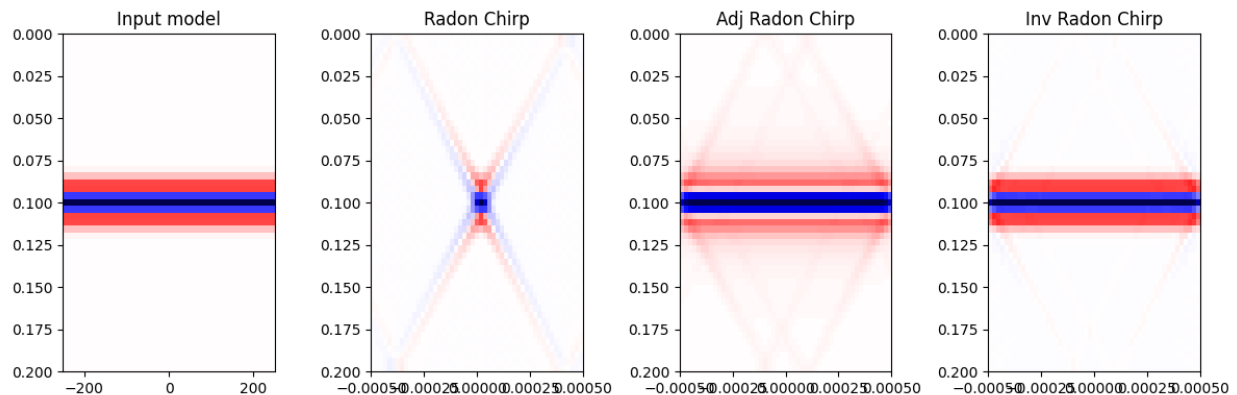
(continues on next page)

(continued from previous page)

```

axs[0].set_title("Input model")
axs[0].axis("tight")
axs[1].imshow(
    dL_chirp.T,
    cmap="seismic_r",
    vmin=-dL_chirp.max(),
    vmax=dL_chirp.max(),
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[1].set_title("Radon Chirp")
axs[1].axis("tight")
axs[2].imshow(
    dadj_chirp.T,
    cmap="seismic_r",
    vmin=-dadj_chirp.max(),
    vmax=dadj_chirp.max(),
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[2].set_title("Adj Radon Chirp")
axs[2].axis("tight")
axs[3].imshow(
    dinv_chirp.T,
    cmap="seismic_r",
    vmin=-d.max(),
    vmax=d.max(),
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[3].set_title("Inv Radon Chirp")
axs[3].axis("tight")
plt.tight_layout()

```



Finally we repeat the same exercise with 3d data.

```

par = {
    "ot": 0,
    "dt": 0.004,
    "nt": 51,
    "ox": -400,
    "dx": 10,
    "nx": 81,
    "oy": -600,
    "dy": 10,
}

```

(continues on next page)

(continued from previous page)

```

    "ny": 61,
    "f0": 20,
}
theta = [
    10,
]
phi = [
    0,
]
t0 = [
    0.1,
]
amp = [
    1.0,
]

# Create axes
t, t2, x, y = pylops.utils.seismicevents.makeaxis(par)
dt, dx, dy = par["dt"], par["dx"], par["dy"]

# Generate data
_, d = pylops.utils.seismicevents.linear3d(x, y, t, 1500.0, t0, theta, phi, amp, wav)

npy, pymax = par["ny"], 3e-4
npx, pxmax = par["nx"], 5e-4
py = np.linspace(-pymax, pymax, npy)
px = np.linspace(-pxmax, pxmax, npx)

R3Op = pylops.signalprocessing.ChirpRadon3D(
    t, y, x, (pymax * dy / dt, pxmax * dx / dt), dtype="float64"
)
dL_chirp = R3Op * d.ravel()
dadj_chirp = R3Op.H * dL_chirp
dinv_chirp = R3Op.inverse(dL_chirp)

dL_chirp = dL_chirp.reshape(par["ny"], par["nx"], par["nt"])
dadj_chirp = dadj_chirp.reshape(par["ny"], par["nx"], par["nt"])
dinv_chirp = dinv_chirp.reshape(par["ny"], par["nx"], par["nt"])

fig, axs = plt.subplots(1, 4, figsize=(12, 4))
axs[0].imshow(
    d[par["ny"] // 2].T,
    vmin=-1,
    vmax=1,
    cmap="seismic_r",
    extent=(x[0], x[-1], t[-1], t[0]),
)
axs[0].set_title("Input model")
axs[0].axis("tight")
axs[1].imshow(
    dL_chirp[par["ny"] // 2].T,
    cmap="seismic_r",
    vmin=-dL_chirp.max(),
    vmax=dL_chirp.max(),
    extent=(px[0], px[-1], t[-1], t[0]),
)

```

(continues on next page)

(continued from previous page)

```

axs[1].set_title("Radon Chirp")
axs[1].axis("tight")
axs[2].imshow(
    dadj_chirp[par["ny"] // 2].T,
    cmap="seismic_r",
    vmin=-dadj_chirp.max(),
    vmax=dadj_chirp.max(),
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[2].set_title("Adj Radon Chirp")
axs[2].axis("tight")
axs[3].imshow(
    dinv_chirp[par["ny"] // 2].T,
    cmap="seismic_r",
    vmin=-d.max(),
    vmax=d.max(),
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[3].set_title("Inv Radon Chirp")
axs[3].axis("tight")
plt.tight_layout()

fig, axs = plt.subplots(1, 4, figsize=(12, 4))
axs[0].imshow(
    d[:, par["nx"] // 2].T,
    vmin=-1,
    vmax=1,
    cmap="seismic_r",
    extent=(x[0], x[-1], t[-1], t[0]),
)
axs[0].set_title("Input model")
axs[0].axis("tight")
axs[1].imshow(
    dL_chirp[:, 2 * par["nx"] // 3].T,
    cmap="seismic_r",
    vmin=-dL_chirp.max(),
    vmax=dL_chirp.max(),
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[1].set_title("Radon Chirp")
axs[1].axis("tight")
axs[2].imshow(
    dadj_chirp[:, par["nx"] // 2].T,
    cmap="seismic_r",
    vmin=-dadj_chirp.max(),
    vmax=dadj_chirp.max(),
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[2].set_title("Adj Radon Chirp")
axs[2].axis("tight")
axs[3].imshow(
    dinv_chirp[:, par["nx"] // 2].T,
    cmap="seismic_r",
    vmin=-d.max(),
    vmax=d.max(),
    extent=(px[0], px[-1], t[-1], t[0]),
)

```

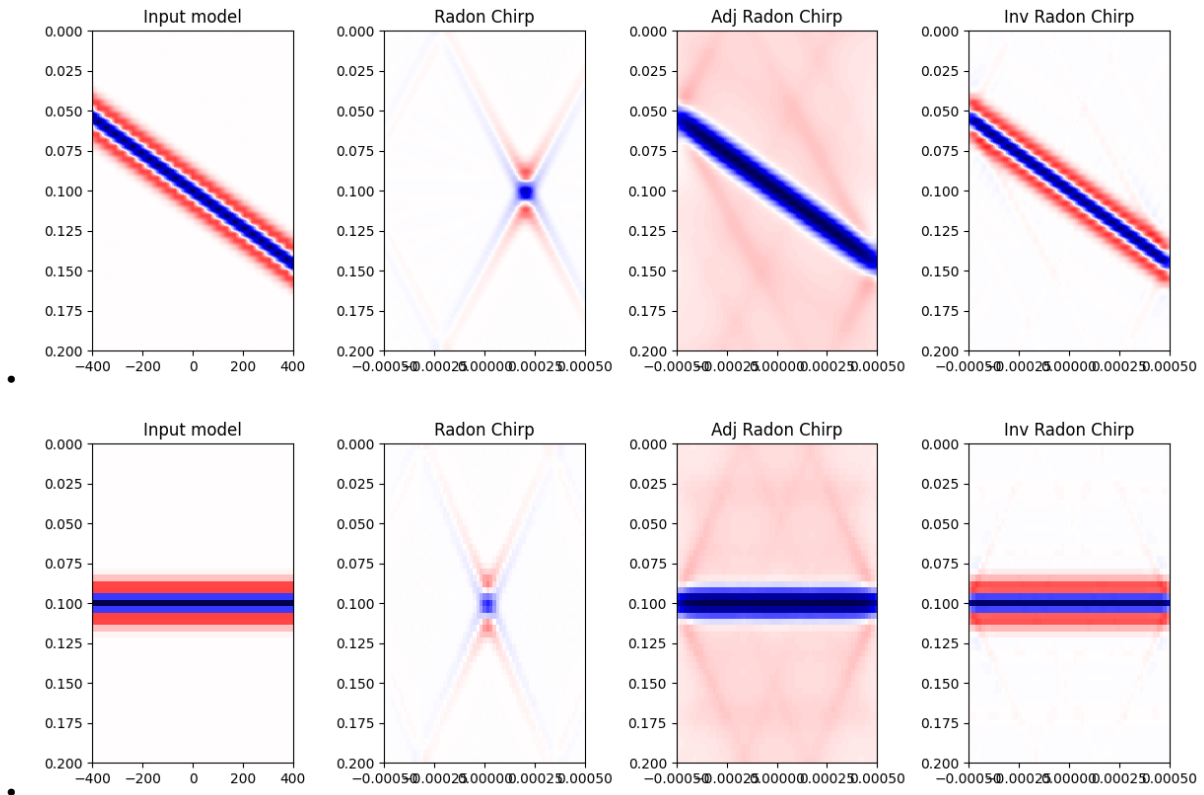
(continues on next page)

(continued from previous page)

```

axs[3].set_title("Inv Radon Chirp")
axs[3].axis("tight")
plt.tight_layout()

```



Total running time of the script: (0 minutes 1.664 seconds)

3.6.10 Conj

This example shows how to use the `pylops.basicoperators.Conj` operator. This operator returns the complex conjugate in both forward and adjoint modes (it is self adjoint).

```

import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt
import numpy as np

import pylops

plt.close("all")

```

Let's define a `Conj` operator to get the complex conjugate of the input.

```

M = 5
x = np.arange(M) + 1j * np.arange(M)[:,::-1]
Rop = pylops.basicoperators.Conj(M, dtype="complex128")

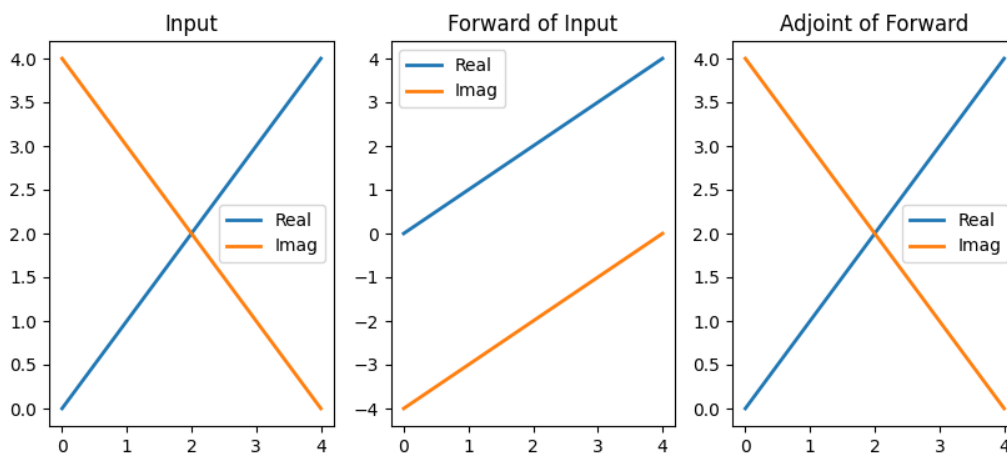
y = Rop * x
xadj = Rop.H * y

```

(continues on next page)

(continued from previous page)

```
_, axs = plt.subplots(1, 3, figsize=(10, 4))
axs[0].plot(np.real(x), lw=2, label="Real")
axs[0].plot(np.imag(x), lw=2, label="Imag")
axs[0].legend()
axs[0].set_title("Input")
axs[1].plot(np.real(y), lw=2, label="Real")
axs[1].plot(np.imag(y), lw=2, label="Imag")
axs[1].legend()
axs[1].set_title("Forward of Input")
axs[2].plot(np.real(xadj), lw=2, label="Real")
axs[2].plot(np.imag(xadj), lw=2, label="Imag")
axs[2].legend()
axs[2].set_title("Adjoint of Forward")
```



Out:

```
Text(0.5, 1.0, 'Adjoint of Forward')
```

Total running time of the script: (0 minutes 0.271 seconds)

3.6.11 Convolution

This example shows how to use the `pylops.signalprocessing.Convolve1D`, `pylops.signalprocessing.Convolve2D` and `pylops.signalprocessing.ConvolveND` operators to perform convolution between two signals.

Such operators can be used in the forward model of several common application in signal processing that require filtering of an input signal for the instrument response. Similarly, removing the effect of the instrument response from signal is equivalent to solving linear system of equations based on `Convolve1D`, `Convolve2D` or `ConvolveND` operators. This problem is generally referred to as *Deconvolution*.

A very practical example of deconvolution can be found in the geophysical processing of seismic data where the effect of the source response (i.e., airgun or vibroseis) should be removed from the recorded signal to be able to better interpret the response of the subsurface. Similar examples can be found in telecommunication and speech analysis.

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

(continues on next page)

(continued from previous page)

```

from scipy.sparse.linalg import lsqr

import pylops
from pylops.utils.wavelets import ricker

plt.close("all")

```

We will start by creating a zero signal of length nt and we will place a unitary spike at its center. We also create our filter to be applied by means of `pylops.signalprocessing.Convolve1D` operator. Following the seismic example mentioned above, the filter is a **Ricker wavelet** with dominant frequency $f_0 = 30Hz$.

```

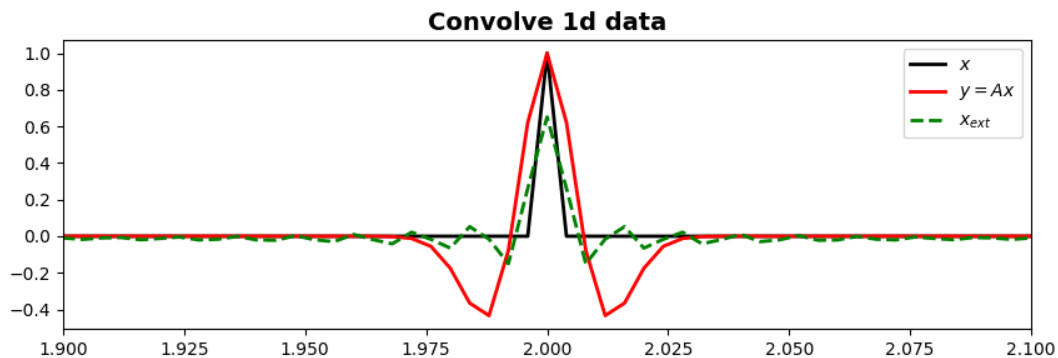
nt = 1001
dt = 0.004
t = np.arange(nt) * dt
x = np.zeros(nt)
x[int(nt / 2)] = 1
h, th, hcenter = ricker(t[:101], f0=30)

Cop = pylops.signalprocessing.Convolve1D(nt, h=h, offset=hcenter, dtype="float32")
y = Cop * x

xinv = Cop / y

fig, ax = plt.subplots(1, 1, figsize=(10, 3))
ax.plot(t, x, "k", lw=2, label=r"$x$")
ax.plot(t, y, "r", lw=2, label=r"$y=Ax$")
ax.plot(t, xinv, "--g", lw=2, label=r"$x_{ext}$")
ax.set_title("Convolve 1d data", fontsize=14, fontweight="bold")
ax.legend()
ax.set_xlim(1.9, 2.1)

```



Out:

```
(1.9, 2.1)
```

We show now that also a filter with mixed phase (i.e., not centered around zero) can be applied and inverted for using the `pylops.signalprocessing.Convolve1D` operator.

```

Cop = pylops.signalprocessing.Convolve1D(nt, h=h, offset=hcenter - 3, dtype="float32")
y = Cop * x
y1 = Cop.H * x
xinv = Cop / y

```

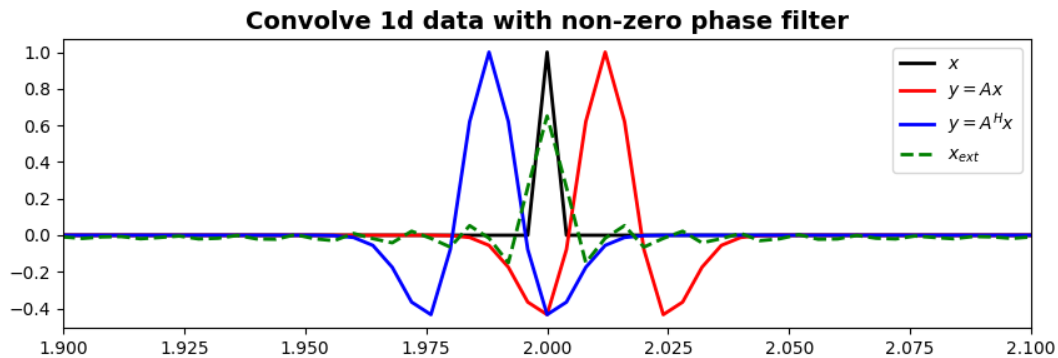
(continues on next page)

(continued from previous page)

```

fig, ax = plt.subplots(1, 1, figsize=(10, 3))
ax.plot(t, x, "k", lw=2, label=r"$x$")
ax.plot(t, y, "r", lw=2, label=r"$y=Ax$")
ax.plot(t, y1, "b", lw=2, label=r"$y=A^Hx$")
ax.plot(t, xinv, "--g", lw=2, label=r"$x_{\text{ext}}$")
ax.set_title(
    "Convolve 1d data with non-zero phase filter", fontsize=14, fontweight="bold"
)
ax.set_xlim(1.9, 2.1)
ax.legend()

```



Out:

```
<matplotlib.legend.Legend object at 0x7f6af6674cf8>
```

We repeat a similar exercise but using two dimensional signals and filters taking advantage of the `pylops.signalprocessing.Convolve2D` operator.

```

nt = 51
nx = 81
dt = 0.004
t = np.arange(nt) * dt
x = np.zeros((nt, nx))
x[int(nt / 2), int(nx / 2)] = 1

nh = [11, 5]
h = np.ones((nh[0], nh[1]))

Cop = pylops.signalprocessing.Convolve2D(
    nt * nx,
    h=h,
    offset=(int(nh[0]) / 2, int(nh[1]) / 2),
    dims=(nt, nx),
    dtype="float32",
)
y = Cop * x.ravel()
xinv = Cop / y

y = y.reshape(nt, nx)
xinv = xinv.reshape(nt, nx)

fig, axs = plt.subplots(1, 3, figsize=(10, 3))
fig.suptitle("Convolve 2d data", fontsize=14, fontweight="bold", y=0.95)

```

(continues on next page)

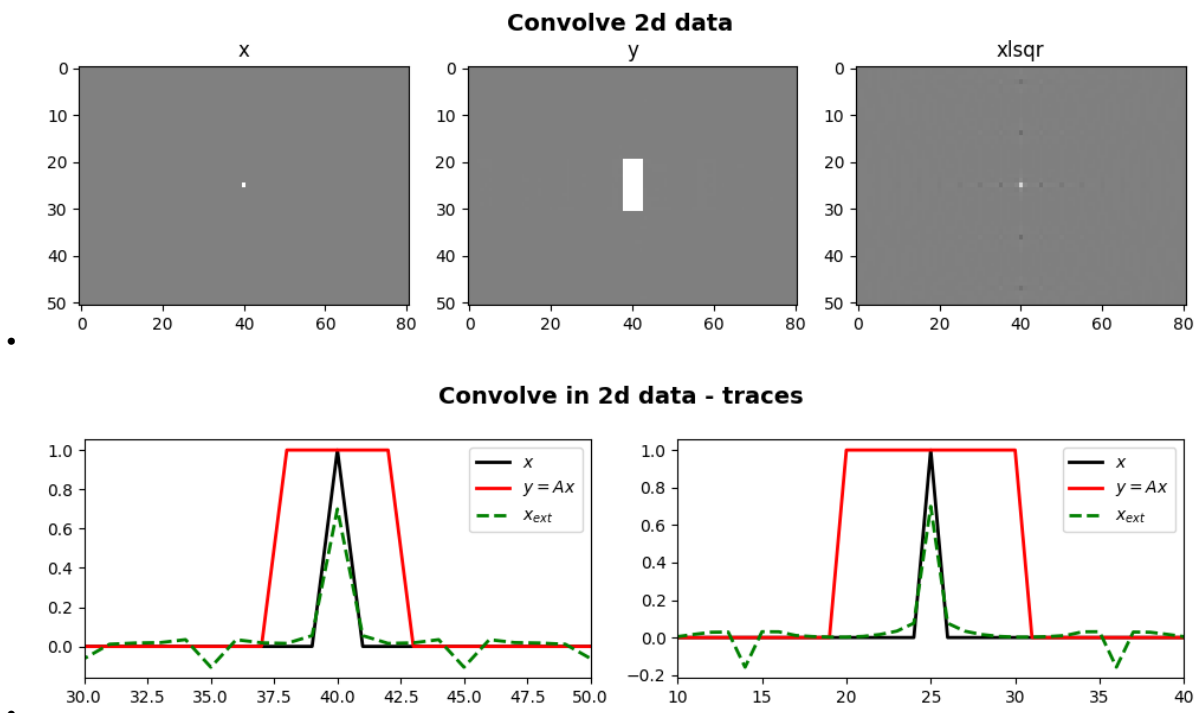
(continued from previous page)

```

axs[0].imshow(x, cmap="gray", vmin=-1, vmax=1)
axs[1].imshow(y, cmap="gray", vmin=-1, vmax=1)
axs[2].imshow(xinv, cmap="gray", vmin=-1, vmax=1)
axs[0].set_title("x")
axs[0].axis("tight")
axs[1].set_title("y")
axs[1].axis("tight")
axs[2].set_title("xlsqr")
axs[2].axis("tight")
plt.tight_layout()
plt.subplots_adjust(top=0.8)

fig, ax = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle("Convolve in 2d data - traces", fontsize=14, fontweight="bold", y=0.95)
ax[0].plot(x[int(nt / 2), :], "k", lw=2, label=r"$x$")
ax[0].plot(y[int(nt / 2), :], "r", lw=2, label=r"$y=Ax$")
ax[0].plot(xinv[int(nt / 2), :], "--g", lw=2, label=r"$x_{\text{ext}}$")
ax[1].plot(x[:, int(nx / 2)], "k", lw=2, label=r"$x$")
ax[1].plot(y[:, int(nx / 2)], "r", lw=2, label=r"$y=Ax$")
ax[1].plot(xinv[:, int(nx / 2)], "--g", lw=2, label=r"$x_{\text{ext}}$")
ax[0].legend()
ax[0].set_xlim(30, 50)
ax[1].legend()
ax[1].set_xlim(10, 40)
plt.tight_layout()
plt.subplots_adjust(top=0.8)

```



Finally we do the same using three dimensional signals and filters taking advantage of the `pylops.signalprocessing.ConvolveND` operator.

```

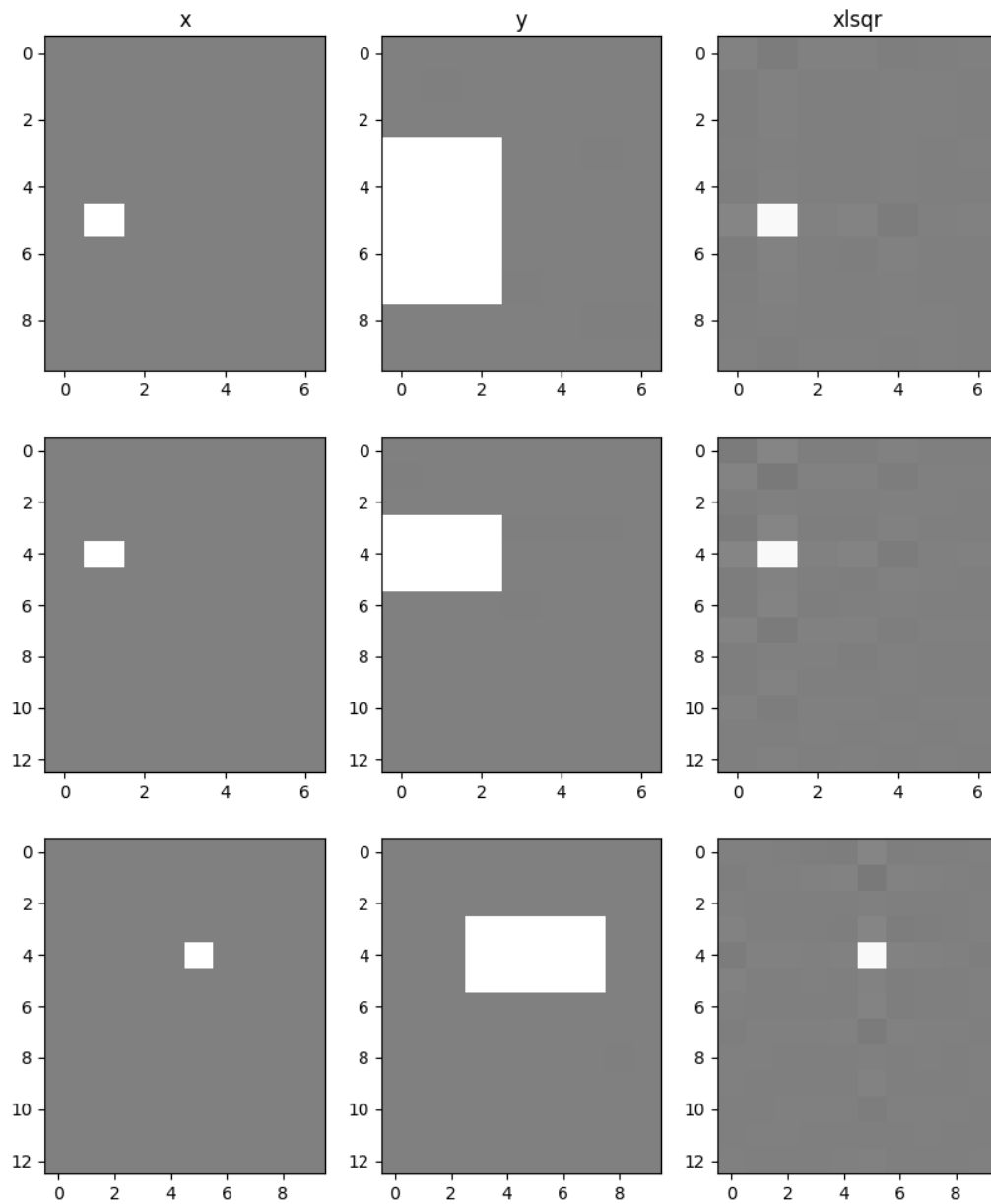
ny, nx, nz = 13, 10, 7
x = np.zeros((ny, nx, nz))
x[ny // 3, nx // 2, nz // 4] = 1
h = np.ones((3, 5, 3))
offset = [1, 2, 1]

Cop = pylops.signalprocessing.ConvolveND(
    nx * ny * nz, h=h, offset=offset, dims=[ny, nx, nz], dirs=[0, 1, 2], dtype=
    ↪ "float32"
)
y = Cop * x.ravel()
xinv = lsqr(Cop, y, damp=0, iter_lim=300, show=0)[0]

y = y.reshape(ny, nx, nz)
xlsqr = xinv.reshape(ny, nx, nz)

fig, axs = plt.subplots(3, 3, figsize=(10, 12))
fig.suptitle("Convolve 3d data", y=0.95, fontsize=14, fontweight="bold")
axs[0][0].imshow(x[ny // 3], cmap="gray", vmin=-1, vmax=1)
axs[0][1].imshow(y[ny // 3], cmap="gray", vmin=-1, vmax=1)
axs[0][2].imshow(xlsqr[ny // 3], cmap="gray", vmin=-1, vmax=1)
axs[0][0].set_title("x")
axs[0][0].axis("tight")
axs[0][1].set_title("y")
axs[0][1].axis("tight")
axs[0][2].set_title("xlsqr")
axs[0][2].axis("tight")
axs[1][0].imshow(x[:, nx // 2], cmap="gray", vmin=-1, vmax=1)
axs[1][1].imshow(y[:, nx // 2], cmap="gray", vmin=-1, vmax=1)
axs[1][2].imshow(xlsqr[:, nx // 2], cmap="gray", vmin=-1, vmax=1)
axs[1][0].axis("tight")
axs[1][1].axis("tight")
axs[1][2].axis("tight")
axs[2][0].imshow(x[..., nz // 4], cmap="gray", vmin=-1, vmax=1)
axs[2][1].imshow(y[..., nz // 4], cmap="gray", vmin=-1, vmax=1)
axs[2][2].imshow(xlsqr[..., nz // 4], cmap="gray", vmin=-1, vmax=1)
axs[2][0].axis("tight")
axs[2][1].axis("tight")
axs[2][2].axis("tight")

```

Convolve 3d data

Out:

```
(-0.5, 9.5, 12.5, -0.5)
```

Total running time of the script: (0 minutes 1.954 seconds)

3.6.12 Derivatives

This example shows how to use the suite of derivative operators, namely `pylops.FirstDerivative`, `pylops.SecondDerivative`, `pylops.Laplacian` and `pylops.Gradient`, `pylops.FirstDirectionalDerivative` and `pylops.SecondDirectionalDerivative`.

The derivative operators are very useful when the model to be inverted for is expected to be smooth in one or more directions. As shown in the *Optimization* tutorial, these operators will be used as part of the regularization term to obtain a smooth solution.

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

import pylops

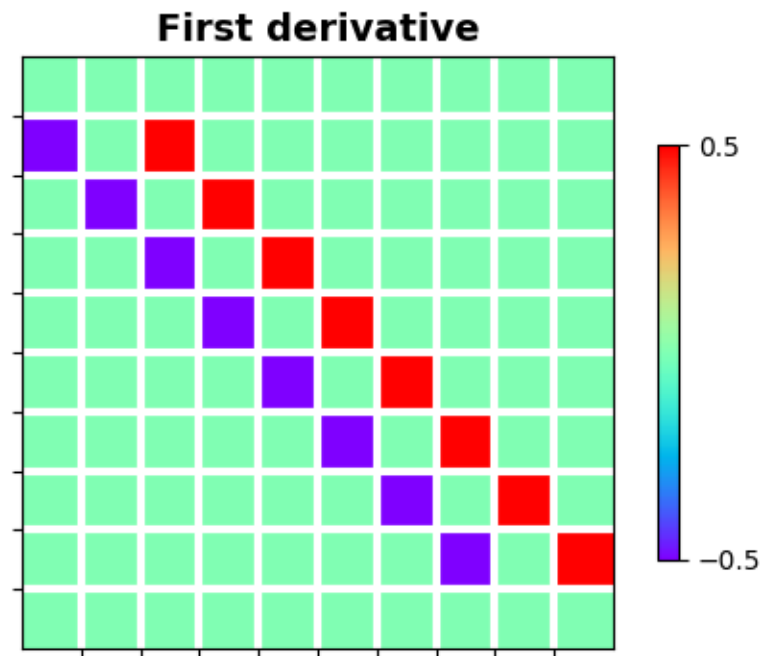
plt.close("all")
np.random.seed(0)
```

Let's start by looking at a simple first-order centered derivative and how could implement it naively by creating a dense matrix. Note that we will not apply the derivative where the stencil is partially outside of the range of the input signal (i.e., at the edge of the signal)

```
nx = 10

D = np.diag(0.5 * np.ones(nx - 1), k=1) - np.diag(0.5 * np.ones(nx - 1), -1)
D[0] = D[-1] = 0

fig, ax = plt.subplots(1, 1, figsize=(6, 4))
im = plt.imshow(D, cmap="rainbow", vmin=-0.5, vmax=0.5)
ax.set_title("First derivative", size=14, fontweight="bold")
ax.set_xticks(np.arange(nx - 1) + 0.5)
ax.set_yticks(np.arange(nx - 1) + 0.5)
ax.grid(linewidth=3, color="white")
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
fig.colorbar(im, ax=ax, ticks=[-0.5, 0.5], shrink=0.7)
```



Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f6aed817080>
```

We now create a signal filled with zero and a single one at its center and apply the derivative matrix by means of a dot product

```
x = np.zeros(nx)
x[int(nx / 2)] = 1

y_dir = np.dot(D, x)
xadj_dir = np.dot(D.T, y_dir)
```

Let's now do the same using the `pylops.FirstDerivative` operator and compare its outputs after applying the forward and adjoint operators to those from the dense matrix.

```
Dlop = pylops.FirstDerivative(nx, dtype="float32")

y_lop = Dlop * x
xadj_lop = Dlop.H * y_lop

fig, axs = plt.subplots(3, 1, figsize=(13, 8))
axs[0].stem(np.arange(nx), x, linefmt="k", markerfmt="ko")
axs[0].set_title("Input", size=20, fontweight="bold")
axs[1].stem(np.arange(nx), y_dir, linefmt="k", markerfmt="ko", label="direct")
axs[1].stem(np.arange(nx), y_lop, linefmt="--r", markerfmt="ro", label="lop")
axs[1].set_title("Forward", size=20, fontweight="bold")
axs[1].legend()
axs[2].stem(np.arange(nx), xadj_dir, linefmt="k", markerfmt="ko", label="direct")
axs[2].stem(np.arange(nx), xadj_lop, linefmt="--r", markerfmt="ro", label="lop")
axs[2].set_title("Adjoint", size=20, fontweight="bold")
```

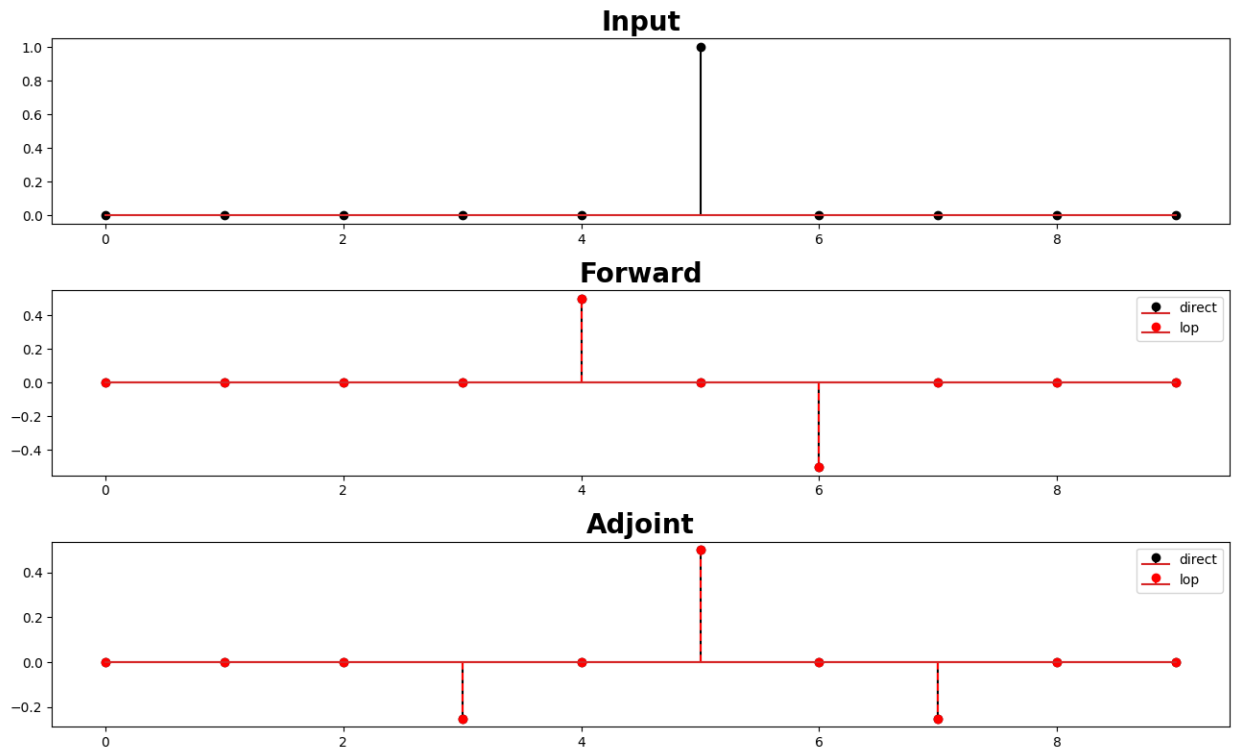
(continues on next page)

(continued from previous page)

```

axs[2].legend()
plt.tight_layout()

```



As expected we obtain the same result, with the only difference that in the second case we did not need to explicitly create a matrix, saving memory and computational time.

Let's move onto applying the same first derivative to a 2d array in the first direction

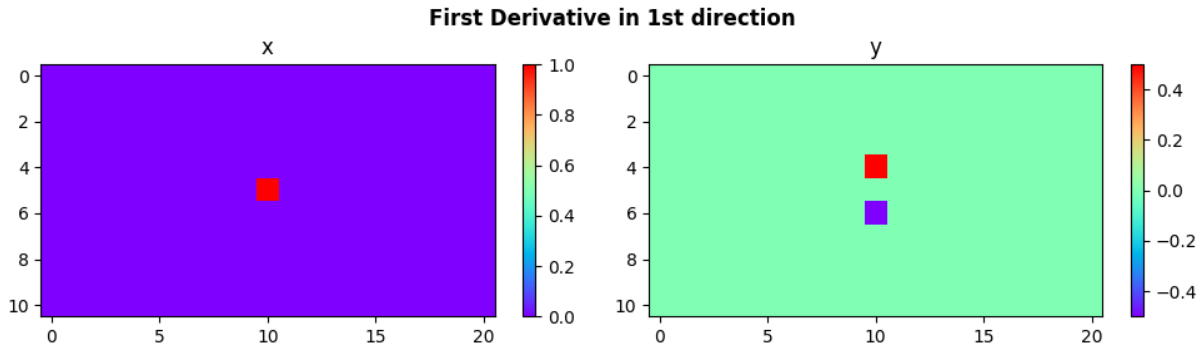
```

nx, ny = 11, 21
A = np.zeros((nx, ny))
A[nx // 2, ny // 2] = 1.0

Dlop = pyllops.FirstDerivative(nx * ny, dims=(nx, ny), dir=0, dtype="float64")
B = np.reshape(Dlop * A.ravel(), (nx, ny))

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle(
    "First Derivative in 1st direction", fontsize=12, fontweight="bold", y=0.95
)
im = axs[0].imshow(A, interpolation="nearest", cmap="rainbow")
axs[0].axis("tight")
axs[0].set_title("x")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(B, interpolation="nearest", cmap="rainbow")
axs[1].axis("tight")
axs[1].set_title("y")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)

```

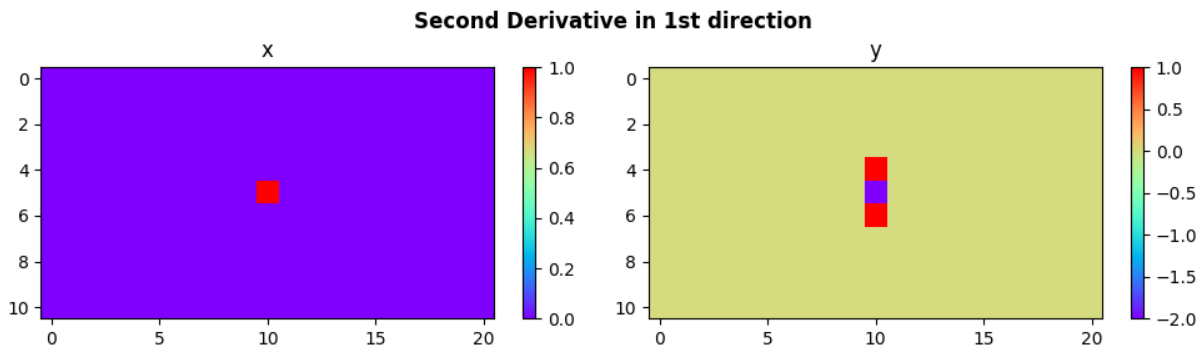


We can now do the same for the second derivative

```
A = np.zeros((nx, ny))
A[nx // 2, ny // 2] = 1.0

D2op = pylops.SecondDerivative(nx * ny, dims=(nx, ny), dir=0, dtype="float64")
B = np.reshape(D2op * A.ravel(), (nx, ny))

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle(
    "Second Derivative in 1st direction", fontsize=12, fontweight="bold", y=0.95
)
im = axs[0].imshow(A, interpolation="nearest", cmap="rainbow")
axs[0].axis("tight")
axs[0].set_title("x")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(B, interpolation="nearest", cmap="rainbow")
axs[1].axis("tight")
axs[1].set_title("y")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)
```



We can also apply the second derivative to the second direction of our data (`dir=1`)

```
D2op = pylops.SecondDerivative(nx * ny, dims=(nx, ny), dir=1, dtype="float64")
B = np.reshape(D2op * np.ndarray.flatten(A), (nx, ny))

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle(
    "Second Derivative in 2nd direction", fontsize=12, fontweight="bold", y=0.95
)
```

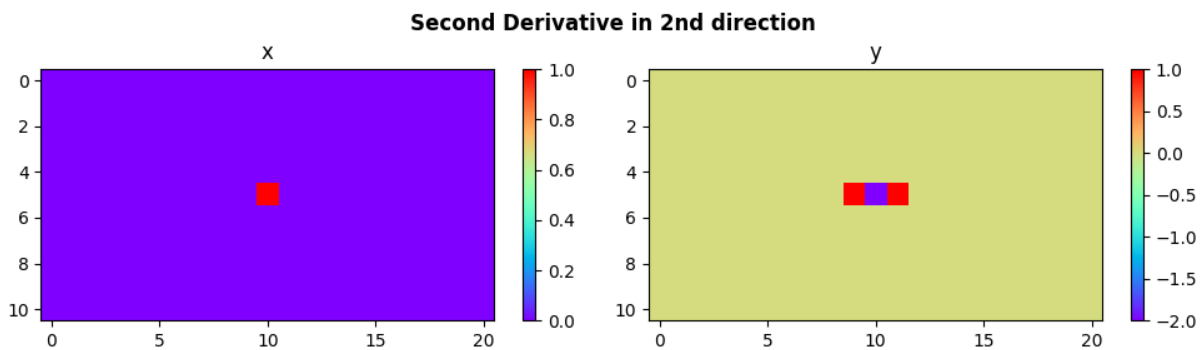
(continues on next page)

(continued from previous page)

```

)
im = axs[0].imshow(A, interpolation="nearest", cmap="rainbow")
axs[0].axis("tight")
axs[0].set_title("x")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(B, interpolation="nearest", cmap="rainbow")
axs[1].axis("tight")
axs[1].set_title("y")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)

```



We use the symmetrical Laplacian operator as well as a asymmetrical version of it (by adding more weight to the derivative along one direction)

```

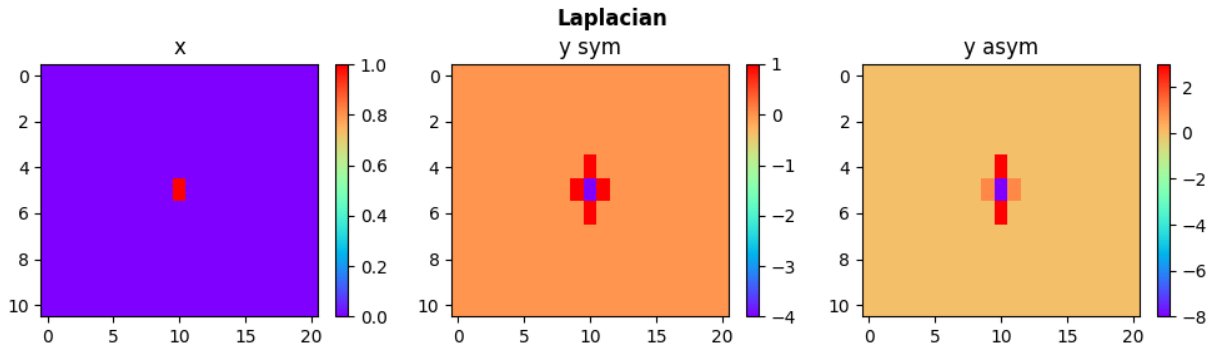
# symmetrical
L2symop = pylops.Laplacian(dims=(nx, ny), weights=(1, 1), dtype="float64")

# asymmetrical
L2asymop = pylops.Laplacian(dims=(nx, ny), weights=(3, 1), dtype="float64")

Bsym = np.reshape(L2symop * A.ravel(), (nx, ny))
Basym = np.reshape(L2asymop * A.ravel(), (nx, ny))

fig, axs = plt.subplots(1, 3, figsize=(10, 3))
fig.suptitle("Laplacian", fontsize=12, fontweight="bold", y=0.95)
im = axs[0].imshow(A, interpolation="nearest", cmap="rainbow")
axs[0].axis("tight")
axs[0].set_title("x")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(Bsym, interpolation="nearest", cmap="rainbow")
axs[1].axis("tight")
axs[1].set_title("y sym")
plt.colorbar(im, ax=axs[1])
im = axs[2].imshow(Basym, interpolation="nearest", cmap="rainbow")
axs[2].axis("tight")
axs[2].set_title("y asym")
plt.colorbar(im, ax=axs[2])
plt.tight_layout()
plt.subplots_adjust(top=0.8)

```

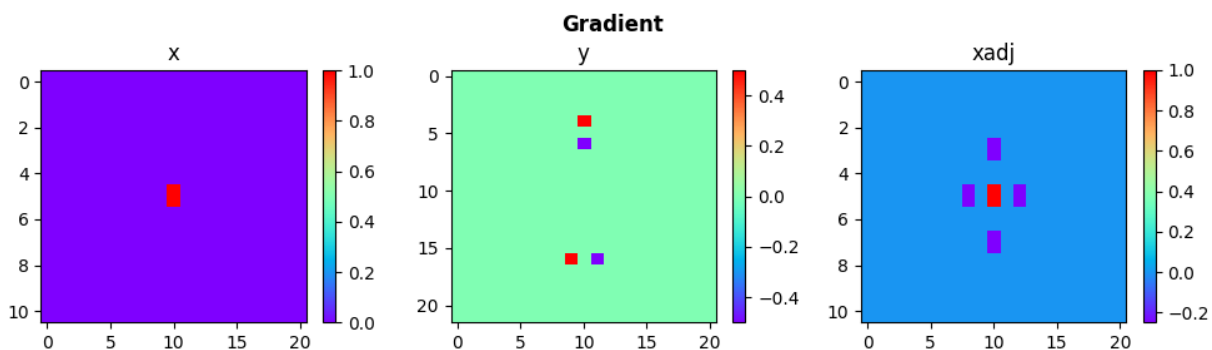


We consider now the gradient operator. Given a 2-dimensional array, this operator applies first-order derivatives on both dimensions and concatenates them.

```
Gop = pylops.Gradient(dims=(nx, ny), dtype="float64")

B = np.reshape(Gop * A.ravel(), (2 * nx, ny))
C = np.reshape(Gop.H * B.ravel(), (nx, ny))

fig, axs = plt.subplots(1, 3, figsize=(10, 3))
fig.suptitle("Gradient", fontsize=12, fontweight="bold", y=0.95)
im = axs[0].imshow(A, interpolation="nearest", cmap="rainbow")
axs[0].axis("tight")
axs[0].set_title("x")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(B, interpolation="nearest", cmap="rainbow")
axs[1].axis("tight")
axs[1].set_title("y")
plt.colorbar(im, ax=axs[1])
im = axs[2].imshow(C, interpolation="nearest", cmap="rainbow")
axs[2].axis("tight")
axs[2].set_title("xadj")
plt.colorbar(im, ax=axs[2])
plt.tight_layout()
plt.subplots_adjust(top=0.8)
```



Finally we use the Gradient operator to compute directional derivatives. We create a model which has some layering in the horizontal and vertical directions and show how the direction derivatives differs from standard derivatives

```
nx, nz = 60, 40

horlayers = np.cumsum(np.random.uniform(2, 10, 20).astype(int))
```

(continues on next page)

(continued from previous page)

```

horlayers = horlayers[horlayers < nz // 2]
nhorlayers = len(horlayers)

vertlayers = np.cumsum(np.random.uniform(2, 20, 10).astype(int))
vertlayers = vertlayers[vertlayers < nx]
nvertlayers = len(vertlayers)

A = 1500 * np.ones((nz, nx))
for top, base in zip(horlayers[:-1], horlayers[1:]):
    A[top:base] = np.random.normal(2000, 200)
for top, base in zip(vertlayers[:-1], vertlayers[1:]):
    A[horlayers[-1] :, top:base] = np.random.normal(2000, 200)

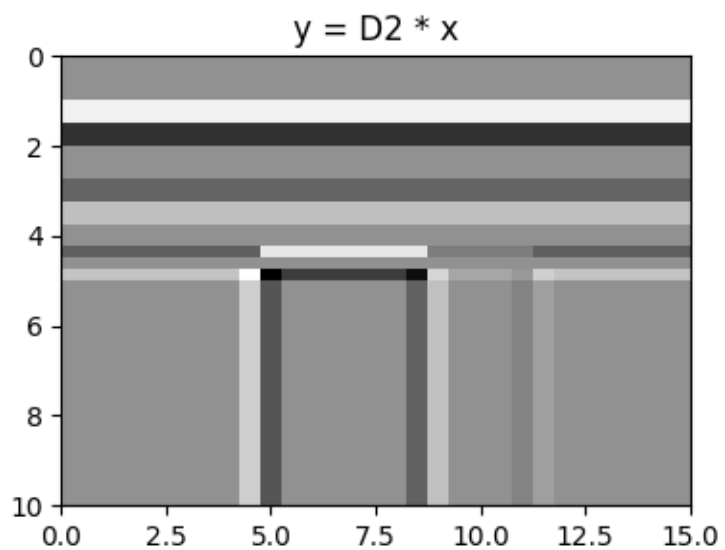
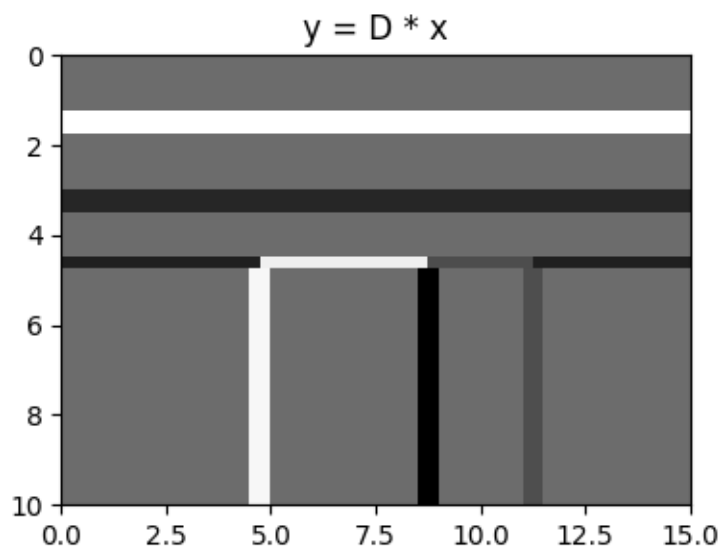
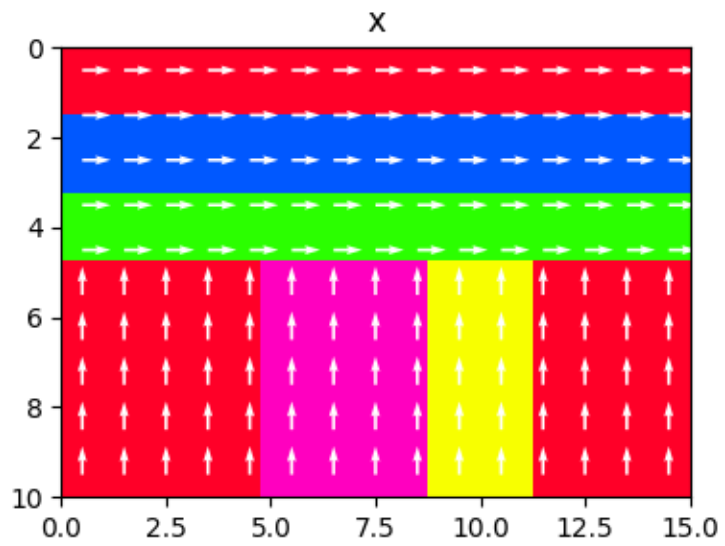
v = np.zeros((2, nz, nx))
v[0, :, horlayers[-1]] = 1
v[1, horlayers[-1] :] = 1

Ddop = pylops.FirstDirectionalDerivative((nz, nx), v=v, sampling=(nz, nx))
D2dop = pylops.SecondDirectionalDerivative((nz, nx), v=v, sampling=(nz, nx))

dirder = Ddop * A.ravel()
dirder = dirder.reshape(nz, nx)
dir2der = D2dop * A.ravel()
dir2der = dir2der.reshape(nz, nx)

jump = 4
fig, axs = plt.subplots(3, 1, figsize=(4, 9))
im = axs[0].imshow(A, cmap="gist_rainbow", extent=(0, nx // jump, nz // jump, 0))
q = axs[0].quiver(
    np.arange(nx // jump) + 0.5,
    np.arange(nz // jump) + 0.5,
    np.flipud(v[1, ::jump, ::jump]),
    np.flipud(v[0, ::jump, ::jump]),
    color="w",
    linewidths=20,
)
axs[0].set_title("x")
axs[0].axis("tight")
axs[1].imshow(dirder, cmap="gray", extent=(0, nx // jump, nz // jump, 0))
axs[1].set_title("y = D * x")
axs[1].axis("tight")
axs[2].imshow(dir2der, cmap="gray", extent=(0, nx // jump, nz // jump, 0))
axs[2].set_title("y = D2 * x")
axs[2].axis("tight")
plt.tight_layout()

```



Total running time of the script: (0 minutes 2.856 seconds)

3.6.13 Describe

This example focuses on the usage of the `pylops.utils.describe.describe` method, which allows expressing any PyLops operator into its equivalent mathematical representation. This is done with the aid of `sympy`, a Python library for symbolic computing

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

import pylops
from pylops.utils.describe import describe

plt.close("all")
```

Let's start by defining 3 PyLops operators. Note that once an operator is defined we can attach a name to the operator; by doing so, this name will be used in the mathematical description of the operator. Alternatively, the describe method will randomly choose a name for us.

```
A = pylops.MatrixMult(np.ones((10, 5)))
A.name = "A"
B = pylops.Diagonal(np.ones(5))
B.name = "A"
C = pylops.MatrixMult(np.ones((10, 5)))

# Simple operator
describe(A)

# Transpose
AT = A.T
describe(AT)

# Adjoint
AH = A.H
describe(AH)

# Scaled
A3 = 3 * A
describe(A3)

# Sum
D = A + C
describe(D)
```

Out:

```
A
where: {'A': 'MatrixMult'}
A.T
where: {'A': 'MatrixMult'}
Adjoint(A)
where: {'A': 'MatrixMult'}
3*A
where: {'A': 'MatrixMult'}
A + T
where: {'A': 'MatrixMult', 'T': 'MatrixMult'}
```

So far so good. Let's see what happens if we accidentally call two different operators with the same name. You will see that PyLops catches that and changes the name for us (and provides us with a nice warning!)

```
D = A * B
describe(D)
```

Out:

```
The user has used the same name A for two distinct operators, changing name of
↪operator Diagonal to J...
A*J
where: {'A': 'MatrixMult', 'J': 'Diagonal'}
```

We can move now to something more complicated using various composition operators

```
H = pylops.HStack((A * B, C * B))
describe(H)

H = pylops.Block([[A * B, C], [A, A]])
describe(H)
```

Out:

```
Matrix([[A*J, T*J]])
where: {'A': 'MatrixMult', 'J': 'Diagonal', 'T': 'MatrixMult'}
Matrix([
  [Matrix([[A*J, T]])],
  [ Matrix([[A, A]])]]
where: {'A': 'MatrixMult', 'J': 'Diagonal', 'T': 'MatrixMult'}
```

Finally, note that you can get the best out of the describe method if working inside a Jupyter notebook. There, the mathematical expression will be rendered using a LaTeX format! See an example [notebook](#).

Total running time of the script: (0 minutes 0.500 seconds)

3.6.14 Diagonal

This example shows how to use the `pylops.Diagonal` operator to perform *Element-wise multiplication* between the input vector and a vector `d`.

In other words, the operator acts as a diagonal operator \mathbf{D} whose elements along the diagonal are the elements of the vector `d`.

```
import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt
import numpy as np

import pylops

plt.close("all")
```

Let's define a diagonal operator `d` with increasing numbers from 0 to `N` and a unitary model `x`.

```
N = 10
d = np.arange(N)
x = np.ones(N)
```

(continues on next page)

(continued from previous page)

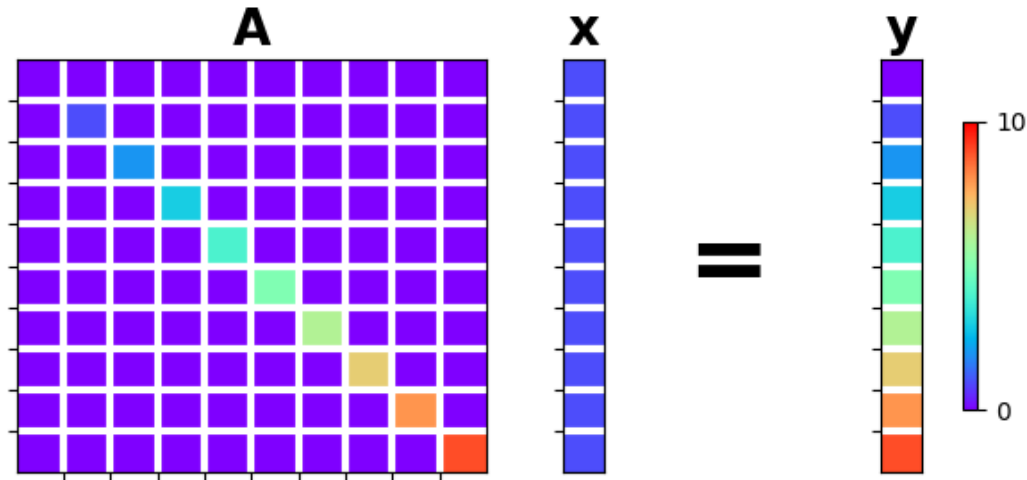
```

Dop = pylops.Diagonal(d)

y = Dop * x
y1 = Dop.H * x

gs = plt.GridSpec(1, 6)
fig = plt.figure(figsize=(7, 3))
ax = plt.subplot(gs[0, 0:3])
im = ax.imshow(Dop.matrix(), cmap="rainbow", vmin=0, vmax=N)
ax.set_title("A", size=20, fontweight="bold")
ax.set_xticks(np.arange(N - 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, 3])
ax.imshow(x[:, np.newaxis], cmap="rainbow", vmin=0, vmax=N)
ax.set_title("x", 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(y[:, np.newaxis], cmap="rainbow", vmin=0, vmax=N)
ax.set_title("y", 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([])
fig.colorbar(im, ax=ax, ticks=[0, N], pad=0.3, shrink=0.7)

```



Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f6aecf08c18>
```

Similarly we can consider the input model as composed of two or more dimensions. In this case the diagonal operator can be still applied to each element or broadcasted along a specific direction. Let's start with the simplest case where each element is multiplied by a different value

```
nx, ny = 3, 5
x = np.ones((nx, ny))
print("x =\n%s" % x)

d = np.arange(nx * ny).reshape(nx, ny)
Dop = pylops.Diagonal(d)

y = Dop * x.ravel()
y1 = Dop.H * x.ravel()

print("y = D*x =\n%s" % y.reshape(nx, ny))
print("xadj = D'*x =\n%s" % y1.reshape(nx, ny))
```

Out:

```
x =
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
y = D*x =
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]]
xadj = D'*x =
[[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]]
```

And we now broadcast

```

nx, ny = 3, 5
x = np.ones((nx, ny))
print("x =\n%s" % x)

# 1st dim
d = np.arange(nx)
Dop = pylops.Diagonal(d, dims=(nx, ny), dir=0)

y = Dop * x.ravel()
y1 = Dop.H * x.ravel()

print("1st dim: y = D*x =\n%s" % y.reshape(nx, ny))
print("1st dim: xadj = D'*x =\n%s " % y1.reshape(nx, ny))

# 2nd dim
d = np.arange(ny)
Dop = pylops.Diagonal(d, dims=(nx, ny), dir=1)

y = Dop * x.ravel()
y1 = Dop.H * x.ravel()

print("2nd dim: y = D*x =\n%s" % y.reshape(nx, ny))
print("2nd dim: xadj = D'*x =\n%s " % y1.reshape(nx, ny))

```

Out:

```

x =
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
1st dim: y = D*x =
[[0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1.]
 [2. 2. 2. 2. 2.]]
1st dim: xadj = D'*x =
[[0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1.]
 [2. 2. 2. 2. 2.]]
2nd dim: y = D*x =
[[0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]]
2nd dim: xadj = D'*x =
[[0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]]

```

Total running time of the script: (0 minutes 0.218 seconds)

3.6.15 Flip along an axis

This example shows how to use the `pylops.Flip` operator to simply flip an input signal along an axis.

```

import matplotlib.pyplot as plt
import numpy as np

```

(continues on next page)

(continued from previous page)

```
import pylops
plt.close("all")
```

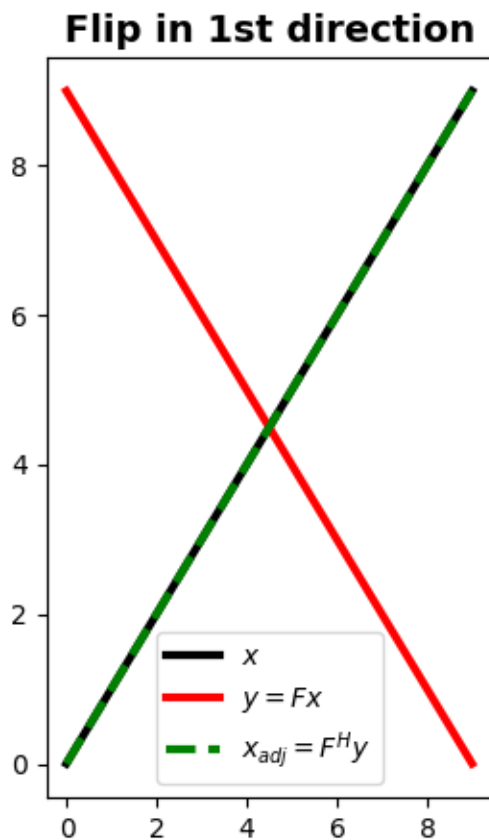
Let's start with a 1D example. Define an input signal composed of nt samples

```
nt = 10
x = np.arange(nt)
```

We can now create our flip operator and apply it to the input signal. We can also apply the adjoint to the flipped signal and we can see how for this operator the adjoint is effectively equivalent to the inverse.

```
Fop = pylops.Flip(nt)
y = Fop * x
xadj = Fop.H * y

plt.figure(figsize=(3, 5))
plt.plot(x, "k", lw=3, label=r"$x$")
plt.plot(y, "r", lw=3, label=r"$y=Fx$")
plt.plot(xadj, "--g", lw=3, label=r"$x_{adj} = F^H y$")
plt.title("Flip in 1st direction", fontsize=14, fontweight="bold")
plt.legend()
```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aee2f0208>
```

Let's now repeat the same exercise on a two dimensional signal. We will first flip the model along the first axis and then along the second axis

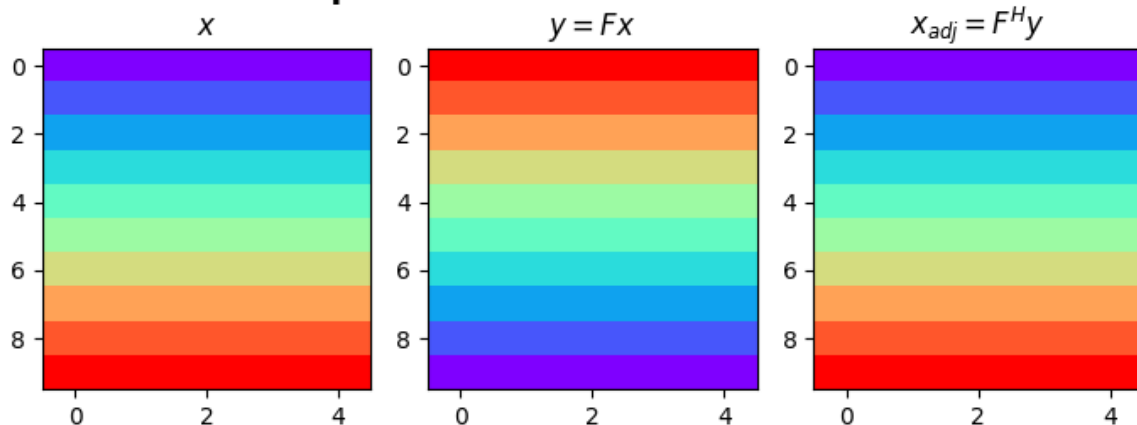
```
nt, nx = 10, 5
x = np.outer(np.arange(nt), np.ones(nx))
Fop = pylops.Flip(nt * nx, dims=(nt, nx), dir=0)
y = Fop * x.ravel()
xadj = Fop.H * y.ravel()
y = y.reshape(nt, nx)
xadj = xadj.reshape(nt, nx)

fig, axs = plt.subplots(1, 3, figsize=(7, 3))
fig.suptitle(
    "Flip in 1st direction for 2d data", fontsize=14, fontweight="bold", y=0.95
)
axs[0].imshow(x, cmap="rainbow")
axs[0].set_title(r"$x$")
axs[0].axis("tight")
axs[1].imshow(y, cmap="rainbow")
axs[1].set_title(r"$y = F x$")
axs[1].axis("tight")
axs[2].imshow(xadj, cmap="rainbow")
axs[2].set_title(r"$x_{adj} = F^H y$")
axs[2].axis("tight")
plt.tight_layout()
plt.subplots_adjust(top=0.8)

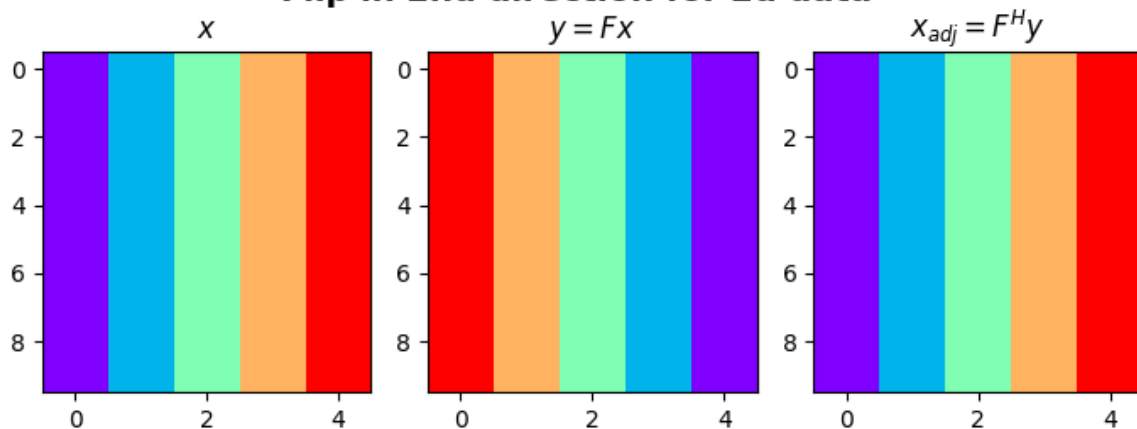
x = np.outer(np.ones(nt), np.arange(nx))
Fop = pylops.Flip(nt * nx, dims=(nt, nx), dir=1)
y = Fop * x.ravel()
xadj = Fop.H * y.ravel()
y = y.reshape(nt, nx)
xadj = xadj.reshape(nt, nx)

# sphinx_gallery_thumbnail_number = 3
fig, axs = plt.subplots(1, 3, figsize=(7, 3))
fig.suptitle(
    "Flip in 2nd direction for 2d data", fontsize=14, fontweight="bold", y=0.95
)
axs[0].imshow(x, cmap="rainbow")
axs[0].set_title(r"$x$")
axs[0].axis("tight")
axs[1].imshow(y, cmap="rainbow")
axs[1].set_title(r"$y = F x$")
axs[1].axis("tight")
axs[2].imshow(xadj, cmap="rainbow")
axs[2].set_title(r"$x_{adj} = F^H y$")
axs[2].axis("tight")
plt.tight_layout()
plt.subplots_adjust(top=0.8)
```

Flip in 1st direction for 2d data



Flip in 2nd direction for 2d data



Total running time of the script: (0 minutes 0.673 seconds)

3.6.16 Fourier Transform

This example shows how to use the `pylops.signalprocessing.FFT`, `pylops.signalprocessing.FFT2D` and `pylops.signalprocessing.FFTND` operators to apply the Fourier Transform to the model and the inverse Fourier Transform to the data.

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

import pylops

plt.close("all")
```

Let's start by applying the one dimensional FFT to a one dimensional sinusoidal signal $d(t) = \sin(2\pi f_0 t)$ using a time axis of length nt and sampling dt

```
dt = 0.005
nt = 100
```

(continues on next page)

(continued from previous page)

```

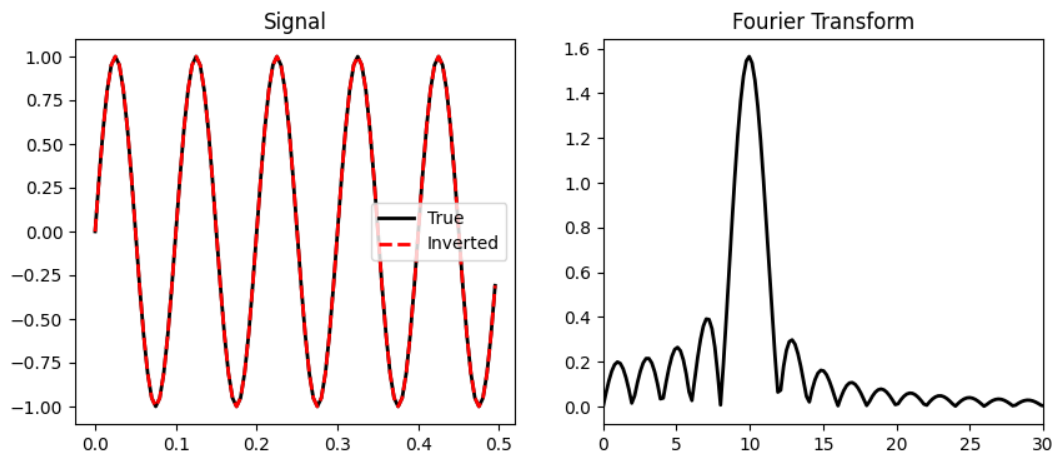
t = np.arange(nt) * dt
f0 = 10
nfft = 2 ** 10
d = np.sin(2 * np.pi * f0 * t)

FFTop = pylops.signalprocessing.FFT(dims=nt, nfft=nfft, sampling=dt, engine="numpy")
D = FFTop * d

# Adjoint = inverse for FFT
dinv = FFTop.H * D
dinv = FFTop / D

fig, axs = plt.subplots(1, 2, figsize=(10, 4))
axs[0].plot(t, d, "k", lw=2, label="True")
axs[0].plot(t, dinv, "--r", lw=2, label="Inverted")
axs[0].legend()
axs[0].set_title("Signal")
axs[1].plot(FFTop.f[: int(FFTop.nfft / 2)], np.abs(D[: int(FFTop.nfft / 2)]), "k",
           lw=2)
axs[1].set_title("Fourier Transform")
axs[1].set_xlim([0, 3 * f0])

```



Out:

(0.0, 30.0)

In this example we used numpy as our engine for the `fft` and `ifft`. PyLops implements a second engine (`engine='fftw'`) which uses the well-known `FFTW` via the python wrapper `pyfftw.FFTW`. This optimized `fft` tends to outperform the one from numpy in many cases but it is not inserted in the mandatory requirements of PyLops. If interested to use `FFTW` backend, read the `fft routines` section at [Advanced installation](#).

```

FFTop = pylops.signalprocessing.FFT(dims=nt, nfft=nfft, sampling=dt, engine="fftw")
D = FFTop * d

# Adjoint = inverse for FFT
dinv = FFTop.H * D
dinv = FFTop / D

fig, axs = plt.subplots(1, 2, figsize=(10, 4))

```

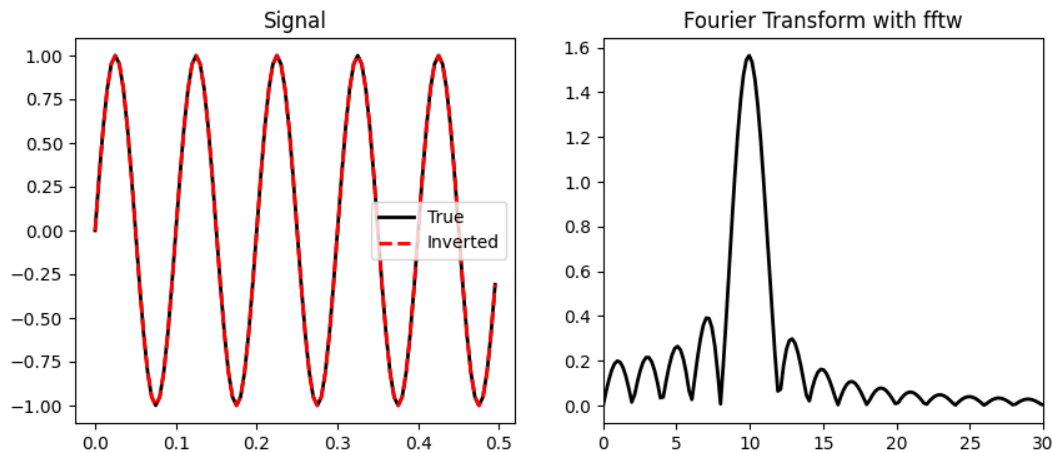
(continues on next page)

(continued from previous page)

```

axs[0].plot(t, d, "k", lw=2, label="True")
axs[0].plot(t, dinv, "--r", lw=2, label="Inverted")
axs[0].legend()
axs[0].set_title("Signal")
axs[1].plot(FFTop.f[: int(FFTop.nfft / 2)], np.abs(D[: int(FFTop.nfft / 2)]), "k", lw=2)
axs[1].set_title("Fourier Transform with fftw")
axs[1].set_xlim([0, 3 * f0])

```



Out:

(0.0, 30.0)

We can also apply the one dimensional FFT to to a two-dimensional signal (along one of the first axis)

```

dt = 0.005
nt, nx = 100, 20
t = np.arange(nt) * dt
f0 = 10
nfft = 2 ** 10
d = np.outer(np.sin(2 * np.pi * f0 * t), np.arange(nx) + 1)

FFTop = pylops.signalprocessing.FFT(dims=(nt, nx), dir=0, nfft=nfft, sampling=dt)
D = FFTop * d.ravel()

# Adjoint = inverse for FFT
dinv = FFTop.H * D
dinv = FFTop / D
dinv = np.real(dinv).reshape(nt, nx)

fig, axs = plt.subplots(2, 2, figsize=(10, 6))
axs[0][0].imshow(d, vmin=-20, vmax=20, cmap="seismic")
axs[0][0].set_title("Signal")
axs[0][0].axis("tight")
axs[0][1].imshow(np.abs(D.reshape(nfft, nx)[:200, :]), cmap="seismic")
axs[0][1].set_title("Fourier Transform")
axs[0][1].axis("tight")
axs[1][0].imshow(dinv, vmin=-20, vmax=20, cmap="seismic")
axs[1][0].set_title("Inverted")

```

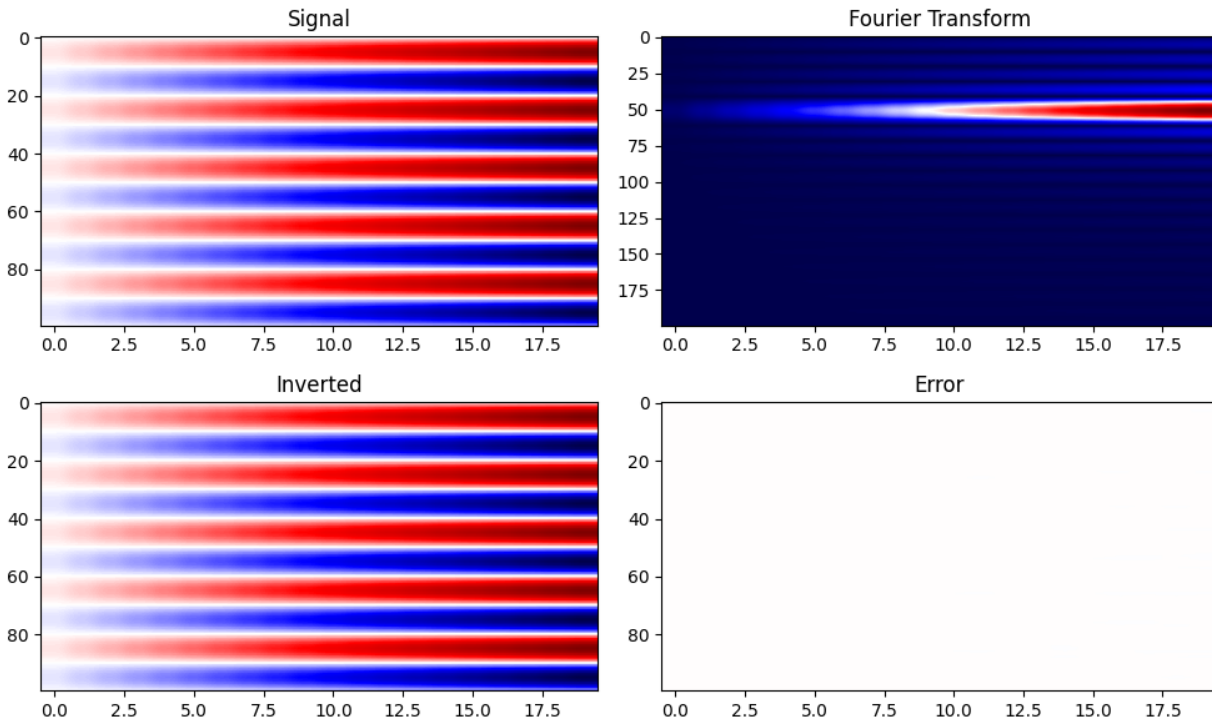
(continues on next page)

(continued from previous page)

```

axs[1][0].axis("tight")
axs[1][1].imshow(d - dinv, vmin=-20, vmax=20, cmap="seismic")
axs[1][1].set_title("Error")
axs[1][1].axis("tight")
fig.tight_layout()

```



We can also apply the two dimensional FFT to a two-dimensional signal

```

dt, dx = 0.005, 5
nt, nx = 100, 201
t = np.arange(nt) * dt
x = np.arange(nx) * dx
f0 = 10
nfft = 2 ** 10
d = np.outer(np.sin(2 * np.pi * f0 * t), np.arange(nx) + 1)

FFTop = pyllops.signalprocessing.FFT2D(
    dims=(nt, nx), nffts=(nfft, nfft), sampling=(dt, dx)
)
D = FFTop * d.ravel()

dinv = FFTop.H * D
dinv = FFTop / D
dinv = np.real(dinv).reshape(nt, nx)

fig, axs = plt.subplots(2, 2, figsize=(10, 6))
axs[0][0].imshow(d, vmin=-100, vmax=100, cmap="seismic")
axs[0][0].set_title("Signal")
axs[0][0].axis("tight")
axs[0][1].imshow(

```

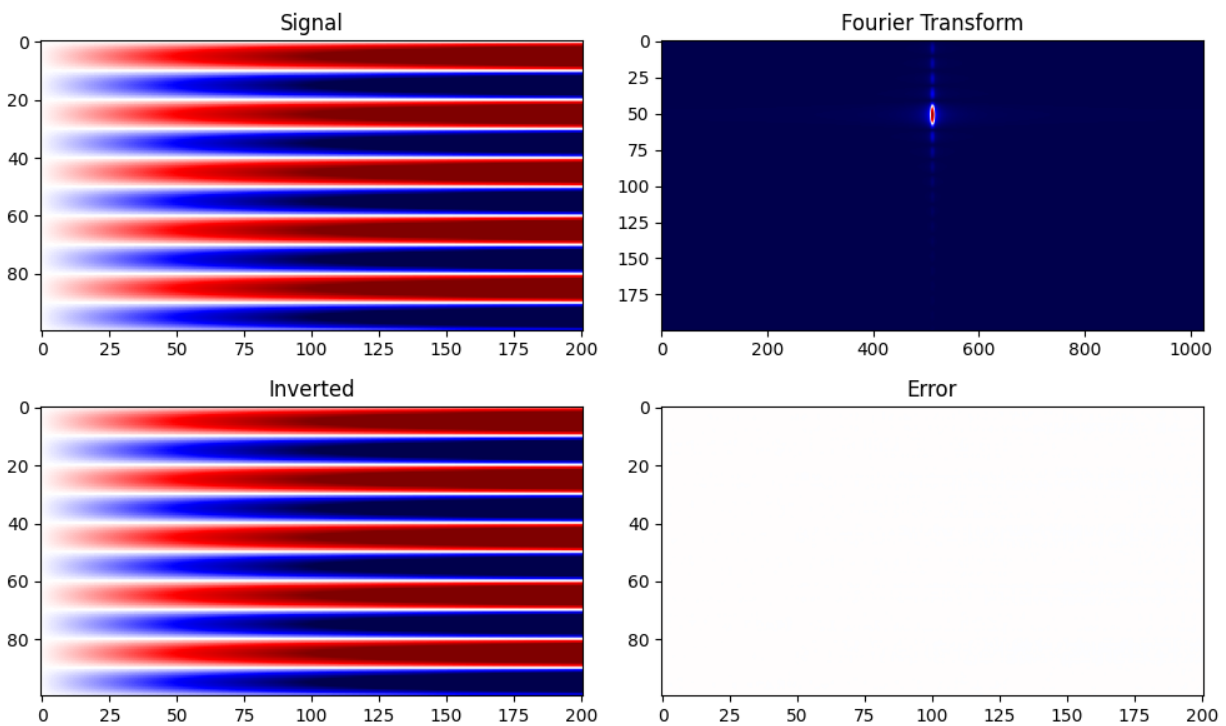
(continues on next page)

(continued from previous page)

```

np.abs(np.fft.fftshift(D.reshape(nfft, nfft), axes=1)[:200, :]), cmap="seismic"
)
axs[0][1].set_title("Fourier Transform")
axs[0][1].axis("tight")
axs[1][0].imshow(dinv, vmin=-100, vmax=100, cmap="seismic")
axs[1][0].set_title("Inverted")
axs[1][0].axis("tight")
axs[1][1].imshow(d - dinv, vmin=-100, vmax=100, cmap="seismic")
axs[1][1].set_title("Error")
axs[1][1].axis("tight")
fig.tight_layout()

```



Finally can apply the three dimensional FFT to to a three-dimensional signal

```

dt, dx, dy = 0.005, 5, 3
nt, nx, ny = 30, 21, 11
t = np.arange(nt) * dt
x = np.arange(nx) * dx
y = np.arange(ny) * dy
f0 = 10
nfft = 2 ** 6
nfftk = 2 ** 5

d = np.outer(np.sin(2 * np.pi * f0 * t), np.arange(nx) + 1)
d = np.tile(d[:, :, np.newaxis], [1, 1, ny])

FFTop = pyllops.signalprocessing.FFTND(
    dims=(nt, nx, ny), nffts=(nfft, nfftk, nfftk), sampling=(dt, dx, dy)
)
D = FFTop * d.ravel()

```

(continues on next page)

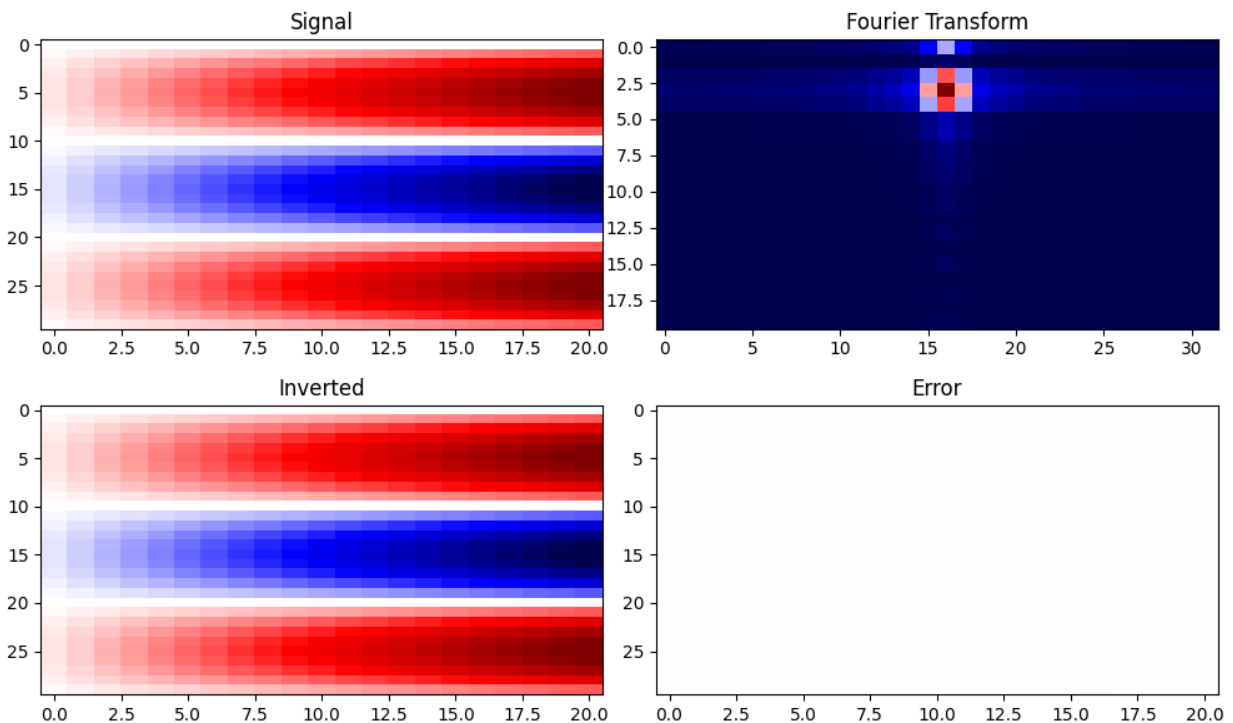
(continued from previous page)

```

dinv = FFTop.H * D
dinv = FFTop / D
dinv = np.real(dinv).reshape(nt, nx, ny)

fig, axs = plt.subplots(2, 2, figsize=(10, 6))
axs[0][0].imshow(d[:, :, ny // 2], vmin=-20, vmax=20, cmap="seismic")
axs[0][0].set_title("Signal")
axs[0][0].axis("tight")
axs[0][1].imshow(
    np.abs(np.fft.fftshift(D.reshape(nfft, nfftk, nfftk), axes=1)[:20, :, nfftk // 2]),
    cmap="seismic",
)
axs[0][1].set_title("Fourier Transform")
axs[0][1].axis("tight")
axs[1][0].imshow(dinv[:, :, ny // 2], vmin=-20, vmax=20, cmap="seismic")
axs[1][0].set_title("Inverted")
axs[1][0].axis("tight")
axs[1][1].imshow(
    d[:, :, ny // 2] - dinv[:, :, ny // 2], vmin=-20, vmax=20, cmap="seismic"
)
axs[1][1].set_title("Error")
axs[1][1].axis("tight")
fig.tight_layout()

```



Total running time of the script: (0 minutes 2.042 seconds)

3.6.17 Identity

This example shows how to use the `pylops.Identity` operator to transfer model into data and viceversa.

```
import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt
import numpy as np

import pylops

plt.close("all")
```

Let's define an identity operator **Iop** with same number of elements for data and model ($N = M$).

```
N, M = 5, 5
x = np.arange(M)
Iop = pylops.Identity(M, dtype="int")

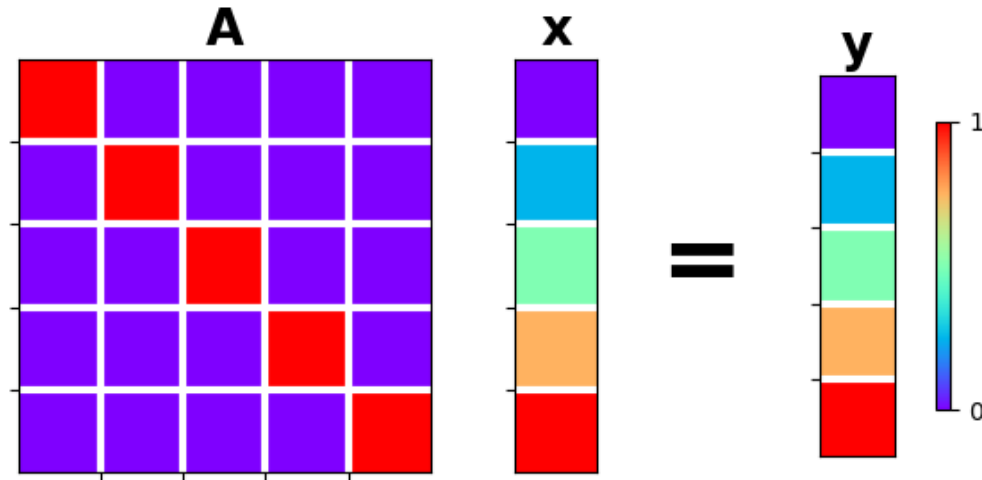
y = Iop * x
xadj = Iop.H * y

gs = pltgs.GridSpec(1, 6)
fig = plt.figure(figsize=(7, 3))
ax = plt.subplot(gs[0, 0:3])
im = ax.imshow(np.eye(N), cmap="rainbow")
ax.set_title("A", 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 = plt.subplot(gs[0, 3])
ax.imshow(x[:, np.newaxis], cmap="rainbow")
ax.set_title("x", size=20, fontweight="bold")
ax.set_xticks([])
ax.set_yticks(np.arange(M - 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(y[:, np.newaxis], cmap="rainbow")
ax.set_title("y", 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([])
```

(continues on next page)

(continued from previous page)

```
fig.colorbar(im, ax=ax, ticks=[0, 1], pad=0.3, shrink=0.7)
```



Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f6aee2f03c8>
```

Similarly we can consider the case with data bigger than model

```
N, M = 10, 5
x = np.arange(M)
Iop = pylops.Identity(N, M, dtype="int")

y = Iop * x
xadj = Iop.H * y

print("x = %s " % x)
print("I*x = %s " % y)
print("I'*y = %s " % xadj)
```

Out:

```
x = [0 1 2 3 4]
I*x = [0 1 2 3 4 0 0 0 0 0]
I'*y = [0 1 2 3 4]
```

and model bigger than data

```
N, M = 5, 10
x = np.arange(M)
Iop = pylops.Identity(N, M, dtype="int")

y = Iop * x
xadj = Iop.H * y

print("x = %s " % x)
print("I*x = %s " % y)
print("I'*y = %s " % xadj)
```

Out:

```
x = [0 1 2 3 4 5 6 7 8 9]
I*x = [0 1 2 3 4]
I'*y = [0 1 2 3 4 0 0 0 0 0]
```

Note that this operator can be useful in many real-life applications when for example we want to manipulate a subset of the model array and keep intact the rest of the array. For example:

$$\begin{bmatrix} \mathbf{A} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \mathbf{A}\mathbf{x}_1 + \mathbf{x}_2$$

Refer to the tutorial on *Optimization* for more details on this.

Total running time of the script: (0 minutes 0.164 seconds)

3.6.18 Imag

This example shows how to use the `pylops.basicoperators.Imag` operator. This operator returns the imaginary part of the data as a real value in forward mode, and the real part of the model as an imaginary value in adjoint mode (with zero real part).

```
import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt
import numpy as np

import pylops

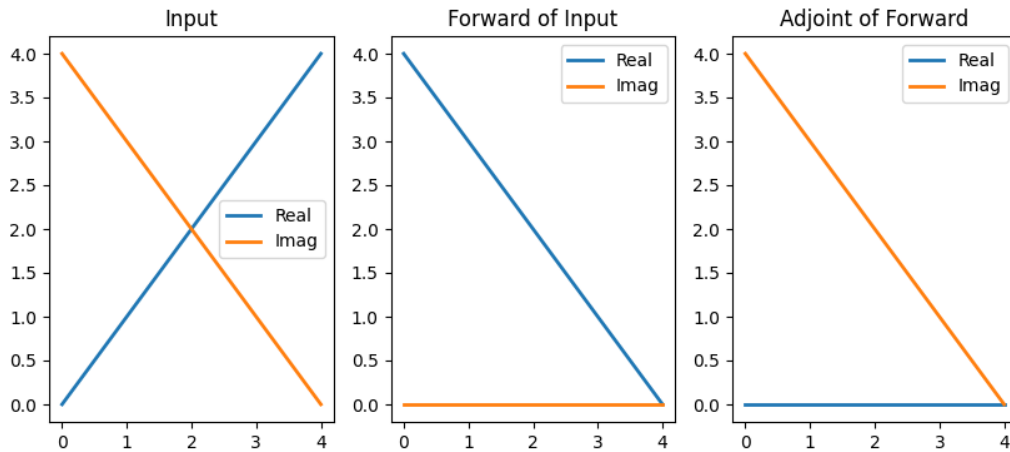
plt.close("all")
```

Let's define a `Imag` operator \Im to extract the imaginary component of the input.

```
M = 5
x = np.arange(M) + 1j * np.arange(M)[:-1]
Rop = pylops.basicoperators.Imag(M, dtype="complex128")

y = Rop * x
xadj = Rop.H * y

_, axs = plt.subplots(1, 3, figsize=(10, 4))
axs[0].plot(np.real(x), lw=2, label="Real")
axs[0].plot(np.imag(x), lw=2, label="Imag")
axs[0].legend()
axs[0].set_title("Input")
axs[1].plot(np.real(y), lw=2, label="Real")
axs[1].plot(np.imag(y), lw=2, label="Imag")
axs[1].legend()
axs[1].set_title("Forward of Input")
axs[2].plot(np.real(xadj), lw=2, label="Real")
axs[2].plot(np.imag(xadj), lw=2, label="Imag")
axs[2].legend()
axs[2].set_title("Adjoint of Forward")
```

Out:

```
Text(0.5, 1.0, 'Adjoint of Forward')
```

Total running time of the script: (0 minutes 0.260 seconds)

3.6.19 Linear Regression

This example shows how to use the `pylops.LinearRegression` operator to perform *Linear regression analysis*.

In short, linear regression is the problem of finding the best fitting coefficients, namely intercept x_0 and gradient x_1 , for this equation:

$$y_i = x_0 + x_1 t_i \quad \forall i = 0, 1, \dots, N - 1$$

As we can express this problem in a matrix form:

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

our solution can be obtained by solving the following optimization problem:

$$J = \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2$$

See documentation of `pylops.LinearRegression` for more detailed definition of the forward problem.

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

import pylops

plt.close("all")
np.random.seed(10)
```

Define the input parameters: number of samples along the t-axis (N), linear regression coefficients (x), and standard deviation of noise to be added to data (sigma).

```
N = 30
x = np.array([1.0, 2.0])
sigma = 1
```

Let's create the time axis and initialize the `pylops.LinearRegression` operator

```
t = np.arange(N, dtype="float64")
LROP = pylops.LinearRegression(t, dtype="float64")
```

We can then apply the operator in forward mode to compute our data points along the x-axis (y). We will also generate some random gaussian noise and create a noisy version of the data (yn).

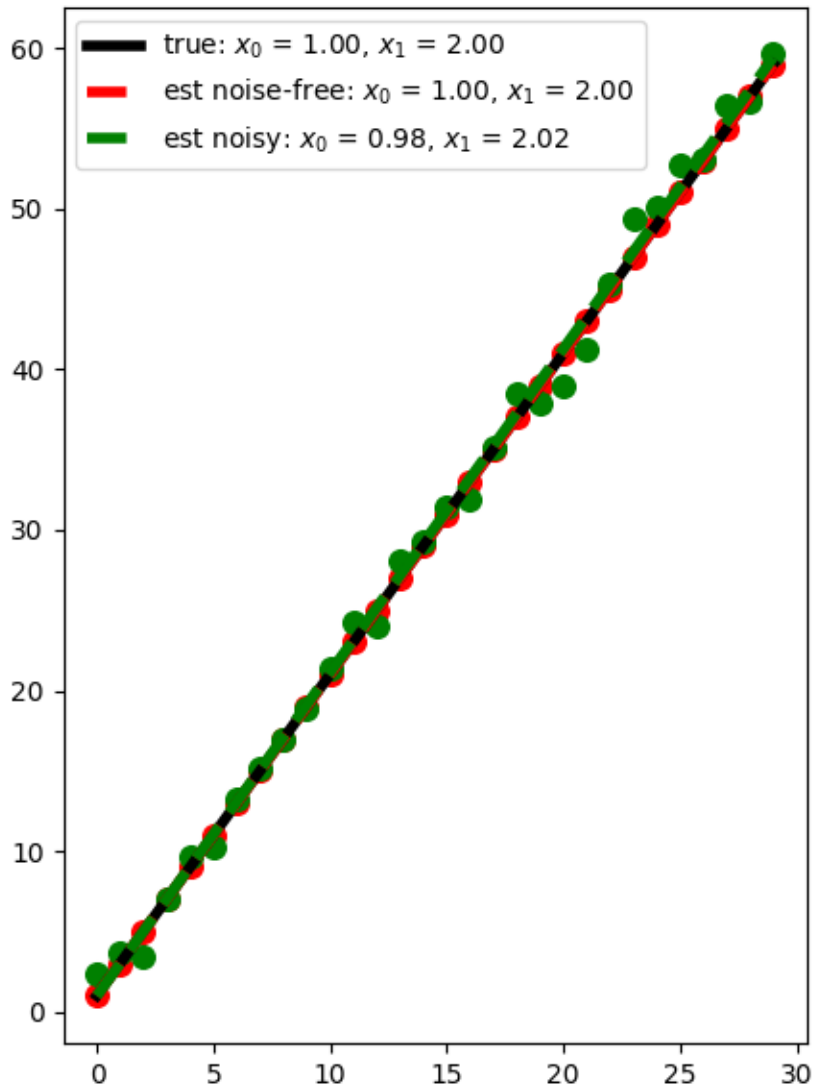
```
y = LROP * x
yn = y + np.random.normal(0, sigma, N)
```

We are now ready to solve our problem. As we are using an operator from the `pylops.LinearOperator` family, we can simply use `/`, which in this case will solve the system by means of an iterative solver (i.e., `scipy.sparse.linalg.lsqr`).

```
xest = LROP / y
xnest = LROP / yn
```

Let's plot the best fitting line for the case of noise free and noisy data

```
plt.figure(figsize=(5, 7))
plt.plot(
    np.array([t.min(), t.max()]),
    np.array([t.min(), t.max()]) * x[1] + x[0],
    "k",
    lw=4,
    label=r"true: $x_0$ = %.2f, $x_1$ = %.2f" % (x[0], x[1]),
)
plt.plot(
    np.array([t.min(), t.max()]),
    np.array([t.min(), t.max()]) * xest[1] + xest[0],
    "--r",
    lw=4,
    label=r"est noise-free: $x_0$ = %.2f, $x_1$ = %.2f" % (xest[0], xest[1]),
)
plt.plot(
    np.array([t.min(), t.max()]),
    np.array([t.min(), t.max()]) * xnest[1] + xnest[0],
    "--g",
    lw=4,
    label=r"est noisy: $x_0$ = %.2f, $x_1$ = %.2f" % (xnest[0], xnest[1]),
)
plt.scatter(t, y, c="r", s=70)
plt.scatter(t, yn, c="g", s=70)
plt.legend()
```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aed96b9e8>
```

Once that we have estimated the best fitting coefficients \mathbf{x} we can now use them to compute the y values for a different set of values along the t -axis.

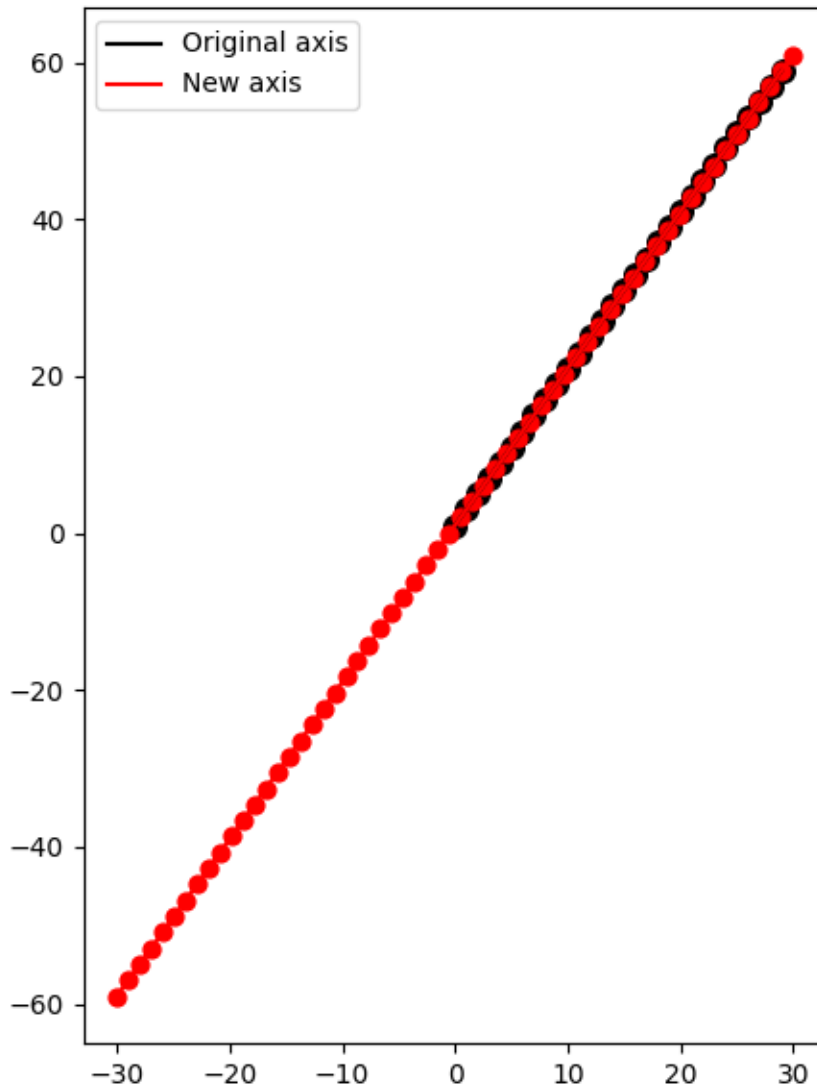
```
t1 = np.linspace(-N, N, 2 * N, dtype="float64")
y1 = LRep.apply(t1, xest)

plt.figure(figsize=(5, 7))
```

(continues on next page)

(continued from previous page)

```
plt.plot(t, y, "k", label="Original axis")
plt.plot(t1, y1, "r", label="New axis")
plt.scatter(t, y, c="k", s=70)
plt.scatter(t1, y1, c="r", s=40)
plt.legend()
```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aed2422b0>
```

We consider now the case where some of the observations have large errors. Such elements are generally referred to as

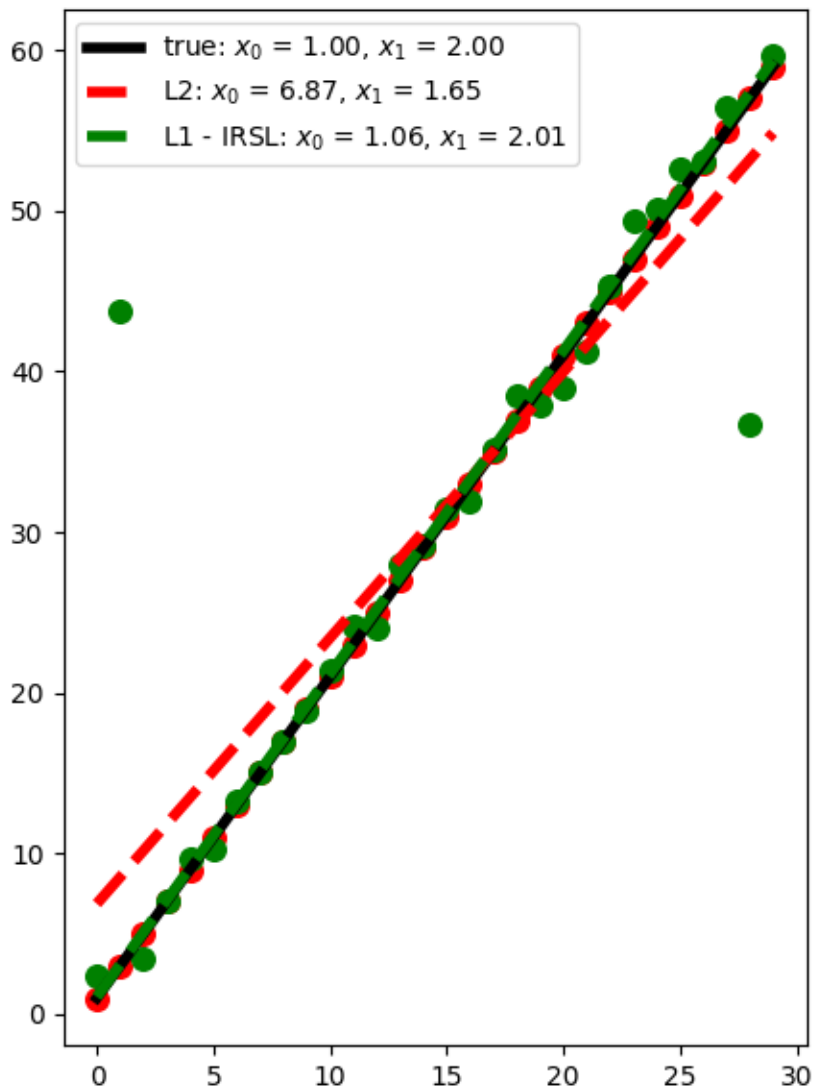
outliers and can affect the quality of the least-squares solution if not treated with care. In this example we will see how using a L1 solver such as `pylops.optimization.sparsity.IRLS` can dramatically improve the quality of the estimation of intercept and gradient.

```
# Add outliers
yn[1] += 40
yn[N - 2] -= 20

# IRLS
nouter = 20
epsR = 1e-2
epsI = 0
tolIRLS = 1e-2

xnest = LROP / yn
xirls, nouter, xirls_hist, rw_hist = pylops.optimization.sparsity.IRLS(
    LROP,
    yn,
    nouter,
    threshR=False,
    epsR=epsR,
    epsI=epsI,
    tolIRLS=tolIRLS,
    returnhistory=True,
)
print("IRLS converged at %d iterations..." % nouter)

plt.figure(figsize=(5, 7))
plt.plot(
    np.array([t.min(), t.max()]),
    np.array([t.min(), t.max()]) * x[1] + x[0],
    "k",
    lw=4,
    label=r"true: $x_0$ = %.2f, $x_1$ = %.2f" % (x[0], x[1]),
)
plt.plot(
    np.array([t.min(), t.max()]),
    np.array([t.min(), t.max()]) * xnest[1] + xnest[0],
    "--r",
    lw=4,
    label=r"L2: $x_0$ = %.2f, $x_1$ = %.2f" % (xnest[0], xnest[1]),
)
plt.plot(
    np.array([t.min(), t.max()]),
    np.array([t.min(), t.max()]) * xirls[1] + xirls[0],
    "--g",
    lw=4,
    label=r"L1 - IRLS: $x_0$ = %.2f, $x_1$ = %.2f" % (xirls[0], xirls[1]),
)
plt.scatter(t, y, c="r", s=70)
plt.scatter(t, yn, c="g", s=70)
plt.legend()
```



Out:

```
IRLS converged at 14 iterations...
<matplotlib.legend.Legend object at 0x7f6aec450668>
```

Let's finally take a look at the convergence of IRLS. First we visualize the evolution of intercept and gradient

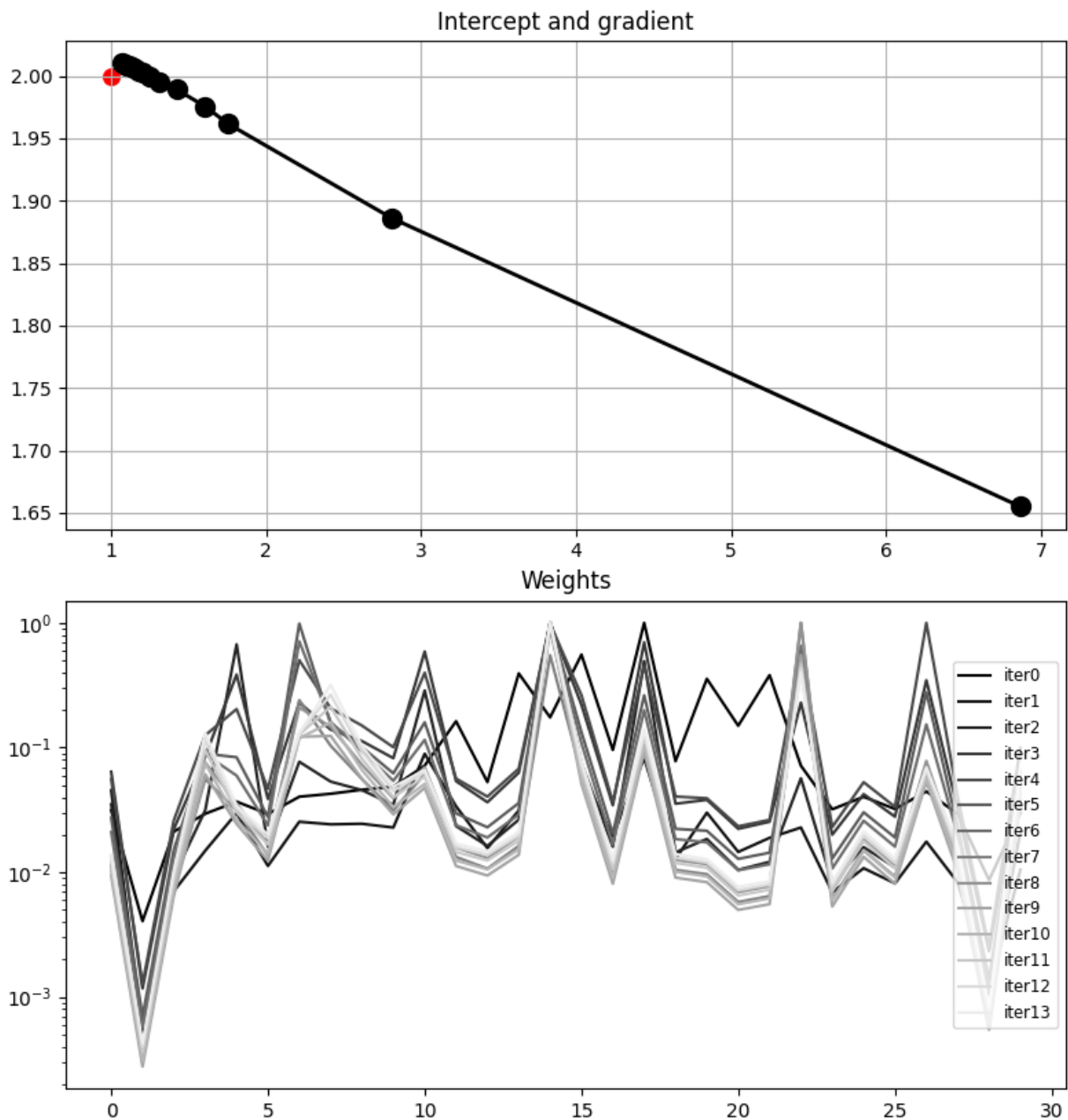
```
fig, axs = plt.subplots(2, 1, figsize=(8, 10))
fig.suptitle("IRLS evolution", fontsize=14, fontweight="bold", y=0.95)
axs[0].plot(xirls_hist[:, 0], xirls_hist[:, 1], "-k", lw=2, ms=20)
```

(continues on next page)

(continued from previous page)

```
axs[0].scatter(x[0], x[1], c="r", s=70)
axs[0].set_title("Intercept and gradient")
axs[0].grid()
for iiter in range(nouter):
    axs[1].semilogy(
        rw_hist[iiter],
        color=(iiter / nouter, iiter / nouter, iiter / nouter),
        label="iter%d" % iiter,
    )
axs[1].set_title("Weights")
axs[1].legend(loc=5, fontsize="small")
plt.tight_layout()
plt.subplots_adjust(top=0.8)
```

IRLS evolution



Total running time of the script: (0 minutes 1.077 seconds)

3.6.20 MP, OMP, ISTA and FISTA

This example shows how to use the `pylops.optimization.sparsity.OMP`, `pylops.optimization.sparsity.IRLS`, `pylops.optimization.sparsity.ISTA`, and `pylops.optimization.sparsity.FISTA` solvers.

These solvers can be used when the model to retrieve is supposed to have a sparse representation in a certain domain. MP and OMP use a L0 norm and mathematically translates to solving the following constrained problem:

$$\|\mathbf{O}\mathbf{p}\mathbf{x} - \mathbf{b}\|_2 \leq \sigma,$$

while IRLS, ISTA and FISTA solve an unconstrained problem with a L1 regularization term:

$$J = \|\mathbf{d} - \mathbf{O}\mathbf{p}\mathbf{x}\|_2 + \epsilon \|\mathbf{x}\|_1$$

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

import pylops

plt.close("all")
np.random.seed(0)
```

Let's start with a simple example, where we create a dense mixing matrix and a sparse signal and we use OMP and ISTA to recover such a signal. Note that the mixing matrix leads to an underdetermined system of equations ($N < M$) so being able to add some extra prior information regarding the sparsity of our desired model is essential to be able to invert such a system.

```
N, M = 15, 20
A = np.random.randn(N, M)
A = A / np.linalg.norm(A, axis=0)
Aop = pylops.MatrixMult(A)

x = np.random.rand(M)
x[x < 0.9] = 0
y = Aop * x

# MP/OMP
eps = 1e-2
maxit = 500
x_mp = pylops.optimization.sparsity.OMP(Aop, y, maxit, niter_inner=0, sigma=1e-4)[0]
x_omp = pylops.optimization.sparsity.OMP(Aop, y, maxit, sigma=1e-4)[0]

# IRLS
x_irls = pylops.optimization.sparsity.IRLS(
    Aop, y, 50, epsI=1e-5, kind="model", **dict(iter_lim=10)
)[0]

# ISTA
x_ista = pylops.optimization.sparsity.ISTA(
    Aop, y, maxit, eps=eps, tol=1e-3, returninfo=True
)[0]

fig, ax = plt.subplots(1, 1, figsize=(8, 3))
m, s, b = ax.stem(x, linefmt="k", basefmt="k", markerfmt="ko", label="True")
plt.setp(m, markersize=15)
m, s, b = ax.stem(x_mp, linefmt="--c", basefmt="--c", markerfmt="co", label="MP")
```

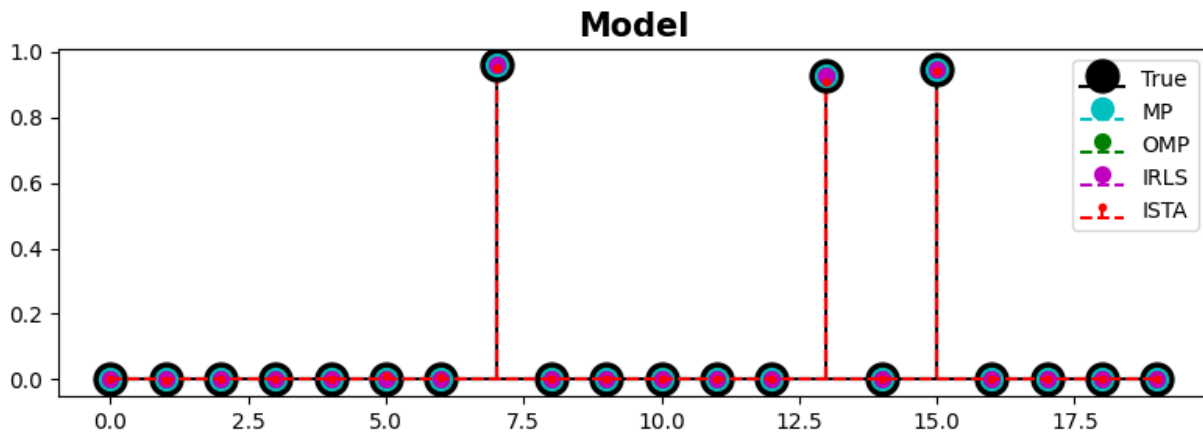
(continues on next page)

(continued from previous page)

```

plt.setp(m, markersize=10)
m, s, b = ax.stem(x_omp, linefmt="--g", basefmt="--g", markerfmt="go", label="OMP")
plt.setp(m, markersize=7)
m, s, b = ax.stem(x_irls, linefmt="--m", basefmt="--m", markerfmt="mo", label="IRLS")
plt.setp(m, markersize=7)
m, s, b = ax.stem(x_ista, linefmt="--r", basefmt="--r", markerfmt="ro", label="ISTA")
plt.setp(m, markersize=3)
ax.set_title("Model", size=15, fontweight="bold")
ax.legend()
plt.tight_layout()

```



We now consider a more interesting problem, *wavelet deconvolution* from a signal that we assume being composed by a train of spikes convolved with a certain wavelet. We will see how solving such a problem with a least-squares solver such as `pylops.optimization.leastsquares.RegularizedInversion` does not produce the expected results (especially in the presence of noisy data), conversely using the `pylops.optimization.sparsity.ISTA` and `pylops.optimization.sparsity.FISTA` solvers allows us to successfully retrieve the input signal even in the presence of noise. `pylops.optimization.sparsity.FISTA` shows faster convergence which is particularly useful for this problem.

```

nt = 61
dt = 0.004
t = np.arange(nt) * dt
x = np.zeros(nt)
x[10] = -0.4
x[int(nt / 2)] = 1
x[nt - 20] = 0.5

h, th, hcenter = pylops.utils.wavelets.ricker(t[:101], f0=20)

Cop = pylops.signalprocessing.Convolve1D(nt, h=h, offset=hcenter, dtype="float32")
y = Cop * x
yn = y + np.random.normal(0, 0.1, y.shape)

# noise free
xls = Cop / y

xomp, nitero, costo = pylops.optimization.sparsity.OMP(
    Cop, y, niter_outer=200, sigma=1e-8
)

```

(continues on next page)

(continued from previous page)

```

xista, niteri, costi = pylops.optimization.sparsity.ISTA(
    Cop, y, niter=400, eps=5e-1, tol=1e-8, returninfo=True
)

fig, ax = plt.subplots(1, 1, figsize=(8, 3))
ax.plot(t, x, "k", lw=8, label=r"$x$")
ax.plot(t, y, "r", lw=4, label=r"$y=Ax$")
ax.plot(t, xls, "--g", lw=4, label=r"$x_{LS}$")
ax.plot(t, xomp, "--b", lw=4, label=r"$x_{OMP}$ (niter=%d)" % nitero)
ax.plot(t, xista, "--m", lw=4, label=r"$x_{ISTA}$ (niter=%d)" % niteri)

ax.set_title("Noise-free deconvolution", fontsize=14, fontweight="bold")
ax.legend()
plt.tight_layout()

# noisy
xls = pylops.optimization.leastsquares.RegularizedInversion(
    Cop, [], yn, returninfo=False, **dict(damp=1e-1, atol=1e-3, iter_lim=100, show=0)
)

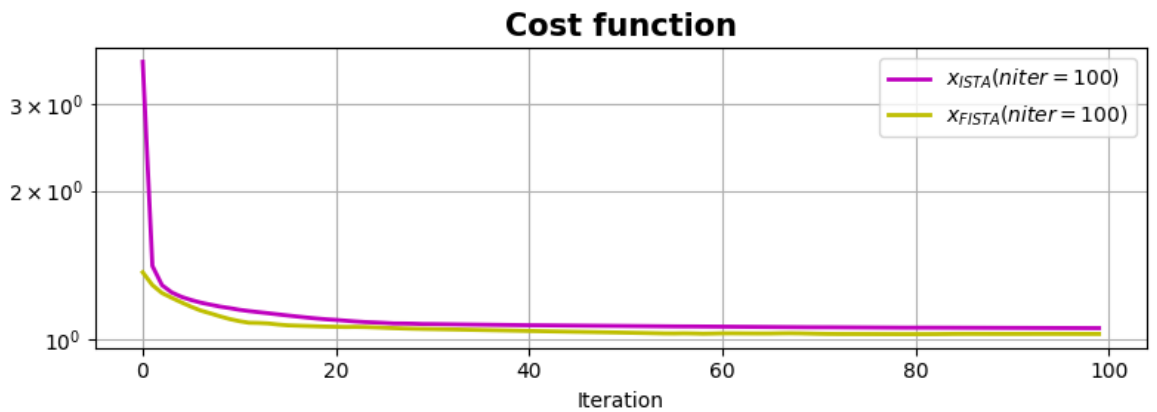
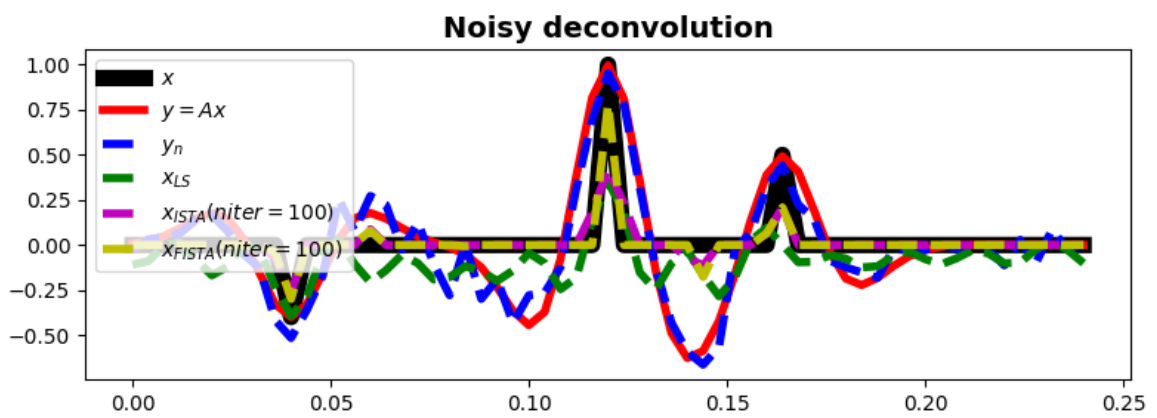
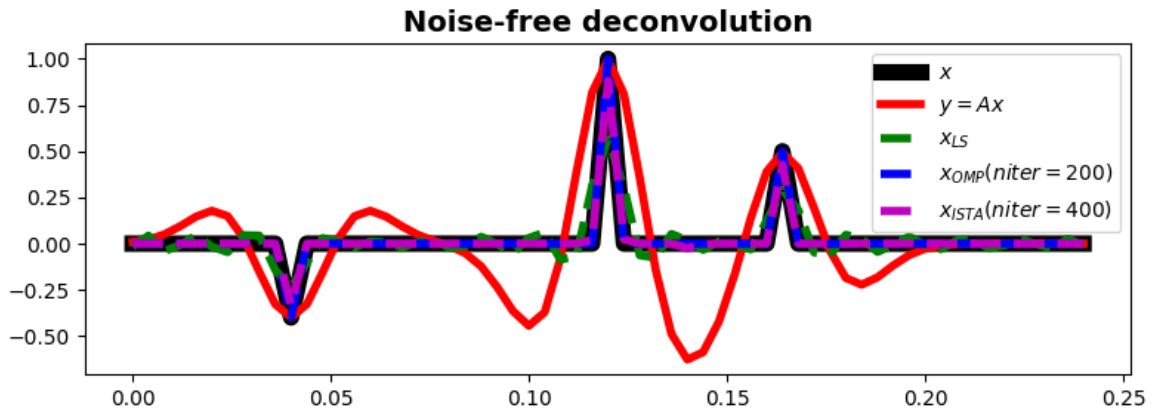
xista, niteri, costi = pylops.optimization.sparsity.ISTA(
    Cop, yn, niter=100, eps=5e-1, tol=1e-5, returninfo=True
)

xfista, niterf, costf = pylops.optimization.sparsity.FISTA(
    Cop, yn, niter=100, eps=5e-1, tol=1e-5, returninfo=True
)

fig, ax = plt.subplots(1, 1, figsize=(8, 3))
ax.plot(t, x, "k", lw=8, label=r"$x$")
ax.plot(t, y, "r", lw=4, label=r"$y=Ax$")
ax.plot(t, yn, "--b", lw=4, label=r"$y_n$")
ax.plot(t, xls, "--g", lw=4, label=r"$x_{LS}$")
ax.plot(t, xista, "--m", lw=4, label=r"$x_{ISTA}$ (niter=%d)" % niteri)
ax.plot(t, xfista, "--y", lw=4, label=r"$x_{FISTA}$ (niter=%d)" % niterf)
ax.set_title("Noisy deconvolution", fontsize=14, fontweight="bold")
ax.legend()
plt.tight_layout()

fig, ax = plt.subplots(1, 1, figsize=(8, 3))
ax.semilogy(costi, "m", lw=2, label=r"$x_{ISTA}$ (niter=%d)" % niteri)
ax.semilogy(costf, "y", lw=2, label=r"$x_{FISTA}$ (niter=%d)" % niterf)
ax.set_title("Cost function", size=15, fontweight="bold")
ax.set_xlabel("Iteration")
ax.legend()
ax.grid(True, which="both")
plt.tight_layout()

```



Total running time of the script: (0 minutes 1.825 seconds)

3.6.21 Matrix Multiplication

This example shows how to use the `pylops.MatrixMult` operator to perform *Matrix inversion* of the following linear system.

$$y = Ax$$

You will see that since this operator is a simple overloading to a `numpy.ndarray` object, the solution of the linear system can be obtained via both direct inversion (i.e., by means explicit solver such as `scipy.linalg.solve` or `scipy.linalg.lstsq`) and iterative solver (i.e., from `scipy.sparse.linalg.lsqr`).

Note that in case of rectangular **A**, an exact inverse does not exist and a least-square solution is computed instead.

```
import warnings

import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt
import numpy as np
from scipy.sparse import rand
from scipy.sparse.linalg import lsqr

import pylops

plt.close("all")
warnings.filterwarnings("ignore")
# sphinx_gallery_thumbnail_number = 2
```

Let's define the sizes of the matrix **A** (N and M) and fill the matrix with random numbers

```
N, M = 20, 20
A = np.random.normal(0, 1, (N, M))
Aop = pylops.MatrixMult(A, dtype="float64")

x = np.ones(M)
```

We can now apply the forward operator to create the data vector **y** and use `/` to solve the system by means of an explicit solver.

```
y = Aop * x
xest = Aop / y
```

Let's visually plot the system of equations we just solved.

```
gs = pltgs.GridSpec(1, 6)
fig = plt.figure(figsize=(7, 3))
ax = plt.subplot(gs[0, 0])
ax.imshow(y[:, np.newaxis], cmap="rainbow")
ax.set_title("y", 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, 1])
ax.text(
    0.35,
    0.5,
    "=",
    horizontalalignment="center",
    verticalalignment="center",
    size=40,
    fontweight="bold",
)
ax.axis("off")
ax = plt.subplot(gs[0, 2:5])
```

(continues on next page)

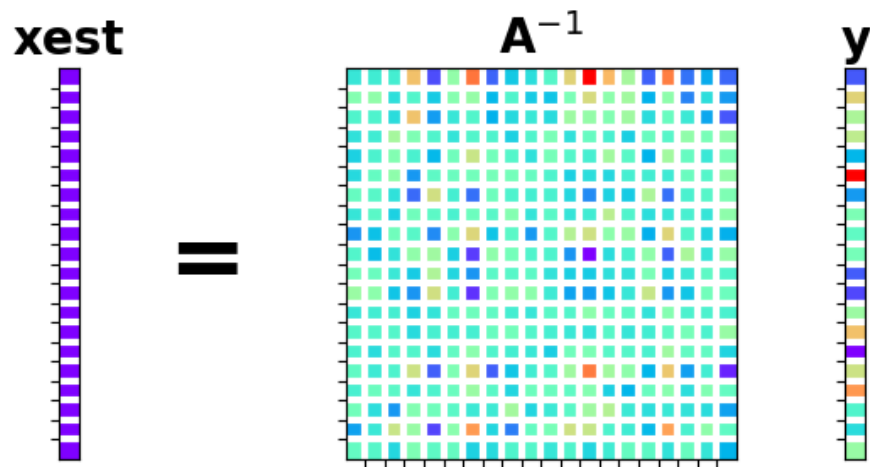
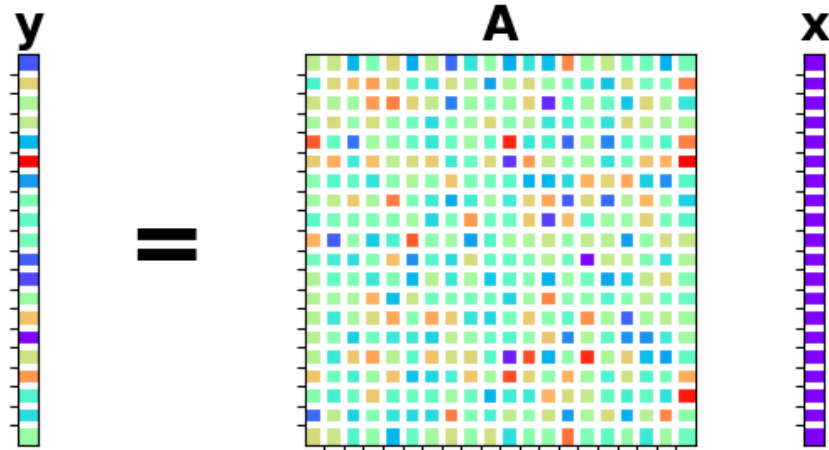
(continued from previous page)

```

ax.imshow(Aop.A, cmap="rainbow")
ax.set_title("A", 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 = plt.subplot(gs[0, 5])
ax.imshow(x[:, np.newaxis], cmap="rainbow")
ax.set_title("x", size=20, fontweight="bold")
ax.set_xticks([])
ax.set_yticks(np.arange(M - 1) + 0.5)
ax.grid(linewidth=3, color="white")
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])

gs = plt.GridSpec(1, 6)
fig = plt.figure(figsize=(7, 3))
ax = plt.subplot(gs[0, 0])
ax.imshow(x[:, np.newaxis], cmap="rainbow")
ax.set_title("xest", size=20, fontweight="bold")
ax.set_xticks([])
ax.set_yticks(np.arange(M - 1) + 0.5)
ax.grid(linewidth=3, color="white")
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
ax = plt.subplot(gs[0, 1])
ax.text(
    0.35,
    0.5,
    "=",
    horizontalalignment="center",
    verticalalignment="center",
    size=40,
    fontweight="bold",
)
ax.axis("off")
ax = plt.subplot(gs[0, 2:5])
ax.imshow(Aop.inv(), cmap="rainbow")
ax.set_title(r"A$^{-1}$", 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 = plt.subplot(gs[0, 5])
ax.imshow(y[:, np.newaxis], cmap="rainbow")
ax.set_title("y", 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([])

```



Out:

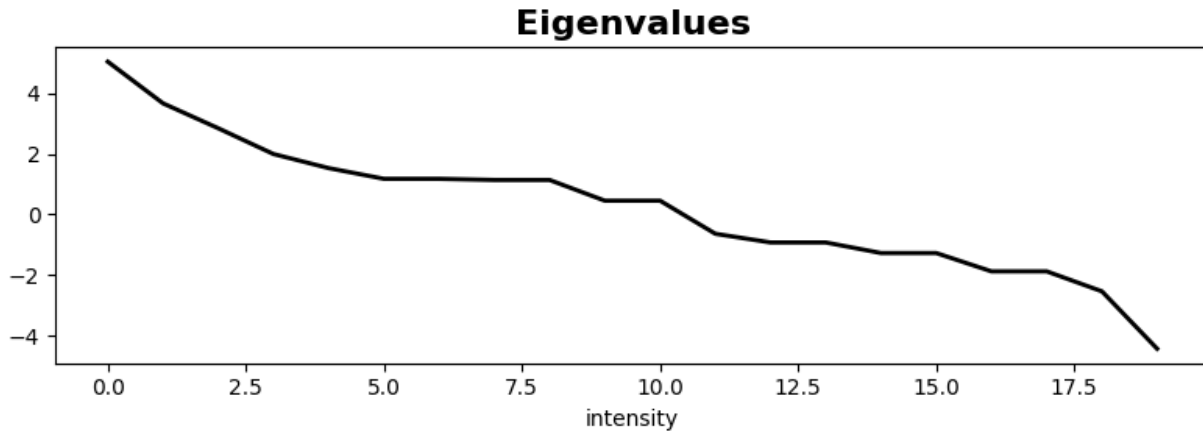
```
[Text(0, 0.5, ''), Text(0, 1.5, ''), Text(0, 2.5, ''), Text(0, 3.5, ''), Text(0, 4.5,
↪ ''), Text(0, 5.5, ''), Text(0, 6.5, ''), Text(0, 7.5, ''), Text(0, 8.5, ''), Text(0,
↪ 9.5, ''), Text(0, 10.5, ''), Text(0, 11.5, ''), Text(0, 12.5, ''), Text(0, 13.5, '
↪ '), Text(0, 14.5, ''), Text(0, 15.5, ''), Text(0, 16.5, ''), Text(0, 17.5, ''), ↪
↪ Text(0, 18.5, '')]

```

Let's also plot the matrix eigenvalues

```
plt.figure(figsize=(8, 3))
plt.plot(Aop.eigs(), "k", lw=2)
plt.title("Eigenvalues", size=16, fontweight="bold")
plt.xlabel("#eigenvalue")
plt.xlabel("intensity")
plt.tight_layout()

```



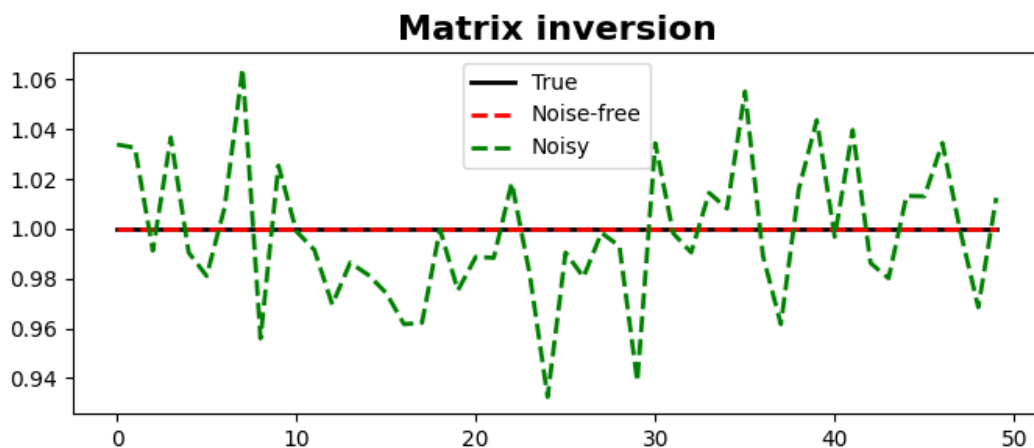
We can also repeat the same exercise for a non-square matrix

```
N, M = 200, 50
A = np.random.normal(0, 1, (N, M))
x = np.ones(M)

Aop = pylops.MatrixMult(A, dtype="float64")
y = Aop * x
yn = y + np.random.normal(0, 0.3, N)

xest = Aop / y
xnest = Aop / yn

plt.figure(figsize=(8, 3))
plt.plot(x, "k", lw=2, label="True")
plt.plot(xest, "--r", lw=2, label="Noise-free")
plt.plot(xnest, "--g", lw=2, label="Noisy")
plt.title("Matrix inversion", size=16, fontweight="bold")
plt.legend()
```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aec4cc6a0>
```

And we can also use a sparse matrix from the `scipy.sparse` family of sparse matrices.


```

N, M = 5, 5
A = rand(N, M, density=0.75)
x = np.ones(M)

Aop = pylops.MatrixMult(A, dtype="float64")
y = Aop * x
xest = Aop / y

print("A= %s" % Aop.A.todense())
print("A^-1=", Aop.inv().todense())
print("eigs=", Aop.eigs())
print("x= %s" % x)
print("y= %s" % y)
print("lsqr solution xest= %s" % xest)

```

Out:

```

A= [[0.09004394 0.07465359 0.          0.87688481 0.47238513]
 [0.79938802 0.85640696 0.71661781 0.85258904 0.          ]
 [0.          0.14734131 0.56655652 0.16975416 0.          ]
 [0.29282169 0.46547813 0.90723735 0.62796526 0.          ]
 [0.50711649 0.9563702  0.77100806 0.01124646 0.          ]]
A^-1= [[ 0.          13.07319194 29.81875979 -25.7426719  -3.77140228]
 [ 0.          -10.01880875 -25.46927387 20.41578795  4.00446478]
 [ 0.           3.89143177 12.16350133 -8.55004722 -1.19720943]
 [ 0.          -4.29168649 -12.5984235 10.81561744  0.51995029]
 [ 2.11691676  7.05799755 21.72748426 -18.39641109 -0.87913918]]
eigs= [ 1.91944139+0.j          0.50019806+0.j          -0.17658613-0.24554532j]
x= [1. 1. 1. 1. 1.]
y= [1.51396747 3.22500183 0.88365199 2.29350244 2.24574121]
lsqr solution xest= [1. 1. 1. 1. 1.]

```

Finally, in several circumstances the input model \mathbf{x} may be more naturally arranged as a matrix or a multi-dimensional array and it may be desirable to apply the same matrix to every columns of the model. This can be mathematically expressed as:

$$\mathbf{y} = \begin{bmatrix} \mathbf{A} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix}$$

and apply it using the same implementation of the `pylops.MatrixMult` operator by simply telling the operator how we want the model to be organized through the `dims` input parameter.

```

A = np.array([[1.0, 2.0], [4.0, 5.0]])
x = np.array([[1.0, 1.0], [2.0, 2.0], [3.0, 3.0]])

Aop = pylops.MatrixMult(A, dims=(3,), dtype="float64")
y = Aop * x.ravel()

xest, istop, itn, r1norm, r2norm = lsqr(Aop, y, damp=1e-10, iter_lim=10, show=0)[0:5]
xest = xest.reshape(3, 2)

print("A= %s" % A)
print("x= %s" % x)
print("y= %s" % y)
print("lsqr solution xest= %s" % xest)

```

Out:

```
A= [[1. 2.]
     [4. 5.]]
x= [[1. 1.]
     [2. 2.]
     [3. 3.]]
y= [ 5.  7.  8. 14. 19. 23.]
lsqr solution xest= [[1. 1.]
                     [2. 2.]
                     [3. 3.]]
```

Total running time of the script: (0 minutes 0.951 seconds)

3.6.22 Multi-Dimensional Convolution

This example shows how to use the `pylops.waveeqprocessing.MDC` operator to convolve a 3D kernel with an input seismic data. The resulting data is a blurred version of the input data and the problem of removing such blurring is referred to as *Multi-dimensional Deconvolution (MDD)* and its implementation is discussed in more details in the **MDD** tutorial.

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

import pylops
from pylops.utils.seismicevents import hyperbolic2d, makeaxis
from pylops.utils.tapers import taper3d
from pylops.utils.wavelets import ricker

plt.close("all")
```

Let's start by creating a set of hyperbolic events to be used as our MDC kernel

```
# Input parameters
par = {
    "ox": -300,
    "dx": 10,
    "nx": 61,
    "oy": -500,
    "dy": 10,
    "ny": 101,
    "ot": 0,
    "dt": 0.004,
    "nt": 400,
    "f0": 20,
    "nfmax": 200,
}

t0_m = 0.2
vrms_m = 1100.0
amp_m = 1.0

t0_G = (0.2, 0.5, 0.7)
vrms_G = (1200.0, 1500.0, 2000.0)
amp_G = (1.0, 0.6, 0.5)
```

(continues on next page)

(continued from previous page)

```

# 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"]]

# Move frequency/time to first axis
m, mwav = m.T, mwav.T
Gwav_fft = Gwav_fft.transpose(2, 0, 1)

# Create operator
MDCop = pylops.waveeqprocessing.MDC(
    Gwav_fft,
    nt=2 * par["nt"] - 1,
    nv=1,
    dt=0.004,
    dr=1.0,
    transpose=False,
    dtype="float32",
)

# Create data
d = MDCop * m.ravel()
d = d.reshape(2 * par["nt"] - 1, par["ny"])

# Apply adjoint operator to data
madj = MDCop.H * d.ravel()
madj = madj.reshape(2 * par["nt"] - 1, par["nx"])

```

Finally let's display the operator, input model, data and adjoint model

```

fig, axs = plt.subplots(1, 2, figsize=(9, 6))
axs[0].imshow(

```

(continues on next page)

(continued from previous page)

```

        Gwav2[int(par["ny"] / 2)].T,
        aspect="auto",
        interpolation="nearest",
        cmap="gray",
        vmin=-Gwav2.max(),
        vmax=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")
    axs[1].set_ylabel("t")
    axs[1].imshow(
        Gwav2[:, int(par["nx"] / 2)].T,
        aspect="auto",
        interpolation="nearest",
        cmap="gray",
        vmin=-Gwav2.max(),
        vmax=Gwav2.max(),
        extent=(y.min(), y.max(), t2.max(), t2.min()),
    )
    axs[1].set_title("G - inline view", fontsize=15)
    axs[1].set_xlabel("s")
    axs[1].set_ylabel("t")
    fig.tight_layout()

fig, axs = plt.subplots(1, 3, figsize=(9, 6))
axs[0].imshow(
    mwav,
    aspect="auto",
    interpolation="nearest",
    cmap="gray",
    vmin=-mwav.max(),
    vmax=mwav.max(),
    extent=(x.min(), x.max(), t2.max(), t2.min()),
)
axs[0].set_title(r"$m$", fontsize=15)
axs[0].set_xlabel("r")
axs[0].set_ylabel("t")
axs[1].imshow(
    d,
    aspect="auto",
    interpolation="nearest",
    cmap="gray",
    vmin=-d.max(),
    vmax=d.max(),
    extent=(x.min(), x.max(), t2.max(), t2.min()),
)
axs[1].set_title(r"$d$", fontsize=15)
axs[1].set_xlabel("s")
axs[1].set_ylabel("t")
axs[2].imshow(
    madj,
    aspect="auto",
    interpolation="nearest",
    cmap="gray",
    vmin=-madj.max(),
    vmax=madj.max(),

```

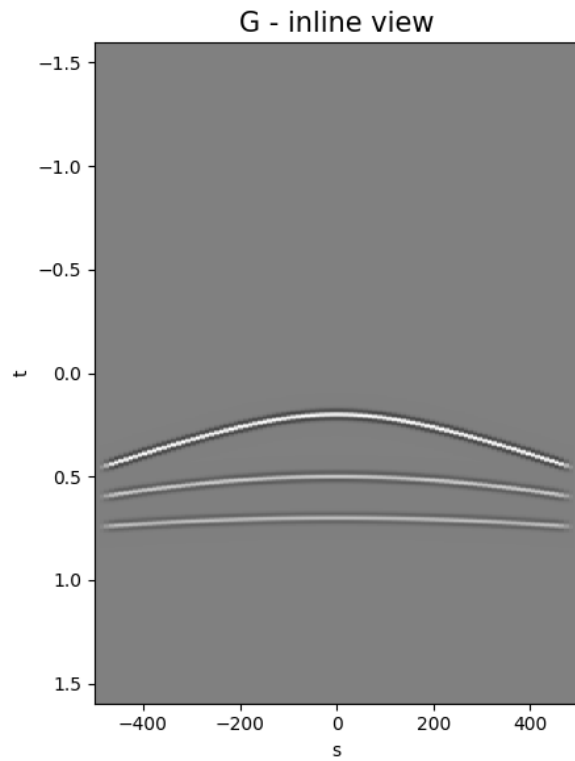
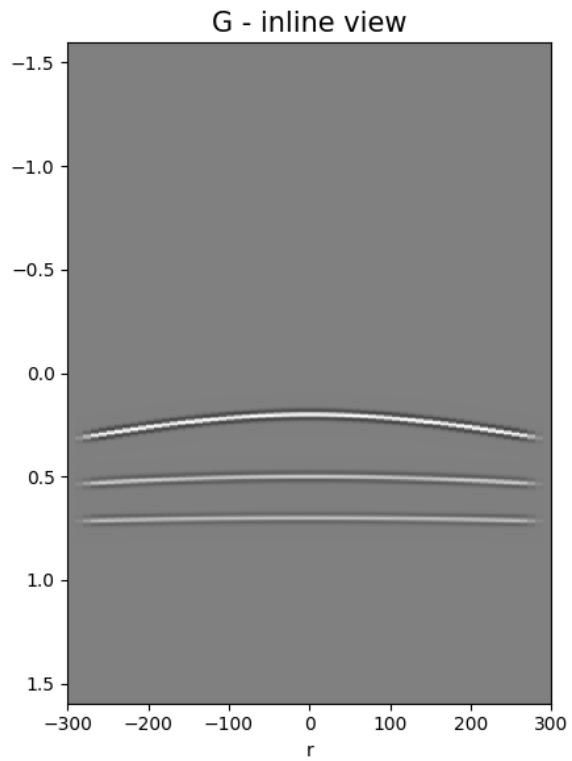
(continues on next page)

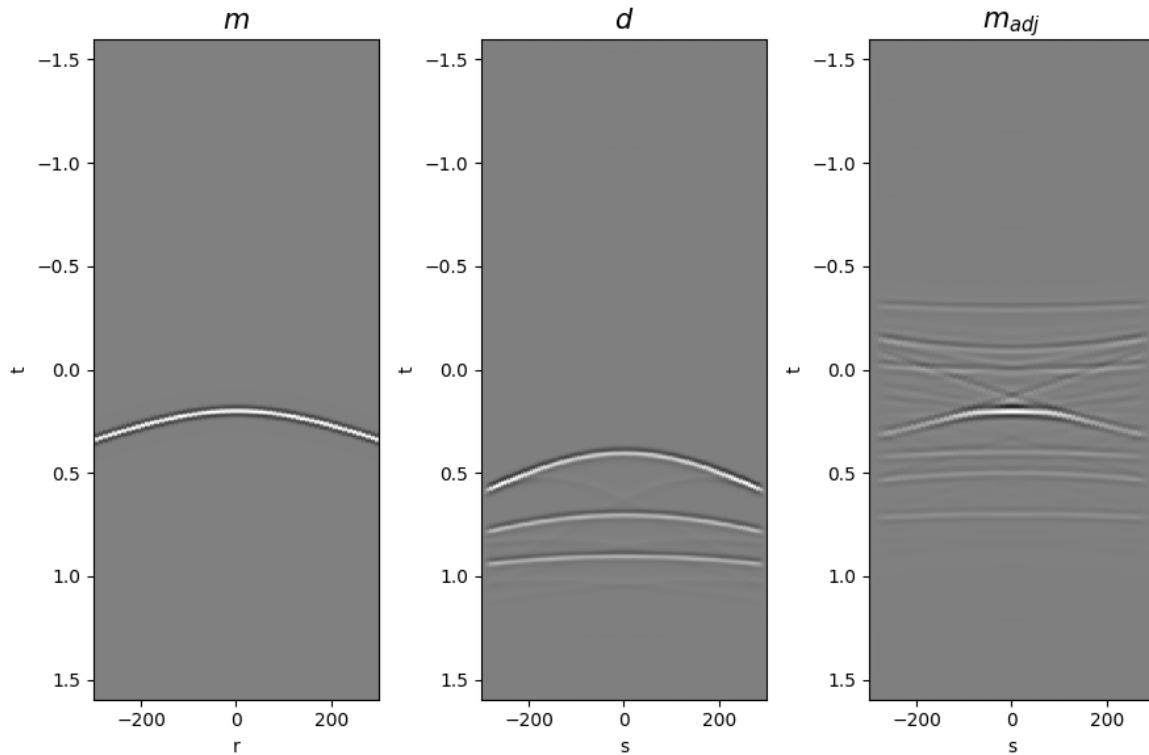
(continued from previous page)

```

    extent=(x.min(), x.max(), t2.max(), t2.min()),
)
axs[2].set_title(r"$m_{adj}$", fontsize=15)
axs[2].set_xlabel("s")
axs[2].set_ylabel("t")
fig.tight_layout()

```





Total running time of the script: (0 minutes 1.227 seconds)

3.6.23 Normal Moveout (NMO) Correction

This example shows how to create your own operator for performing normal moveout (NMO) correction to a seismic record. We will perform classic NMO using an operator created from scratch, as well as using the `pylops.Spread` operator.

```
from math import floor
from time import time

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.axes_grid1 import ImageGrid, make_axes_locatable
from numba import jit, prange
from scipy.interpolate import griddata
from scipy.ndimage import gaussian_filter

from pylops import LinearOperator, Spread
from pylops.utils import dottest
from pylops.utils.seismicevents import hyperbolic2d, makeaxis
from pylops.utils.wavelets import ricker

def create_colorbar(im, ax):
    divider = make_axes_locatable(ax)
    cax = divider.append_axes("right", size="5%", pad=0.1)
    cb = fig.colorbar(im, cax=cax, orientation="vertical")
    return cax, cb
```

Given a common-shot or common-midpoint (CMP) record, the objective of NMO correction is to “flatten” events, that is, align events at later offsets to that of the zero offset. NMO has long been a staple of seismic data processing, used even today for initial velocity analysis and QC purposes. In addition, it can be the domain of choice for many useful processing steps, such as angle muting.

To get started, let us create a 2D seismic dataset containing some hyperbolic events representing reflections from flat reflectors. Events are created with a true RMS velocity, which we will be using as if we picked them from, for example, a semblance panel.

```
par = dict(ox=0, dx=40, nx=80, ot=0, dt=0.004, nt=520)
t, _, x, _ = makeaxis(par)

t0s_true = np.array([0.5, 1.22, 1.65])
vrms_true = np.array([2000.0, 2400.0, 2500.0])
amps = np.array([1, 0.2, 0.5])

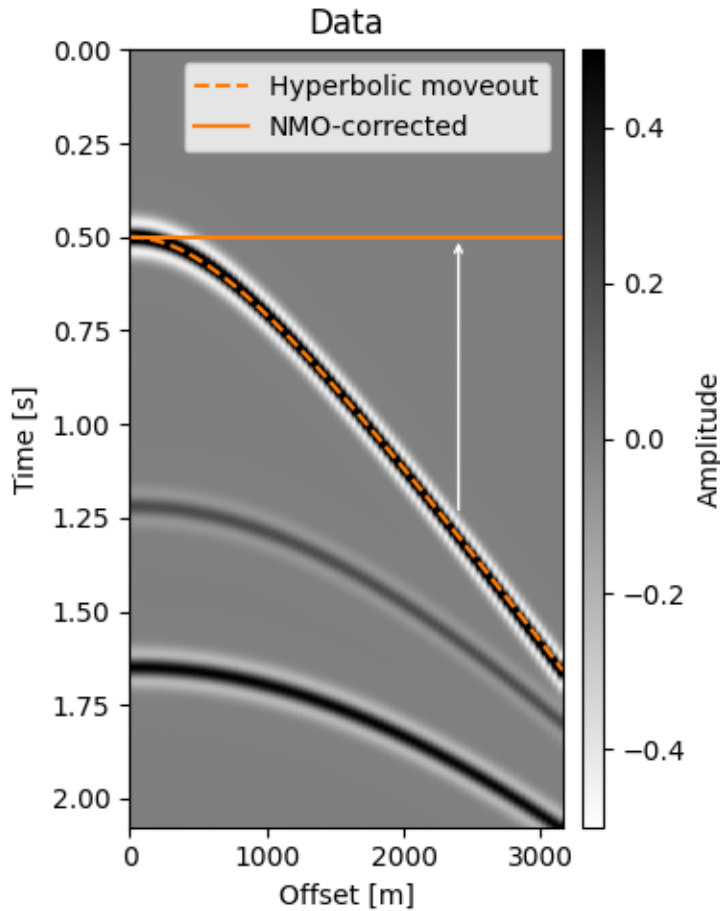
freq = 10 # Hz
wav, *_ = ricker(t[:41], f0=freq)

_, data = hyperbolic2d(x, t, t0s_true, vrms_true, amp=amps, wav=wav)
```

```
# NMO correction plot
pclip = 0.5
dmax = np.max(np.abs(data))
opts = dict(
    cmap="gray_r",
    extent=[x[0], x[-1], t[-1], t[0]],
    aspect="auto",
    vmin=-pclip * dmax,
    vmax=pclip * dmax,
)

# Offset-dependent travelttime of the first hyperbolic event
t_nmo_ev1 = np.sqrt(t0s_true[0] ** 2 + (x / vrms_true[0]) ** 2)

fig, ax = plt.subplots(figsize=(4, 5))
vmax = np.max(np.abs(data))
im = ax.imshow(data.T, **opts)
ax.plot(x, t_nmo_ev1, "C1--", label="Hyperbolic moveout")
ax.plot(x, t0s_true[0] + x * 0, "C1", label="NMO-corrected")
idx = 3 * par["nx"] // 4
ax.annotate(
    "",
    xy=(x[idx], t0s_true[0]),
    xycoords="data",
    xytext=(x[idx], t_nmo_ev1[idx]),
    textcoords="data",
    fontsize=7,
    arrowprops=dict(edgecolor="w", arrowstyle="->", shrinkA=10),
)
ax.set(title="Data", xlabel="Offset [m]", ylabel="Time [s]")
cax, _ = create_colorbar(im, ax)
cax.set_ylabel("Amplitude")
ax.legend()
fig.tight_layout()
```



NMO correction consists of applying an offset- and time-dependent shift to each sample of the trace in such a way that all events corresponding to the same reflection will be located at the same time intercept after correction.

An arbitrary hyperbolic event at position (t, h) is linked to its zero-offset traveltime t_0 by the following equation

$$t(x) = \sqrt{t_0^2 + \frac{h^2}{v_{\text{rms}}^2(t_0)}}$$

Our strategy in applying the correction is to loop over our time axis (which we will associate to t_0) and respective RMS velocities and, for each offset, move the sample at $t(x)$ to location $t_0(x) \equiv t_0$. In the figure above, we are considering a single $t_0 = 0.5\text{s}$ which would have values along the dotted curve (i.e., $t(x)$) moved to t_0 for every offset.

Notice that we need NMO velocities for each sample of our time axis. In this example, we actually only have 3 samples, when we need nt samples. In practice, we would have many more samples, but probably not one for each nt . To resolve this issue, we will interpolate these 3 samples to all samples of our time axis (or, more accurately, their slownesses to preserve traveltimes).

```
def interpolate_vrms(t0_picks, vrms_picks, taxis, smooth=None):
    assert len(t0_picks) == len(vrms_picks)

    # Sampled points in time axis
    points = np.zeros((len(t0_picks) + 2,))
    points[0] = taxis[0]
    points[-1] = taxis[-1]
```

(continues on next page)

(continued from previous page)

```

points[1:-1] = t0_picks

# Sampled values of slowness (in s/km)
values = np.zeros((len(vrms_picks) + 2,))
values[0] = 1000.0 / vrms_picks[0] # Use first slowness before t0_picks[0]
values[-1] = 1000.0 / vrms_picks[-1] # Use the last slowness after t0_picks[-1]
values[1:-1] = 1000.0 / vrms_picks

slowness = griddata(points, values, taxis, method="linear")
if smooth is not None:
    slowness = gaussian_filter(slowness, sigma=smooth)

return 1000.0 / slowness

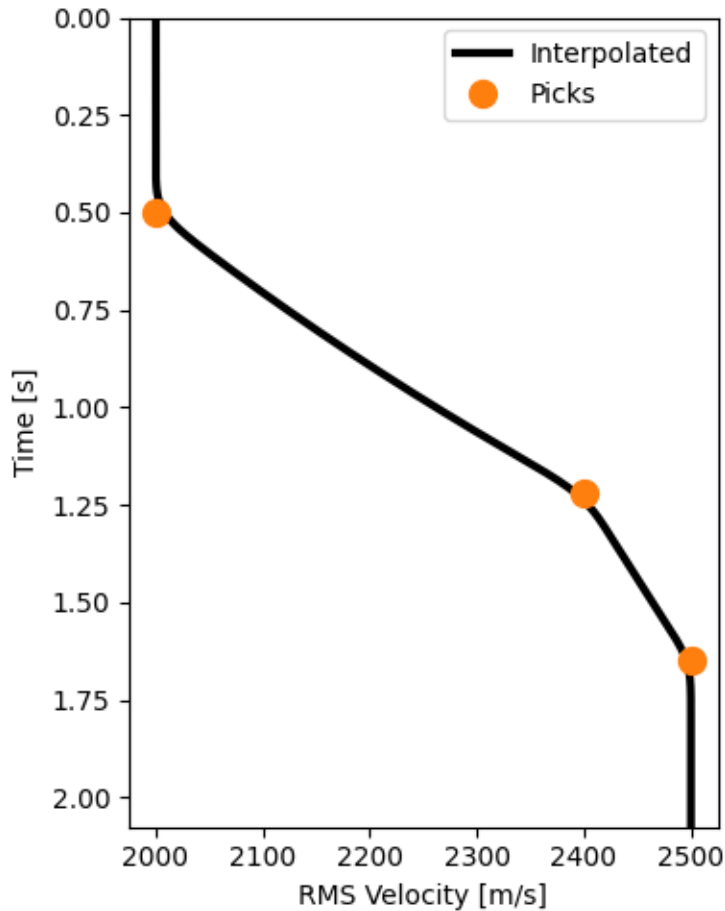
vel_t = interpolate_vrms(t0s_true, vrms_true, t, smooth=11)

```

```

# Plot interpolated RMS velocities which will be used for NMO
fig, ax = plt.subplots(figsize=(4, 5))
ax.plot(vel_t, t, "k", lw=3, label="Interpolated", zorder=-1)
ax.plot(vrms_true, t0s_true, "C1o", markersize=10, label="Picks")
ax.invert_yaxis()
ax.set(xlabel="RMS Velocity [m/s]", ylabel="Time [s]", ylim=[t[-1], t[0]])
ax.legend()
fig.tight_layout()

```



NMO from scratch

We are very close to building our NMO correction, we just need to take care of one final issue. When moving the sample from $t(x)$ to t_0 , we know that, by definition, t_0 is always on our time axis grid. In contrast, $t(x)$ may not fall exactly on a multiple of Δt (our time axis sampling). Suppose its nearest sample smaller than itself (floor) is i . Instead of moving only sample i , we will be moving samples both samples i and $i+1$ with an appropriate weight to account for how far $t(x)$ is from $i \cdot \Delta t$ and $(i+1) \cdot \Delta t$.

```
@jit(nopython=True, fastmath=True, nogil=True, parallel=True)
def nmo_forward(data, taxis, haxis, vels_rms):
    dt = taxis[1] - taxis[0]
    ot = taxis[0]
    nt = len(taxis)
    nh = len(haxis)

    dnmo = np.zeros_like(data)

    # Parallel outer loop on slow axis
    for ih in prange(nh):
        h = haxis[ih]
        for it0, (t0, vrms) in enumerate(zip(taxis, vels_rms)):
            # Compute NMO traveltime
            tx = np.sqrt(t0 ** 2 + (h / vrms) ** 2)
```

(continues on next page)

(continued from previous page)

```

        it_frac = (tx - ot) / dt # Fractional index
        it_floor = floor(it_frac)
        it_ceil = it_floor + 1
        w = it_frac - it_floor
        if 0 <= it_floor and it_ceil < nt: # it_floor and it_ceil must be valid
            # Linear interpolation
            dnmo[ih, it0] += (1 - w) * data[ih, it_floor] + w * data[ih, it_ceil]
    return dnmo

dnmo = nmo_forward(data, t, x, vel_t) # Compile and run

# Time execution
start = time()
nmo_forward(data, t, x, vel_t)
end = time()

print(f"Ran in {1e6*(end-start):.0f} μs")

```

Out:

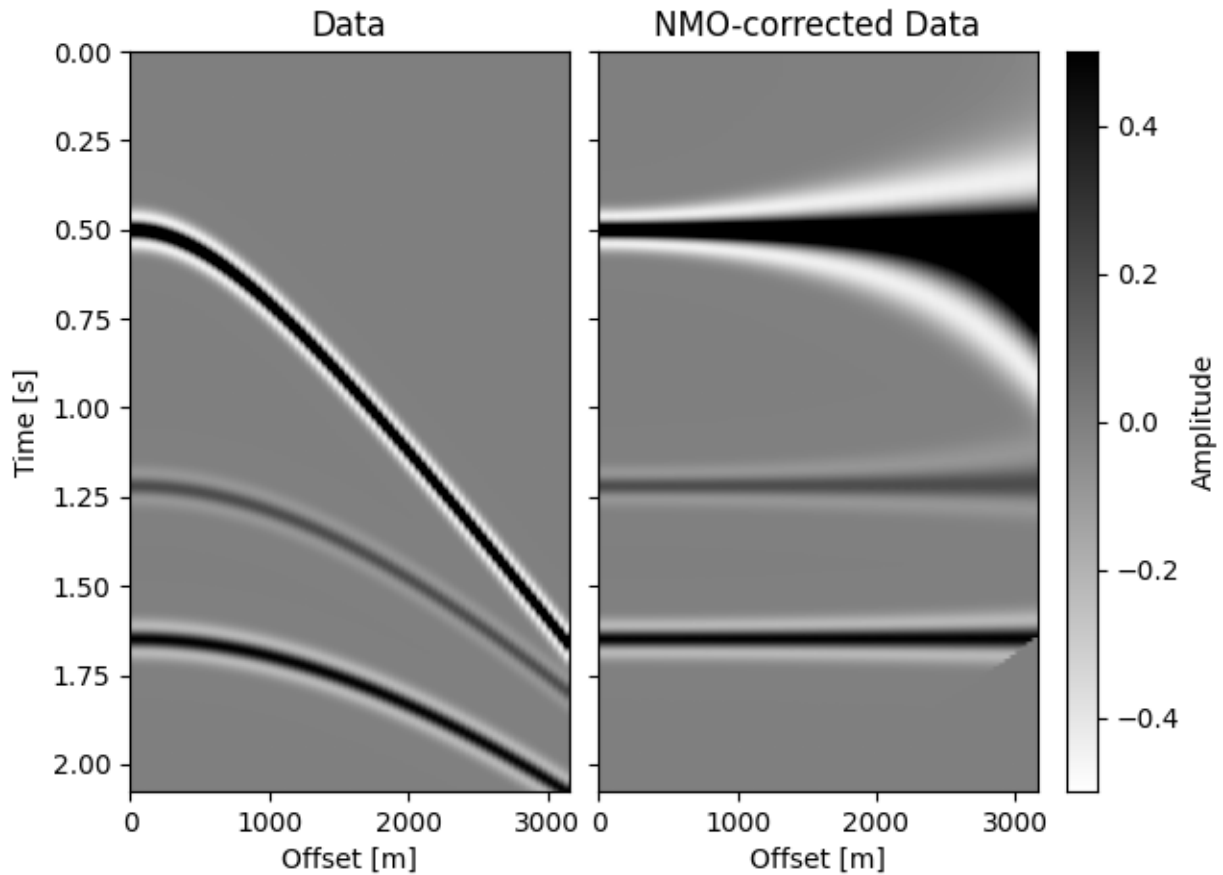
```
Ran in 321 μs
```

```

# Plot Data and NMO-corrected data
fig = plt.figure(figsize=(6.5, 5))
grid = ImageGrid(
    fig,
    111,
    nrows_ncols=(1, 2),
    axes_pad=0.15,
    cbar_location="right",
    cbar_mode="single",
    cbar_size="7%",
    cbar_pad=0.15,
    aspect=False,
    share_all=True,
)
im = grid[0].imshow(data.T, **opts)
grid[0].set(title="Data", xlabel="Offset [m]", ylabel="Time [s]")
grid[0].cax.colorbar(im)
grid[0].cax.set_ylabel("Amplitude")

grid[1].imshow(dnmo.T, **opts)
grid[1].set(title="NMO-corrected Data", xlabel="Offset [m]")
plt.show()

```



Now that we know how to compute the forward, we'll compute the adjoint pass. With these two functions, we can create a `LinearOperator` and ensure that it passes the dot-test.

```
@jit(nopython=True, fastmath=True, nogil=True, parallel=True)
def nmo_adjoint(dnmo, taxis, haxis, vels_rms):
    dt = taxis[1] - taxis[0]
    ot = taxis[0]
    nt = len(taxis)
    nh = len(haxis)

    data = np.zeros_like(dnmo)

    # Parallel outer loop on slow axis; use range if Numba is not installed
    for ih in prange(nh):
        h = haxis[ih]
        for it0, (t0, vrms) in enumerate(zip(taxis, vels_rms)):
            # Compute NMO traveltimes
            tx = np.sqrt(t0 ** 2 + (h / vrms) ** 2)
            it_frac = (tx - ot) / dt # Fractional index
            it_floor = floor(it_frac)
            it_ceil = it_floor + 1
            w = it_frac - it_floor
            if 0 <= it_floor and it_ceil < nt:
                # Linear interpolation
```

(continues on next page)

(continued from previous page)

```

        # In the adjoint, we must spread the same it0 to both it_floor and
        # it_ceil, since in the forward pass, both of these samples were
        # pushed onto it0
        data[ih, it_floor] += (1 - w) * dnmo[ih, it0]
        data[ih, it_ceil] += w * dnmo[ih, it0]

    return data

```

Finally, we can create our linear operator. To exemplify the class-based interface we will subclass `pylops.LinearOperator` and implement the required methods: `_matvec` which will compute the forward and `_rmatvec` which will compute the adjoint.

```

class NMO(LinearOperator):
    def __init__(self, taxis, haxis, vels_rms, dtype=None):
        self.taxis = taxis
        self.haxis = haxis
        self.vels_rms = vels_rms
        self.dims = (len(haxis), len(taxis))

        # Required LinearOperator attributes
        self.explicit = False
        self.shape = (np.prod(self.dims),) * 2
        argdtypes = np.result_type(taxis.dtype, haxis.dtype, vels_rms.dtype)
        self.dtype = argdtypes if dtype is None else dtype
        self.clinear = False

    def _matvec(self, x):
        y = nmo_forward(x.reshape(self.dims), self.taxis, self.haxis, self.vels_rms)
        y = y.ravel()
        return y

    def _rmatvec(self, y):
        x = nmo_adjoint(y.reshape(self.dims), self.taxis, self.haxis, self.vels_rms)
        x = x.ravel()
        return x

```

With our new NMO linear operator, we can instantiate it with our current example and ensure that it passes the dot test which proves that our forward and adjoint transforms truly are adjoints of each other.

```

NMOOp = NMO(t, x, vel_t)
dottest(NMOOp, *NMOOp.shape, tol=1e-4)

```

Out:

```
True
```

NMO using `pylops.Spread`

We learned how to implement an NMO correction and its adjoint from scratch. The adjoint has an interesting pattern, where energy taken from one domain is “spread” along a previously-defined parametric curve (the NMO hyperbola in this case). This pattern is very common in many algorithms, including Radon transform, Kirchhoff migration (also known as Total Focusing Method in ultrasonics) and many others.

For these classes of operators, PyLops offers a `pylops.Spread` constructor, which we will leverage to implement a version of the NMO correction. The `pylops.Spread` operator will take a value in the “input” domain, and spread

it along a parametric curve, defined in the “output” domain. In our case, the spreading operation is the *adjoint* of the NMO, so our “input” domain is the NMO domain, and the “output” domain is the original data domain.

In order to use `pylops.Spread`, we need to define the parametric curves. This can be done through the use of a table with shape (n_{x_i}, n_t, n_{x_o}) , where n_{x_i} and n_t represent the 2d dimensions of the “input” domain (NMO domain) and n_{x_o} and n_t the 2d dimensions of the “output” domain. In our NMO case, $n_{x_i} = n_{x_o} = n_h$ represents the number of offsets. Following the documentation of `pylops.Spread`, the table will be used in the following manner:

```
d_out[ix_o, table[ix_i, it, ix_o]] += d_in[ix_i, it]
```

In our case, `ix_o = ix_i = ih`, and comparing with our NMO adjoint, `it` refers to t_0 while `table[ix, it, ix]` should then provide the appropriate index for $t(x)$. In our implementation we will also be constructing a second table containing the weights to be used for linear interpolation.

```
def create_tables(taxis, haxis, vels_rms):
    dt = taxis[1] - taxis[0]
    ot = taxis[0]
    nt = len(taxis)
    nh = len(haxis)

    # NaN values will be not be spread.
    # Using np.zeros has the same result but much slower.
    table = np.full((nh, nt, nh), fill_value=np.nan)
    dtable = np.full((nh, nt, nh), fill_value=np.nan)

    for ih, h in enumerate(haxis):
        for it0, (t0, vrms) in enumerate(zip(taxis, vels_rms)):
            # Compute NMO traveltime
            tx = np.sqrt(t0 ** 2 + (h / vrms) ** 2)
            it_frac = (tx - ot) / dt
            it_floor = floor(it_frac)
            w = it_frac - it_floor
            # Both it_floor and it_floor + 1 must be valid indices for taxis
            # when using two tables (interpolation).
            if 0 <= it_floor and it_floor + 1 < nt:
                table[ih, it0, ih] = it_floor
                dtable[ih, it0, ih] = w
    return table, dtable

nmo_table, nmo_dtable = create_tables(t, x, vel_t)
```

```
SpreadNMO = Spread(
    data.shape, # "Input" shape: NMO-ed data shape
    data.shape, # "Output" shape: original data shape
    table=nmo_table, # Table of time indices
    dtable=nmo_dtable, # Table of weights for linear interpolation
    engine="numba", # numba or numpy
).H # To perform NMO *correction*, we need the adjoint
dottest(SpreadNMO, *SpreadNMO.shape, tol=1e-4)
```

Out:

```
True
```

We see it passes the dot test, but are the results right? Let’s find out.

```

dnmo_spr = (SpreadNMO @ data.ravel()).reshape(data.shape)

start = time()
SpreadNMO @ data.ravel()
end = time()

print(f"Ran in {1e6*(end-start):.0f}  $\mu$ s")

```

Out:

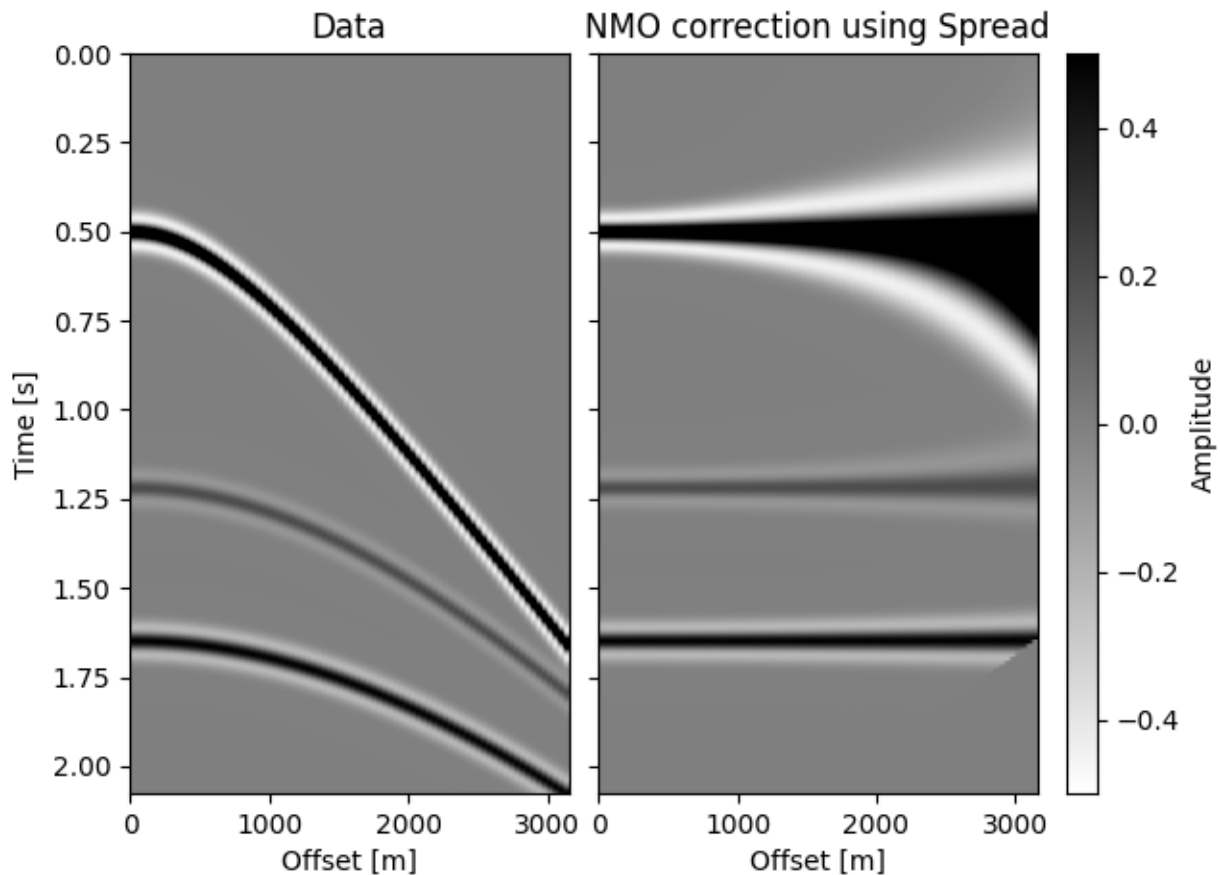
```
Ran in 14122  $\mu$ s
```

```

# Plot Data and NMO-corrected data
fig = plt.figure(figsize=(6.5, 5))
grid = ImageGrid(
    fig,
    111,
    nrows_ncols=(1, 2),
    axes_pad=0.15,
    cbar_location="right",
    cbar_mode="single",
    cbar_size="7%",
    cbar_pad=0.15,
    aspect=False,
    share_all=True,
)
im = grid[0].imshow(data.T, **opts)
grid[0].set(title="Data", xlabel="Offset [m]", ylabel="Time [s]")
grid[0].cax.colorbar(im)
grid[0].cax.set_ylabel("Amplitude")

grid[1].imshow(dnmo_spr.T, **opts)
grid[1].set(title="NMO correction using Spread", xlabel="Offset [m]")
plt.show()

```



Not as blazing fast as our original implementation, but pretty good (try the “numpy” backend for comparison!). In fact, using the `Spread` operator for NMO will always have a speed disadvantage. While iterating over the table, it must loop over the offsets twice: one for the “input” offsets and one for the “output” offsets. We know they are the same for NMO, but since `Spread` is a generic operator, it does not know that. So right off the bat we can expect an 80x slowdown ($nh = 80$). We diminished this cost to about 30x by setting values where $ix_i \neq ix_o$ to NaN, but nothing beats the custom implementation. Despite this, we can still produce the same result to numerical accuracy:

```
np.allclose(dnmo, dnmo_spr)
```

Out:

```
True
```

Total running time of the script: (0 minutes 4.328 seconds)

3.6.24 Operators concatenation

This example shows how to use ‘stacking’ operators such as `pylops.VStack`, `pylops.HStack`, `pylops.Block`, `pylops.BlockDiag`, and `pylops.Kronecker`.

These operators allow for different combinations of multiple linear operators in a single operator. Such functionalities are used within PyLops as the basis for the creation of complex operators as well as in the definition of various types of optimization problems with regularization or preconditioning.

Some of these operators naturally lend to embarrassingly parallel computations. Within PyLops we leverage the multiprocessing module to run multiple processes at the same time evaluating a subset of the operators involved in one of the stacking operations.

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

import pylops

plt.close("all")
```

Let's start by defining two second derivatives `pylops.SecondDerivative` that we will be using in this example.

```
D2hop = pylops.SecondDerivative(11 * 21, dims=[11, 21], dir=1, dtype="float32")
D2vop = pylops.SecondDerivative(11 * 21, dims=[11, 21], dir=0, dtype="float32")
```

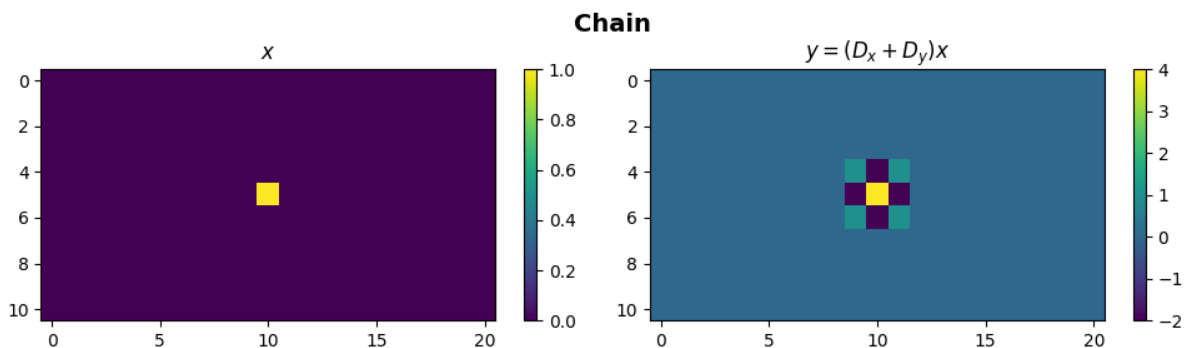
Chaining of operators represents the simplest concatenation that can be performed between two or more linear operators. This can be easily achieved using the `*` symbol

$$\mathbf{D}_{\text{cat}} = \mathbf{D}_v \mathbf{D}_h$$

```
Nv, Nh = 11, 21
X = np.zeros((Nv, Nh))
X[int(Nv / 2), int(Nh / 2)] = 1

D2op = D2vop * D2hop
Y = np.reshape(D2op * X.ravel(), (Nv, Nh))

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle("Chain", fontsize=14, fontweight="bold", y=0.95)
im = axs[0].imshow(X, interpolation="nearest")
axs[0].axis("tight")
axs[0].set_title(r"$x$")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(Y, interpolation="nearest")
axs[1].axis("tight")
axs[1].set_title(r"$y = (D_x + D_y)x$")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)
```



We now want to *vertically stack* three operators

$$\mathbf{D}_{Vstack} = \begin{bmatrix} \mathbf{D}_v \\ \mathbf{D}_h \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} \mathbf{D}_v \mathbf{x} \\ \mathbf{D}_h \mathbf{x} \end{bmatrix}$$

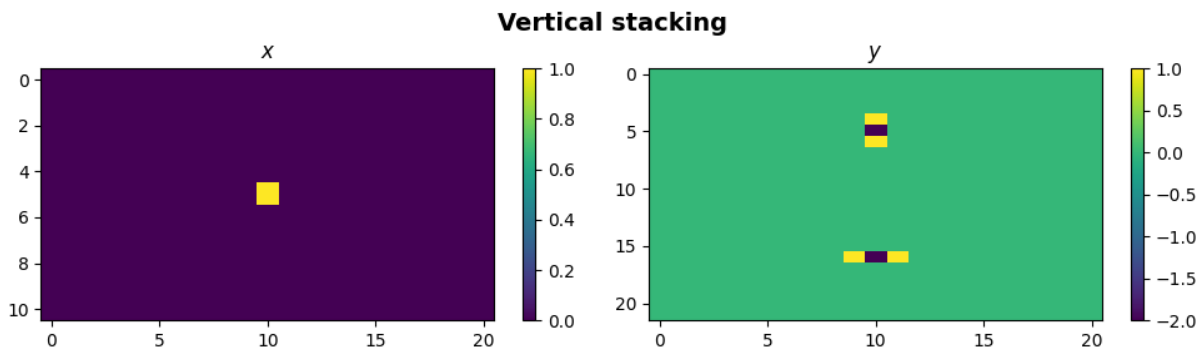
```

Nv, Nh = 11, 21
X = np.zeros((Nv, Nh))
X[int(Nv / 2), int(Nh / 2)] = 1
Dstack = pylops.VStack([D2vop, D2hop])

Y = np.reshape(Dstack * X.ravel(), (Nv * 2, Nh))

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle("Vertical stacking", fontsize=14, fontweight="bold", y=0.95)
im = axs[0].imshow(X, interpolation="nearest")
axs[0].axis("tight")
axs[0].set_title(r"$x$")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(Y, interpolation="nearest")
axs[1].axis("tight")
axs[1].set_title(r"$y$")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)

```



Similarly we can now *horizontally stack* three operators

$$\mathbf{D}_{Hstack} = [\mathbf{D}_v \quad 0.5 * \mathbf{D}_v \quad -1 * \mathbf{D}_h], \quad \mathbf{y} = \mathbf{D}_v \mathbf{x}_1 + 0.5 * \mathbf{D}_v \mathbf{x}_2 - \mathbf{D}_h \mathbf{x}_3$$

```

Nv, Nh = 11, 21
X = np.zeros((Nv * 3, Nh))
X[int(Nv / 2), int(Nh / 2)] = 1
X[int(Nv / 2) + Nv, int(Nh / 2)] = 1
X[int(Nv / 2) + 2 * Nv, int(Nh / 2)] = 1

Hstackop = pylops.HStack([D2vop, 0.5 * D2vop, -1 * D2hop])
Y = np.reshape(Hstackop * X.ravel(), (Nv, Nh))

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle("Horizontal stacking", fontsize=14, fontweight="bold", y=0.95)
im = axs[0].imshow(X, interpolation="nearest")

```

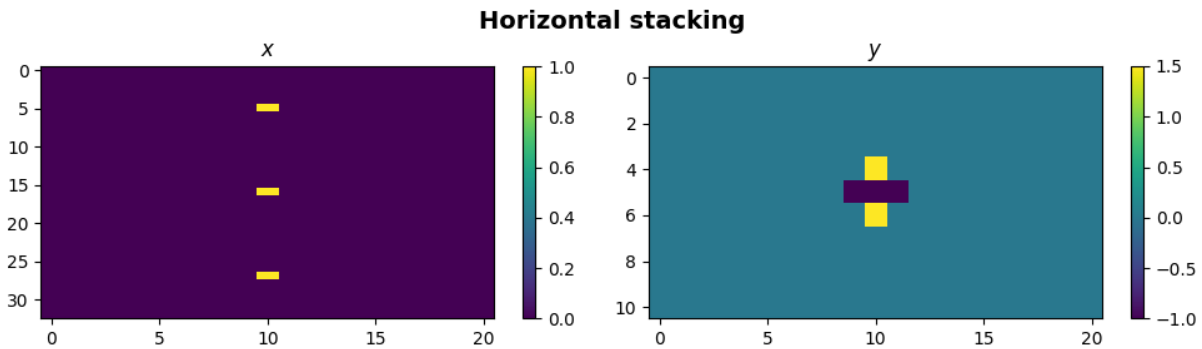
(continues on next page)

(continued from previous page)

```

axs[0].axis("tight")
axs[0].set_title(r"$x$")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(Y, interpolation="nearest")
axs[1].axis("tight")
axs[1].set_title(r"$y$")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)

```



We can even stack them both *horizontally* and *vertically* such that we create a *block* operator

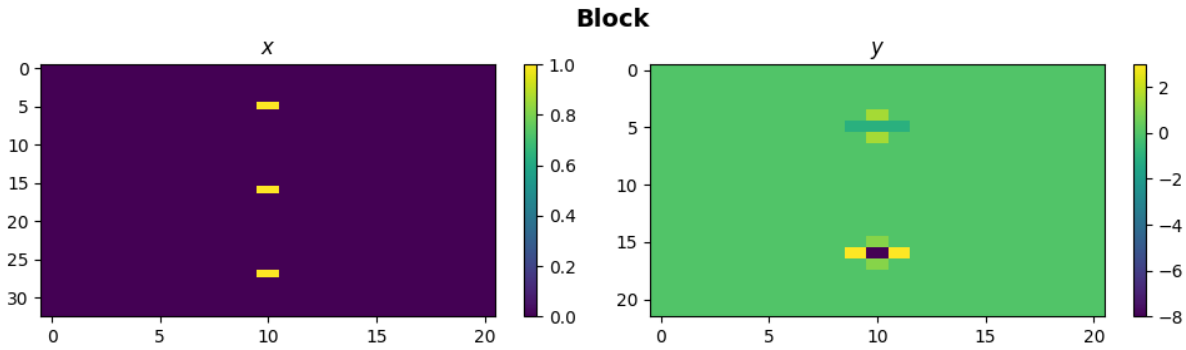
$$\mathbf{D}_{\text{Block}} = \begin{bmatrix} \mathbf{D}_v & 0.5 * \mathbf{D}_v & -1 * \mathbf{D}_h \\ \mathbf{D}_h & 2 * \mathbf{D}_h & \mathbf{D}_v \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} \mathbf{D}_v \mathbf{x}_1 + 0.5 * \mathbf{D}_v \mathbf{x}_2 - \mathbf{D}_h \mathbf{x}_3 \\ \mathbf{D}_h \mathbf{x}_1 + 2 * \mathbf{D}_h \mathbf{x}_2 + \mathbf{D}_v \mathbf{x}_3 \end{bmatrix}$$

```

Bop = pylops.Block([D2vop, 0.5 * D2vop, -1 * D2hop], [D2hop, 2 * D2hop, D2vop])
Y = np.reshape(Bop * X.ravel(), (2 * Nv, Nh))

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle("Block", fontsize=14, fontweight="bold", y=0.95)
im = axs[0].imshow(X, interpolation="nearest")
axs[0].axis("tight")
axs[0].set_title(r"$x$")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(Y, interpolation="nearest")
axs[1].axis("tight")
axs[1].set_title(r"$y$")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)

```

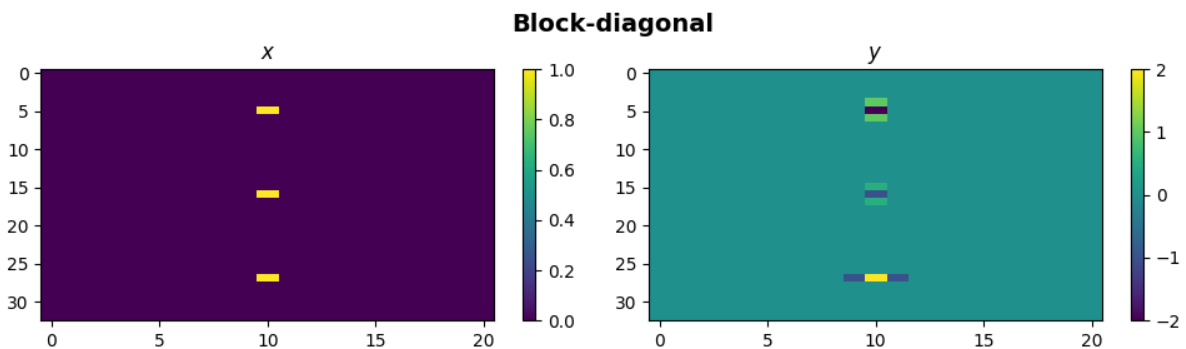


Finally we can use the *block-diagonal operator* to apply three operators to three different subset of the model and data

$$\mathbf{D}_{BDiag} = \begin{bmatrix} \mathbf{D}_v & 0 & 0 \\ 0 & 0.5 * \mathbf{D}_v & 0 \\ 0 & 0 & -\mathbf{D}_h \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} \mathbf{D}_v \mathbf{x}_1 \\ 0.5 * \mathbf{D}_v \mathbf{x}_2 \\ -\mathbf{D}_h \mathbf{x}_3 \end{bmatrix}$$

```
BD = pylops.BlockDiag([D2vop, 0.5 * D2vop, -1 * D2hop])
Y = np.reshape(BD * X.ravel(), (3 * Nv, Nh))

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle("Block-diagonal", fontsize=14, fontweight="bold", y=0.95)
im = axs[0].imshow(X, interpolation="nearest")
axs[0].axis("tight")
axs[0].set_title(r"$x$")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(Y, interpolation="nearest")
axs[1].axis("tight")
axs[1].set_title(r"$y$")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)
```



If we consider now the case of having a large number of operators inside a blockdiagonal structure, it may be convenient to span multiple processes handling subset of operators at the same time. This can be easily achieved by simply defining the number of processes we want to use via `nproc`.

```
X = np.zeros((Nv * 10, Nh))
for iv in range(10):
    X[int(Nv / 2) + iv * Nv, int(Nh / 2)] = 1
```

(continues on next page)

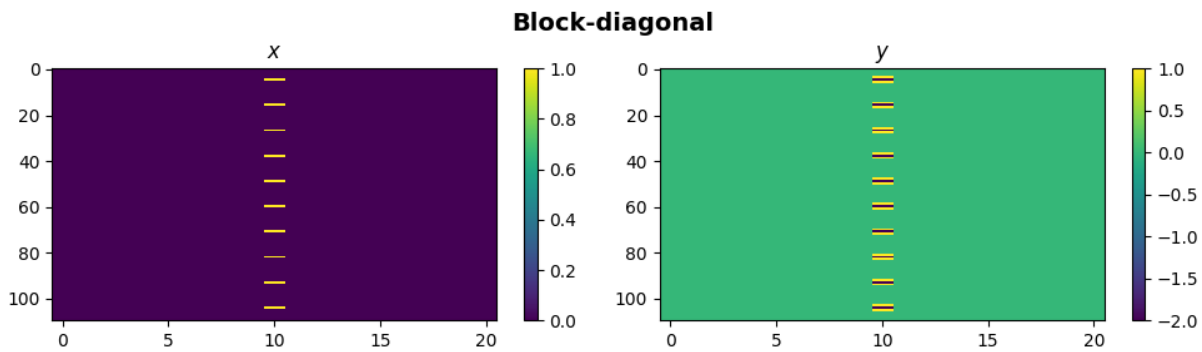
(continued from previous page)

```

BD = pylops.BlockDiag([D2vop] * 10, nproc=2)
print("BD Operator multiprocessing pool", BD.pool)
Y = np.reshape(BD * X.ravel(), (10 * Nv, Nh))
BD.pool.close()

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle("Block-diagonal", fontsize=14, fontweight="bold", y=0.95)
im = axs[0].imshow(X, interpolation="nearest")
axs[0].axis("tight")
axs[0].set_title(r"$x$")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(Y, interpolation="nearest")
axs[1].axis("tight")
axs[1].set_title(r"$y$")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)

```



Out:

```
BD Operator multiprocessing pool <multiprocessing.pool.Pool object at 0x7f6ae9861ac8>
```

Finally we use the *Kronecker operator* and replicate this example on [wiki](#).

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 0 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 0 & 5 & 0 & 10 \\ 6 & 7 & 12 & 14 \\ 0 & 15 & 0 & 20 \\ 18 & 21 & 24 & 28 \end{bmatrix}$$

```

A = np.array([[1, 2], [3, 4]])
B = np.array([[0, 5], [6, 7]])
AB = np.kron(A, B)

n1, m1 = A.shape
n2, m2 = B.shape

Aop = pylops.MatrixMult(A)
Bop = pylops.MatrixMult(B)

ABop = pylops.Kronecker(Aop, Bop)

```

(continues on next page)

(continued from previous page)

```

x = np.ones(m1 * m2)

y = AB.dot(x)
yop = ABop * x
xinv = ABop / yop

print("AB = \n", AB)

print("x = ", x)
print("y = ", y)
print("yop = ", yop)
print("xinv = ", x)

```

Out:

```

AB =
[[ 0  5  0 10]
 [ 6  7 12 14]
 [ 0 15  0 20]
 [18 21 24 28]]
x = [1. 1. 1. 1.]
y = [15. 39. 35. 91.]
yop = [15. 39. 35. 91.]
xinv = [1. 1. 1. 1.]

```

We can also use `pylops.Kronecker` to do something more interesting. Any operator can in fact be applied on a single direction of a multi-dimensional input array if combined with an `pylops.Identity` operator via Kronecker product. We apply here the `pylops.FirstDerivative` to the second dimension of the model.

Note that for those operators whose implementation allows their application to a single axis via the `dir` parameter, using the Kronecker product would lead to slower performance. Nevertheless, the Kronecker product allows any other operator to be applied to a single dimension.

```

Nv, Nh = 11, 21

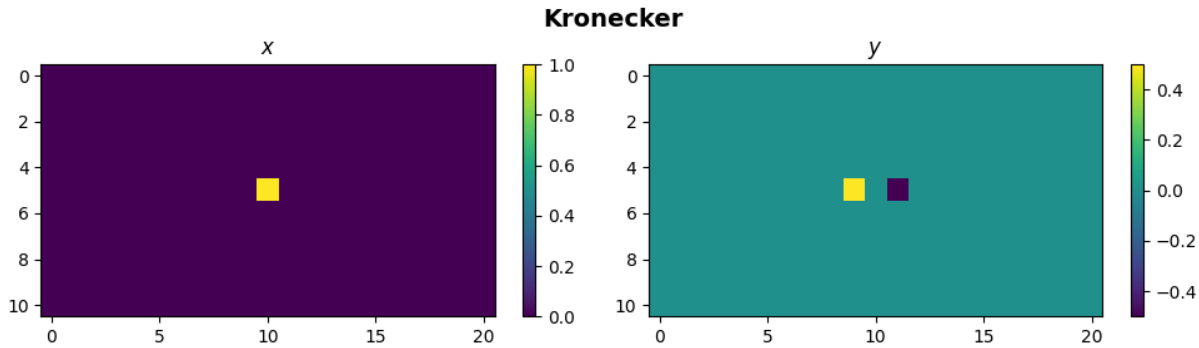
Iop = pylops.Identity(Nv, dtype="float32")
D2hop = pylops.FirstDerivative(Nh, dtype="float32")

X = np.zeros((Nv, Nh))
X[Nv // 2, Nh // 2] = 1
D2hop = pylops.Kronecker(Iop, D2hop)

Y = D2hop * X.ravel()
Y = Y.reshape(Nv, Nh)

fig, axs = plt.subplots(1, 2, figsize=(10, 3))
fig.suptitle("Kronecker", fontsize=14, fontweight="bold", y=0.95)
im = axs[0].imshow(X, interpolation="nearest")
axs[0].axis("tight")
axs[0].set_title(r"$x$")
plt.colorbar(im, ax=axs[0])
im = axs[1].imshow(Y, interpolation="nearest")
axs[1].axis("tight")
axs[1].set_title(r"$y$")
plt.colorbar(im, ax=axs[1])
plt.tight_layout()
plt.subplots_adjust(top=0.8)

```



Total running time of the script: (0 minutes 2.421 seconds)

3.6.25 Operators with Multiprocessing

This example shows how perform a scalability test for one of PyLops operators that uses multiprocessing to spawn multiple processes. Operators that support such feature are `pylops.basicoperators.VStack`, `pylops.basicoperators.HStack`, and `pylops.basicoperators.BlockDiagonal`, and `pylops.basicoperators.Block`.

In this example we will consider the `BlockDiagonal` operator which contains `pylops.basicoperators.MatrixMult` operators along its main diagonal.

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

import pylops

plt.close("all")
```

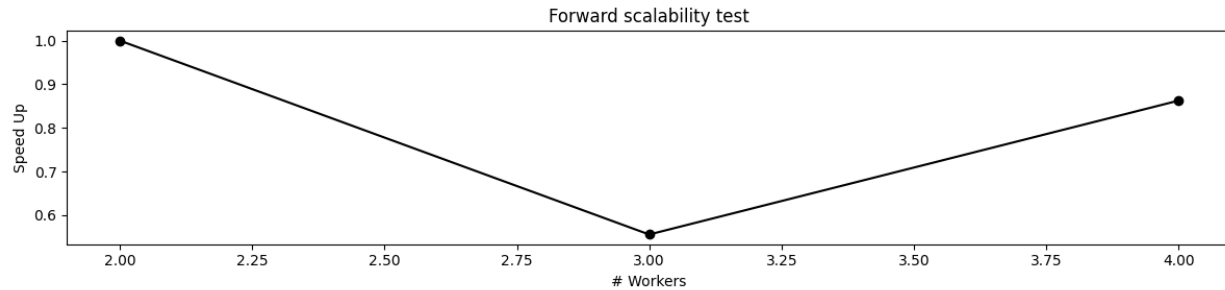
Let's start by creating `N` `MatrixMult` operators and the `BlockDiag` operator

```
N = 100
Nops = 32
Ops = [pylops.MatrixMult(np.random.normal(0.0, 1.0, (N, N))) for _ in range(Nops)]

Op = pylops.BlockDiag(Ops, nproc=1)
```

We can now perform a scalability test on the forward operation

```
workers = [2, 3, 4]
compute_times, speedup = pylops.utils.multiproc.scalability_test(
    Op, np.ones(Op.shape[1]), workers=workers, forward=True
)
plt.figure(figsize=(12, 3))
plt.plot(workers, speedup, "ko-")
plt.xlabel("# Workers")
plt.ylabel("Speed Up")
plt.title("Forward scalability test")
plt.tight_layout()
```

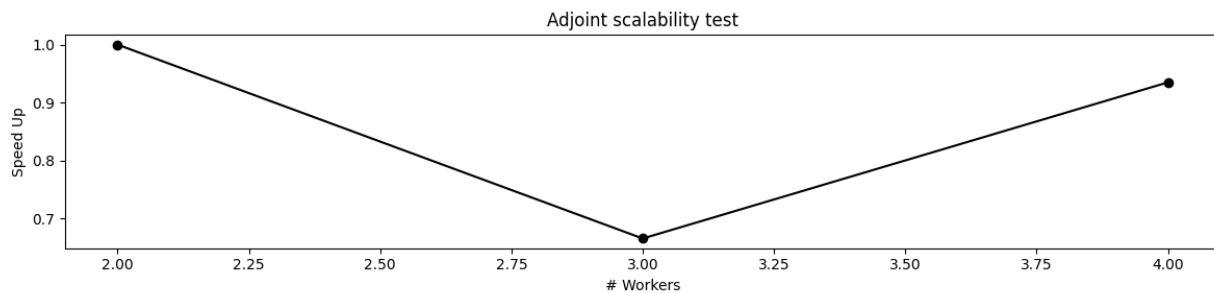


Out:

```
Working with 2 workers...
Working with 3 workers...
Working with 4 workers...
```

And likewise on the adjoint operation

```
compute_times, speedup = pyllops.utils.multiproc.scalability_test(
    Op, np.ones(Op.shape[0]), workers=workers, forward=False
)
plt.figure(figsize=(12, 3))
plt.plot(workers, speedup, "ko-")
plt.xlabel("# Workers")
plt.ylabel("Speed Up")
plt.title("Adjoint scalability test")
plt.tight_layout()
```



Out:

```
Working with 2 workers...
Working with 3 workers...
Working with 4 workers...
```

Note that we have not tested here the case with 1 worker. In this specific case, since the computations are very small, the overhead of spawning processes is actually dominating the time of computations and so computing the forward and adjoint operations with a single worker is more efficient. We hope that this example can serve as a basis to inspect the scalability of multiprocessing-enabled operators and choose the best number of processes.

Total running time of the script: (0 minutes 0.917 seconds)

3.6.26 Padding

This example shows how to use the `pyllops.Pad` operator to zero-pad a model


```
import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt
import numpy as np

import pylops

plt.close("all")
```

Let's define a pad operator Pop for one dimensional data

```
dims = 10
pad = (2, 3)

Pop = pylops.Pad(dims, pad)

x = np.arange(dims) + 1.0
y = Pop * x
xadj = Pop.H * y

print("x = %s " % x)
print("P*x = %s " % y)
print("P'*y = %s " % xadj)
```

Out:

```
x = [ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
P*x = [ 0.  0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.  0.  0.  0.]
P'*y = [ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

We move now to a multi-dimensional case. We pad the input model with different extents along both dimensions

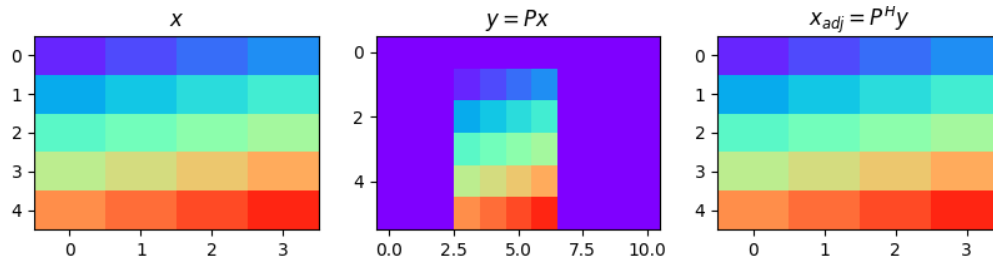
```
dims = (5, 4)
pad = ((1, 0), (3, 4))

Pop = pylops.Pad(dims, pad)

x = (np.arange(np.prod(np.array(dims))) + 1.0).reshape(dims)
y = Pop * x.ravel()
xadj = Pop.H * y

y = y.reshape(Pop.dimsd)
xadj = xadj.reshape(dims)

fig, axs = plt.subplots(1, 3, figsize=(10, 2))
fig.suptitle("Pad for 2d data", fontsize=14, fontweight="bold", y=1.15)
axs[0].imshow(x, cmap="rainbow", vmin=0, vmax=np.prod(np.array(dims)) + 1)
axs[0].set_title(r"$x$")
axs[0].axis("tight")
axs[1].imshow(y, cmap="rainbow", vmin=0, vmax=np.prod(np.array(dims)) + 1)
axs[1].set_title(r"$y = P x$")
axs[1].axis("tight")
axs[2].imshow(xadj, cmap="rainbow", vmin=0, vmax=np.prod(np.array(dims)) + 1)
axs[2].set_title(r"$x_{adj} = P^H y$")
axs[2].axis("tight")
```



Out:

```
(-0.5, 3.5, 4.5, -0.5)
```

Total running time of the script: (0 minutes 0.210 seconds)

3.6.27 PhaseShift operator

This example shows how to use the `pylops.waveeqprocessing.PhaseShift` operator to perform frequency-wavenumber shift of an input multi-dimensional signal. Such a procedure is applied in a variety of disciplines including geophysics, medical imaging and non-destructive testing.

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

import pylops

plt.close("all")
```

Let's first create a synthetic dataset composed of a number of hyperbolas

```
par = {
    "ox": -300,
    "dx": 20,
    "nx": 31,
    "oy": -200,
    "dy": 20,
    "ny": 21,
    "ot": 0,
    "dt": 0.004,
    "nt": 201,
    "f0": 20,
    "nfmax": 210,
}

# Create axis
t, t2, x, y = pylops.utils.seismicevents.makeaxis(par)

# Create wavelet
wav = pylops.utils.wavelets.ricker(np.arange(41) * par["dt"], f0=par["f0"])[0]

vrms = [900, 1300, 1800]
t0 = [0.2, 0.3, 0.6]
amp = [1.0, 0.6, -2.0]

_, m = pylops.utils.seismicevents.hyperbolic2d(x, t, t0, vrms, amp, wav)
```

We can now apply a taper at the edges and also pad the input to avoid artifacts during the phase shift

```
pad = 11
taper = pylops.utils.tapers.taper2d(par["nt"], par["nx"], 5)
mpad = np.pad(m * taper, ((pad, pad), (0, 0)), mode="constant")
```

We perform now forward propagation in a constant velocity $v = 2000$ for a depth of $z_{prop} = 100m$. We should expect the hyperbolas to move forward in time and become flatter.

```
vel = 1500.0
zprop = 100
freq = np.fft.rfftfreq(par["nt"], par["dt"])
kx = np.fft.fftshift(np.fft.fftfreq(par["nx"] + 2 * pad, par["dx"]))
Pop = pylops.waveeqprocessing.PhaseShift(vel, zprop, par["nt"], freq, kx)

mdown = Pop * mpad.T.ravel()
```

We now take the forward propagated wavefield and apply backward propagation, which is in this case simply the adjoint of our operator. We should expect the hyperbolas to move backward in time and show the same traveltimes as the original dataset. Of course, as we are only performing the adjoint operation we should expect some small differences between this wavefield and the input dataset.

```
mup = Pop.H * mdown.ravel()

mdown = np.real(mdown.reshape(par["nt"], par["nx"] + 2 * pad)[: , pad:-pad]))
mup = np.real(mup.reshape(par["nt"], par["nx"] + 2 * pad)[: , pad:-pad]))

fig, axs = plt.subplots(1, 3, figsize=(10, 6), sharey=True)
fig.suptitle("2D Phase shift", fontsize=12, fontweight="bold")
axs[0].imshow(
    m.T,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[0].set_xlabel(r"$x(m)$")
axs[0].set_ylabel(r"$t(s)$")
axs[0].set_title("Original data")
axs[1].imshow(
    mdown,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[1].set_xlabel(r"$x(m)$")
axs[1].set_title("Forward propagation")
axs[2].imshow(
    mup,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
```

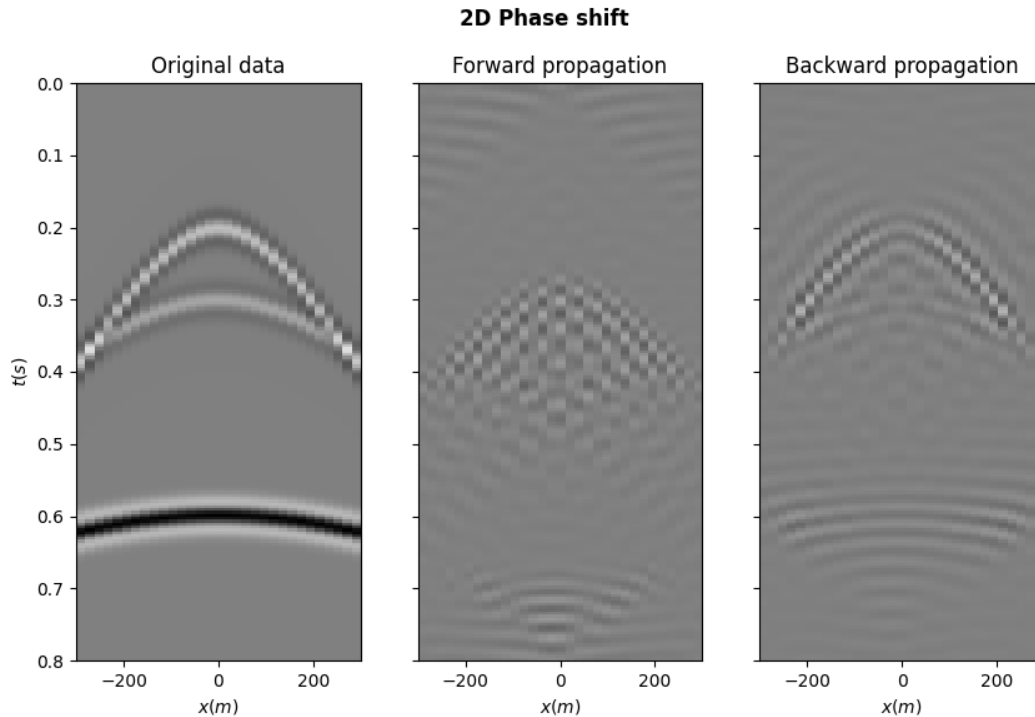
(continues on next page)

(continued from previous page)

```

    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[2].set_xlabel(r"$x(m)$")
axs[2].set_title("Backward propagation")

```



Out:

```
Text(0.5, 1.0, 'Backward propagation')
```

Finally we perform the same for a 3-dimensional signal

```

_, m = pylops.utils.seismicevents.hyperbolic3d(x, y, t, t0, vrms, vrms, amp, wav)

pad = 11
taper = pylops.utils.tapers.taper3d(par["nt"], (par["ny"], par["nx"]), (3, 3))
mpad = np.pad(m * taper, ((pad, pad), (pad, pad), (0, 0)), mode="constant")

kx = np.fft.fftshift(np.fft.fftfreq(par["nx"] + 2 * pad, par["dx"]))
ky = np.fft.fftshift(np.fft.fftfreq(par["ny"] + 2 * pad, par["dy"]))
Pop = pylops.waveeqprocessing.PhaseShift(vel, zprop, par["nt"], freq, kx, ky)

mdown = Pop * mpad.transpose(2, 1, 0).ravel()

mup = Pop.H * mdown.ravel()

mdown = np.real(
    mdown.reshape(par["nt"], par["nx"] + 2 * pad, par["ny"] + 2 * pad)[
        :, pad:-pad, pad:-pad
    ]
)

```

(continues on next page)

(continued from previous page)

```

mup = np.real(
    mup.reshape(par["nt"], par["nx"] + 2 * pad, par["ny"] + 2 * pad)[
        :, pad:-pad, pad:-pad
    ]
)

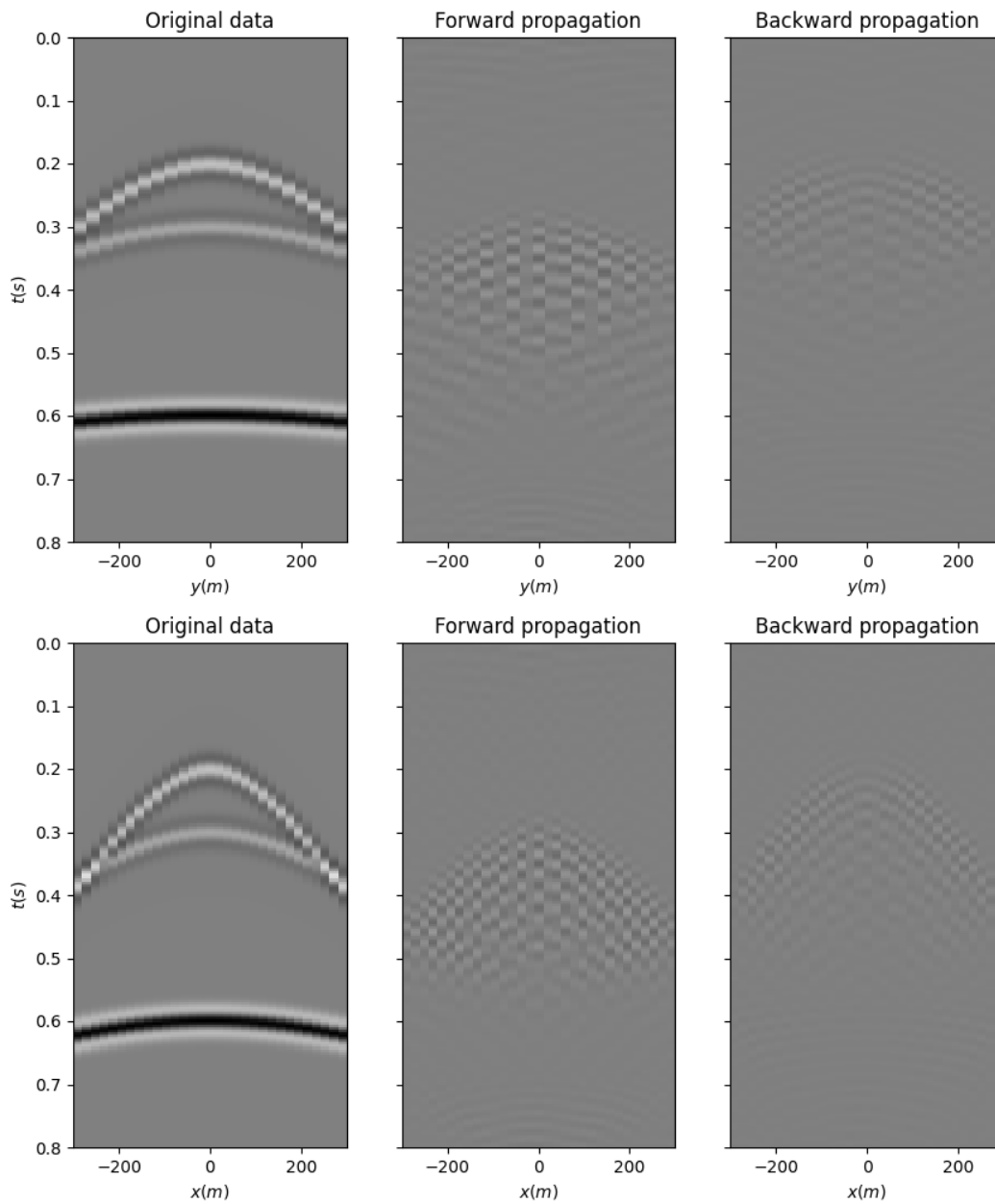
fig, axs = plt.subplots(2, 3, figsize=(10, 12), sharey=True)
fig.suptitle("3D Phase shift", fontsize=12, fontweight="bold")
axs[0][0].imshow(
    m[:, par["nx"] // 2].T,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[0][0].set_xlabel(r"$y(m)$")
axs[0][0].set_ylabel(r"$t(s)$")
axs[0][0].set_title("Original data")
axs[0][1].imshow(
    mdown[:, par["nx"] // 2],
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[0][1].set_xlabel(r"$y(m)$")
axs[0][1].set_title("Forward propagation")
axs[0][2].imshow(
    mup[:, par["nx"] // 2],
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[0][2].set_xlabel(r"$y(m)$")
axs[0][2].set_title("Backward propagation")
axs[1][0].imshow(
    m[par["ny"] // 2].T,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[1][0].set_xlabel(r"$x(m)$")
axs[1][0].set_ylabel(r"$t(s)$")
axs[1][0].set_title("Original data")
axs[1][1].imshow(
    mdown[:, :, par["ny"] // 2],
    aspect="auto",

```

(continues on next page)

(continued from previous page)

```
interpolation="nearest",
vmin=-2,
vmax=2,
cmap="gray",
extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[1][1].set_xlabel(r"$x(m)$")
axs[1][1].set_title("Forward propagation")
axs[1][2].imshow(
    mup[:, :, par["ny"] // 2],
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[1][2].set_xlabel(r"$x(m)$")
axs[1][2].set_title("Backward propagation")
```

3D Phase shift

Out:

```
Text(0.5, 1.0, 'Backward propagation')
```

Total running time of the script: (0 minutes 0.928 seconds)

3.6.28 Polynomial Regression

This example shows how to use the `pylops.Regression` operator to perform *Polynomial regression analysis*.

In short, polynomial regression is the problem of finding the best fitting coefficients for the following equation:

$$y_i = \sum_{n=0}^{\text{order}} x_n t_i^n \quad \forall i = 0, 1, \dots, N-1$$

As we can express this problem in a matrix form:

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

our solution can be obtained by solving the following optimization problem:

$$J = \|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2$$

See documentation of `pylops.Regression` for more detailed definition of the forward problem.

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

import pylops

plt.close("all")
np.random.seed(10)
```

Define the input parameters: number of samples along the t-axis (N), order (order), regression coefficients (x), and standard deviation of noise to be added to data (sigma).

```
N = 30
order = 3
x = np.array([1.0, 0.05, 0.0, -0.01])
sigma = 1
```

Let's create the time axis and initialize the `pylops.Regression` operator

```
t = np.arange(N, dtype="float64") - N // 2
PProp = pylops.Regression(t, order=order, dtype="float64")
```

We can then apply the operator in forward mode to compute our data points along the x-axis (y). We will also generate some random gaussian noise and create a noisy version of the data (yn).

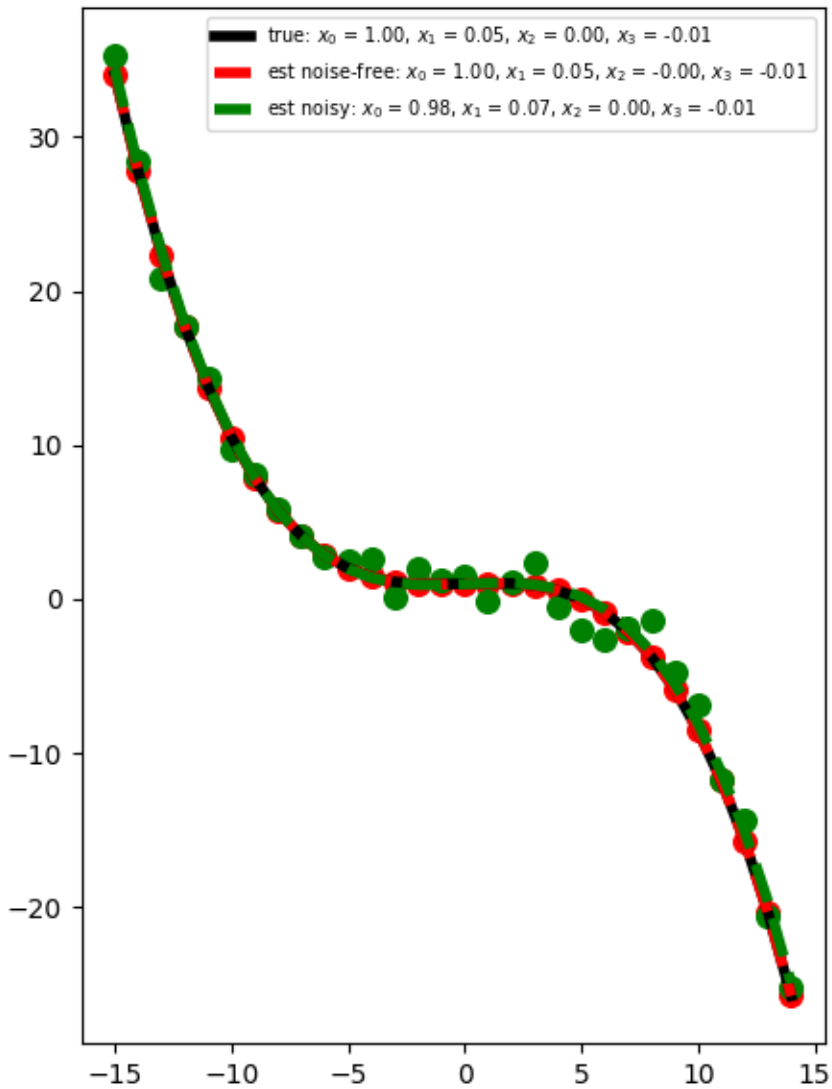
```
y = PProp * x
yn = y + np.random.normal(0, sigma, N)
```

We are now ready to solve our problem. As we are using an operator from the `pylops.LinearOperator` family, we can simply use `/`, which in this case will solve the system by means of an iterative solver (i.e., `scipy.sparse.linalg.lsqr`).


```
xest = PProp / y
xnest = PProp / yn
```

Let's plot the best fitting curve for the case of noise free and noisy data

```
plt.figure(figsize=(5, 7))
plt.plot(
    t,
    PProp * x,
    "k",
    lw=4,
    label=r"true: $x_0$ = %.2f, $x_1$ = %.2f, "
    r"$x_2$ = %.2f, $x_3$ = %.2f" % (x[0], x[1], x[2], x[3]),
)
plt.plot(
    t,
    PProp * xest,
    "--r",
    lw=4,
    label="est noise-free: $x_0$ = %.2f, $x_1$ = %.2f, "
    r"$x_2$ = %.2f, $x_3$ = %.2f" % (xest[0], xest[1], xest[2], xest[3]),
)
plt.plot(
    t,
    PProp * xnest,
    "--g",
    lw=4,
    label="est noisy: $x_0$ = %.2f, $x_1$ = %.2f, "
    r"$x_2$ = %.2f, $x_3$ = %.2f" % (xnest[0], xnest[1], xnest[2], xnest[3]),
)
plt.scatter(t, y, c="r", s=70)
plt.scatter(t, yn, c="g", s=70)
plt.legend(fontsize="x-small")
```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aece2cb70>
```

We consider now the case where some of the observations have large errors. Such elements are generally referred to as *outliers* and can affect the quality of the least-squares solution if not treated with care. In this example we will see how using a L1 solver such as `pylops.optimization.sparsity.IRLS` can dramatically improve the quality of the estimation of intercept and gradient.

```
# Add outliers
yn[1] += 40
```

(continues on next page)

(continued from previous page)

```

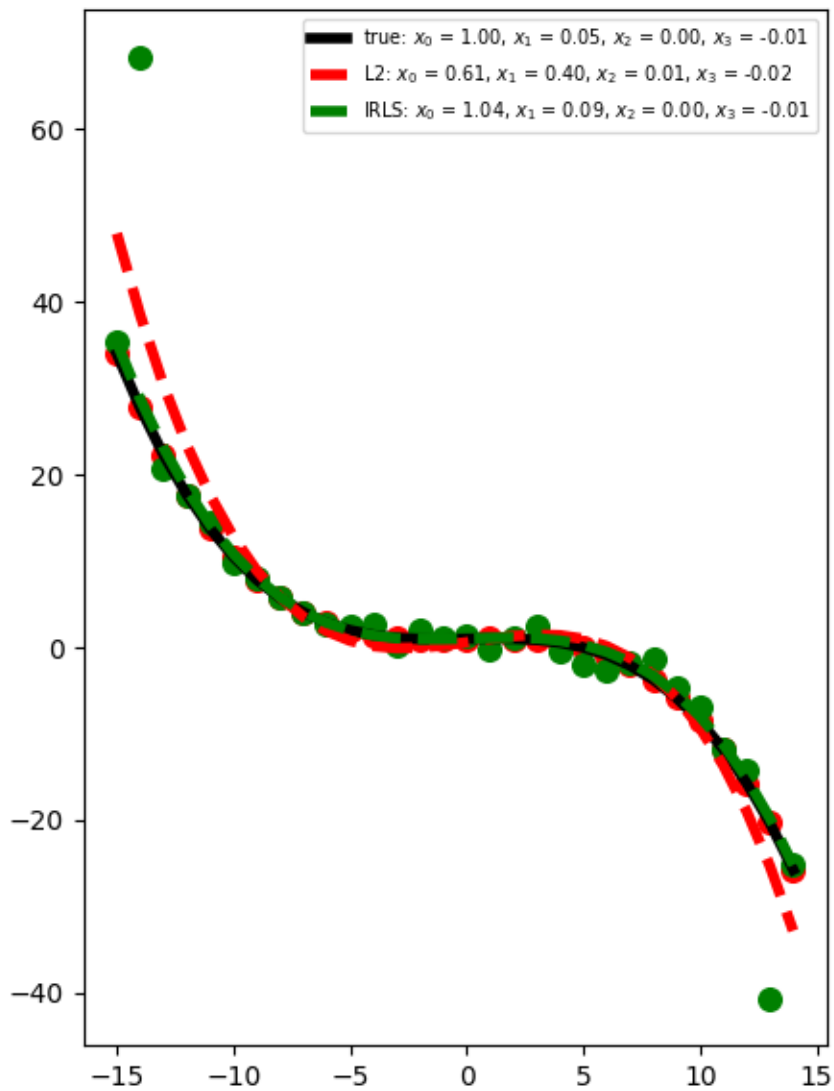
yn[N - 2] -= 20

# IRLS
nouter = 20
epsR = 1e-2
epsI = 0
tolIRLS = 1e-2

xnest = PProp / yn
xirls, nouter, xirls_hist, rw_hist = pylops.optimization.sparsity.IRLS(
    PProp,
    yn,
    nouter,
    threshR=False,
    epsR=epsR,
    epsI=epsI,
    tolIRLS=tolIRLS,
    returnhistory=True,
)
print("IRLS converged at %d iterations..." % nouter)

plt.figure(figsize=(5, 7))
plt.plot(
    t,
    PProp * x,
    "k",
    lw=4,
    label=r"true: $x_0$ = %.2f, $x_1$ = %.2f, "
    r"$x_2$ = %.2f, $x_3$ = %.2f" % (x[0], x[1], x[2], x[3]),
)
plt.plot(
    t,
    PProp * xnest,
    "--r",
    lw=4,
    label=r"L2: $x_0$ = %.2f, $x_1$ = %.2f, "
    r"$x_2$ = %.2f, $x_3$ = %.2f" % (xnest[0], xnest[1], xnest[2], xnest[3]),
)
plt.plot(
    t,
    PProp * xirls,
    "--g",
    lw=4,
    label=r"IRLS: $x_0$ = %.2f, $x_1$ = %.2f, "
    r"$x_2$ = %.2f, $x_3$ = %.2f" % (xirls[0], xirls[1], xirls[2], xirls[3]),
)
plt.scatter(t, y, c="r", s=70)
plt.scatter(t, yn, c="g", s=70)
plt.legend(fontsize="x-small")

```



Out:

```
IRLS converged at 4 iterations...  
<matplotlib.legend.Legend object at 0x7f6ae996f860>
```

Total running time of the script: (0 minutes 0.374 seconds)

3.6.29 Pre-stack modelling

This example shows how to create pre-stack angle gathers using the `pylops.avo.prestack.PrestackLinearModelling` operator.

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

import pylops
from pylops.utils.wavelets import ricker

plt.close("all")
np.random.seed(0)
```

Let's start by creating the input elastic property profiles and wavelet

```
nt0 = 501
dt0 = 0.004
ntheta = 21

t0 = np.arange(nt0) * dt0
thetamin, thetamax = 0, 40
theta = np.linspace(thetamin, thetamax, ntheta)

# Elastic property profiles
vp = (
    1200 + np.arange(nt0) + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 160,
    ↪nt0))
)
vs = 600 + vp / 2 + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 100, nt0))
rho = 1000 + vp + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 120, nt0))
vp[201:] += 500
vs[201:] += 200
rho[201:] += 100

# Wavelet
ntwav = 81
wav, twav, wavc = ricker(t0[: ntwav // 2 + 1], 5)

# vs/vp profile
vsvp = 0.5
vsvp_z = np.linspace(0.4, 0.6, nt0)

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

fig, axs = plt.subplots(1, 3, figsize=(13, 7), sharey=True)
axs[0].plot(vp, t0, "k")
axs[0].set_title("Vp")
axs[0].set_ylabel(r"$t(s)$")
axs[0].invert_yaxis()
axs[0].grid()
axs[1].plot(vs, t0, "k")
axs[1].set_title("Vs")
axs[1].invert_yaxis()
axs[1].grid()
axs[2].plot(rho, t0, "k")
```

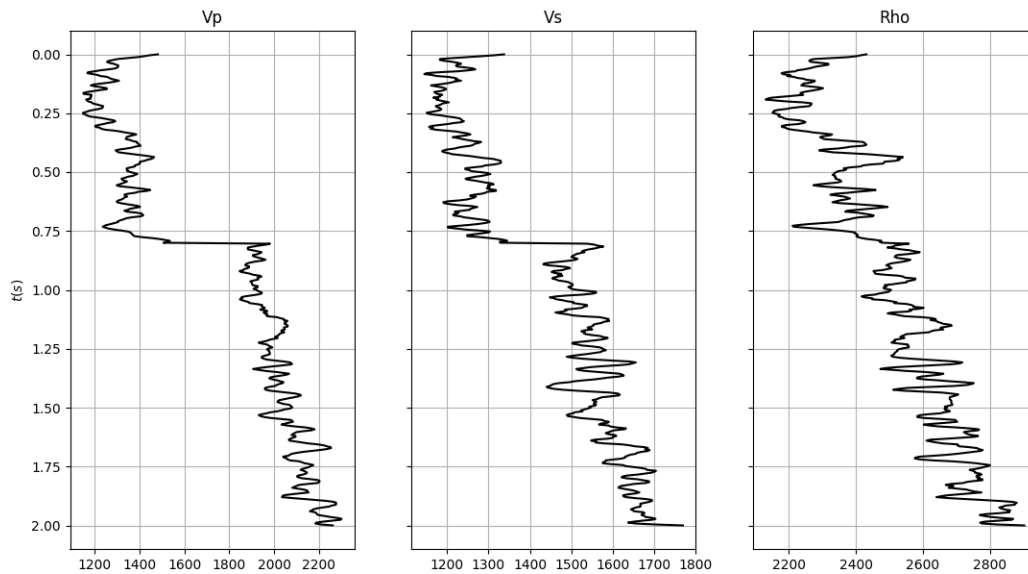
(continues on next page)

(continued from previous page)

```

axs[2].set_title("Rho")
axs[2].invert_yaxis()
axs[2].grid()

```



We create now the operators to model a synthetic pre-stack seismic gather with a zero-phase using both a constant and a depth-variant vsvp profile

```

# constant vsvp
PPop_const = pyllops.avo.prestack.PrestackLinearModelling(
    wav, theta, vsvp=vsvp, nt0=nt0, linearization="akirich"
)

# depth-variant vsvp
PPop_variant = pyllops.avo.prestack.PrestackLinearModelling(
    wav, theta, vsvp=vsvp_z, linearization="akirich"
)

```

Let's apply those operators to the elastic model and create some synthetic data

```

dPP_const = PPop_const * m.ravel()
dPP_const = dPP_const.reshape(nt0, ntheta)

dPP_variant = PPop_variant * m.ravel()
dPP_variant = dPP_variant.reshape(nt0, ntheta)

```

Finally we visualize the two datasets

```

# sphinx_gallery_thumbnail_number = 2
fig = plt.figure(figsize=(6, 7))
ax1 = plt.subplot2grid((3, 2), (0, 0), rowspan=2)
ax2 = plt.subplot2grid((3, 2), (0, 1), rowspan=2)
ax3 = plt.subplot2grid((3, 2), (2, 0))
ax4 = plt.subplot2grid((3, 2), (2, 1))

```

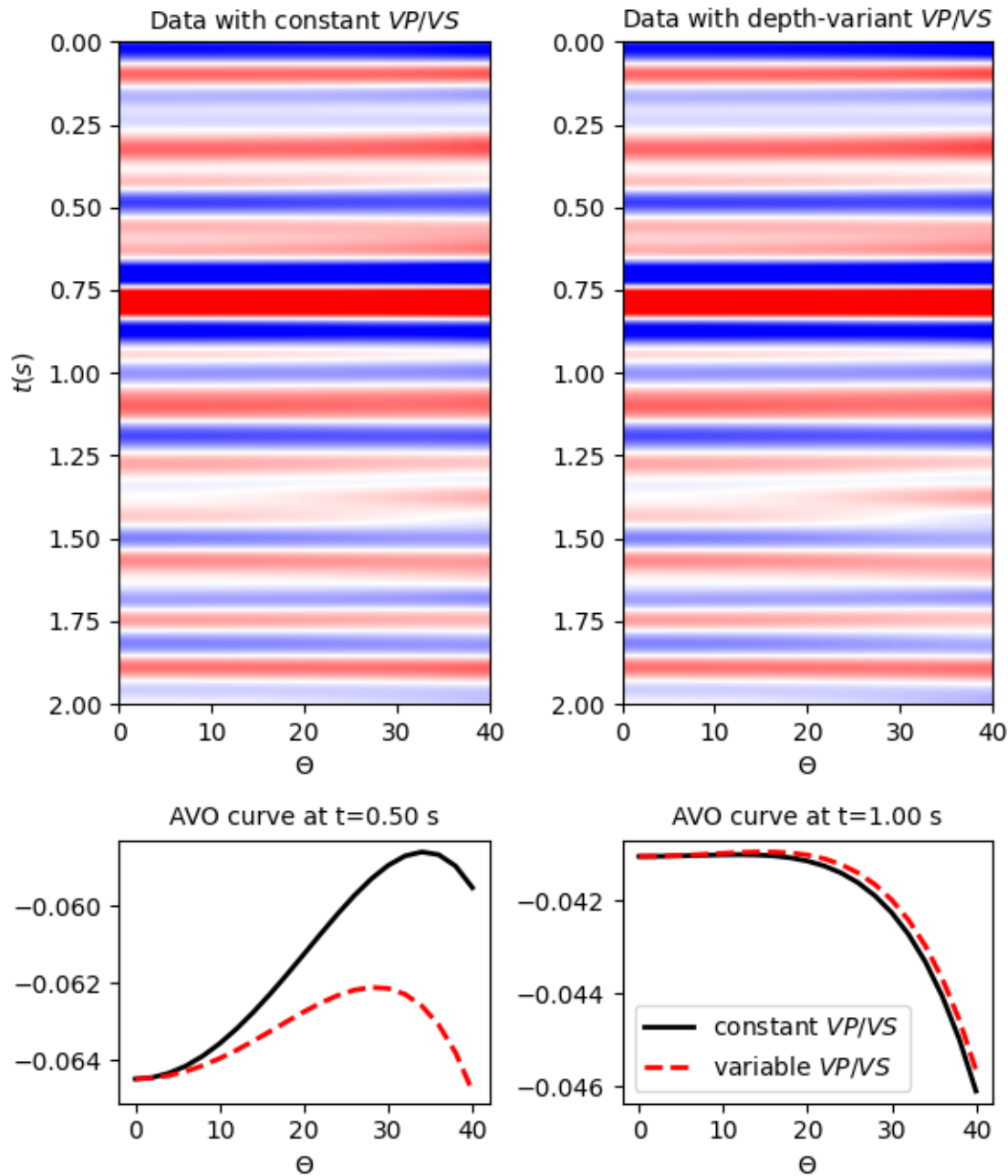
(continues on next page)

(continued from previous page)

```

ax1.imshow(
    dPP_const,
    cmap="bwr",
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=-0.1,
    vmax=0.1,
)
ax1.set_xlabel(r"$\Theta$")
ax1.set_ylabel(r"$t(s)$")
ax1.set_title(r"Data with constant $VP/V_S$", fontsize=10)
ax1.axis("tight")
ax2.imshow(
    dPP_variant,
    cmap="bwr",
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=-0.1,
    vmax=0.1,
)
ax2.set_title(r"Data with depth-variant $VP/V_S$", fontsize=10)
ax2.set_xlabel(r"$\Theta$")
ax2.axis("tight")
ax3.plot(theta, dPP_const[nt0 // 4], "k", lw=2)
ax3.plot(theta, dPP_variant[nt0 // 4], "--r", lw=2)
ax3.set_title("AVO curve at t=%.2f s" % t0[nt0 // 4], fontsize=10)
ax3.set_xlabel(r"$\Theta$")
ax4.plot(theta, dPP_const[nt0 // 2], "k", lw=2, label=r"constant $VP/V_S$")
ax4.plot(theta, dPP_variant[nt0 // 2], "--r", lw=2, label=r"variable $VP/V_S$")
ax4.set_title("AVO curve at t=%.2f s" % t0[nt0 // 2], fontsize=10)
ax4.set_xlabel(r"$\Theta$")
ax4.legend()
plt.tight_layout()

```



Total running time of the script: (0 minutes 0.761 seconds)

3.6.30 Radon Transform

This example shows how to use the `pylops.signalprocessing.Radon2D` and `pylops.signalprocessing.Radon3D` operators to apply the Radon Transform to 2-dimensional or 3-dimensional signals, respectively. In our implementation both linear, parabolic and hyperbolic parametrization can be chosen.

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

(continues on next page)

(continued from previous page)

```
import pylops

plt.close("all")
```

Let's start by creating an empty 2d matrix of size $n_{p_x} \times n_t$ and add a single spike in it. We will see that applying the forward Radon operator will result in a single event (linear, parabolic or hyperbolic) in the resulting data vector.

```
nt, nh = 41, 51
npx, pxmax = 41, 1e-2

dt, dh = 0.005, 1
t = np.arange(nt) * dt
h = np.arange(nh) * dh
px = np.linspace(0, pxmax, npix)

x = np.zeros((npx, nt))
x[4, nt // 2] = 1
```

We can now define our operators for different parametric curves and apply them to the input model vector. We also apply the adjoint to the resulting data vector.

```
RLOp = pylops.signalprocessing.Radon2D(
    t, h, px, centeredh=True, kind="linear", interp=False, engine="numpy"
)
RPOp = pylops.signalprocessing.Radon2D(
    t, h, px, centeredh=True, kind="parabolic", interp=False, engine="numpy"
)
RHOp = pylops.signalprocessing.Radon2D(
    t, h, px, centeredh=True, kind="hyperbolic", interp=False, engine="numpy"
)

# forward
yL = RLOp * x.ravel()
yP = RPOp * x.ravel()
yH = RHOp * x.ravel()
yL = yL.reshape(nh, nt)
yP = yP.reshape(nh, nt)
yH = yH.reshape(nh, nt)

# adjoint
xadjL = RLOp.H * yL.ravel()
xadjP = RPOp.H * yP.ravel()
xadjH = RHOp.H * yH.ravel()
xadjL = xadjL.reshape(npx, nt)
xadjP = xadjP.reshape(npx, nt)
xadjH = xadjH.reshape(npx, nt)
```

Let's now visualize the input model in the Radon domain, the data, and the adjoint model the different parametric curves.

```
fig, axs = plt.subplots(2, 4, figsize=(10, 6))
axs[0][0].imshow(
    x.T, vmin=-1, vmax=1, cmap="seismic_r", extent=(px[0], px[-1], t[-1], t[0])
)
axs[0][0].set_title("Input model")
```

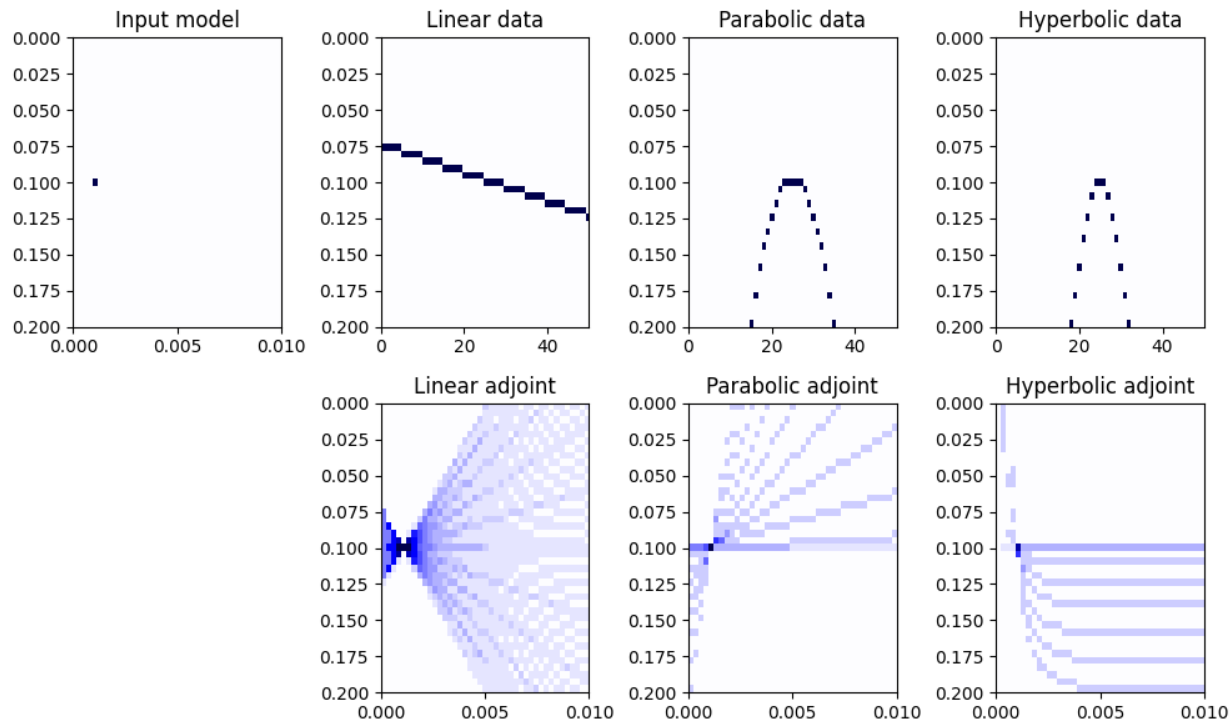
(continues on next page)

(continued from previous page)

```

axs[0][0].axis("tight")
axs[0][1].imshow(
    yL.T, vmin=-1, vmax=1, cmap="seismic_r", extent=(h[0], h[-1], t[-1], t[0])
)
axs[0][1].set_title("Linear data")
axs[0][1].axis("tight")
axs[0][2].imshow(
    yP.T, vmin=-1, vmax=1, cmap="seismic_r", extent=(h[0], h[-1], t[-1], t[0])
)
axs[0][2].set_title("Parabolic data")
axs[0][2].axis("tight")
axs[0][3].imshow(
    yH.T, vmin=-1, vmax=1, cmap="seismic_r", extent=(h[0], h[-1], t[-1], t[0])
)
axs[0][3].set_title("Hyperbolic data")
axs[0][3].axis("tight")
axs[1][1].imshow(
    xadjL.T, vmin=-20, vmax=20, cmap="seismic_r", extent=(px[0], px[-1], t[-1], t[0])
)
axs[1][0].axis("off")
axs[1][1].set_title("Linear adjoint")
axs[1][1].axis("tight")
axs[1][2].imshow(
    xadjP.T, vmin=-20, vmax=20, cmap="seismic_r", extent=(px[0], px[-1], t[-1], t[0])
)
axs[1][2].set_title("Parabolic adjoint")
axs[1][2].axis("tight")
axs[1][3].imshow(
    xadjH.T, vmin=-20, vmax=20, cmap="seismic_r", extent=(px[0], px[-1], t[-1], t[0])
)
axs[1][3].set_title("Hyperbolic adjoint")
axs[1][3].axis("tight")
fig.tight_layout()

```



As we can see in the bottom figures, the adjoint Radon transform is far from being close to the inverse Radon transform, i.e. $\mathbf{R}^H \mathbf{R} \neq \mathbf{I}$ (compared to the case of FFT where the adjoint and inverse are equivalent, i.e. $\mathbf{F}^H \mathbf{F} = \mathbf{I}$). In fact when we apply the adjoint Radon Transform we obtain a *model* that is a smoothed version of the original model polluted by smearing and artifacts. In tutorial [11. Radon filtering](#) we will exploit a sparsity-promoting Radon transform to perform filtering of unwanted signals from an input data.

Finally we repeat the same exercise with 3d data.

```
nt, ny, nx = 21, 21, 11
npy, pymax = 13, 5e-3
npx, pxmax = 11, 5e-3

dt, dy, dx = 0.005, 1, 1
t = np.arange(nt) * dt
hy = np.arange(ny) * dy
hx = np.arange(nx) * dx

py = np.linspace(0, pymax, npy)
px = np.linspace(0, pxmax, npx)

x = np.zeros((npy, npx, nt))
x[npy // 2, npx // 2 - 2, nt // 2] = 1

RLOp = pyllops.signalprocessing.Radon3D(
    t, hy, hx, py, px, centeredh=True, kind="linear", interp=False, engine="numpy"
)
RPOp = pyllops.signalprocessing.Radon3D(
    t, hy, hx, py, px, centeredh=True, kind="parabolic", interp=False, engine="numpy"
)
RHOp = pyllops.signalprocessing.Radon3D(
    t, hy, hx, py, px, centeredh=True, kind="hyperbolic", interp=False, engine="numpy"
)
```

(continues on next page)

(continued from previous page)

```

# forward
yL = Rlop * x.ravel()
yP = RPop * x.ravel()
yH = RHop * x.ravel()
yL = yL.reshape(ny, nx, nt)
yP = yP.reshape(ny, nx, nt)
yH = yH.reshape(ny, nx, nt)

# adjoint
xadjL = Rlop.H * yL.ravel()
xadjP = RPop.H * yP.ravel()
xadjH = RHop.H * yH.ravel()
xadjL = xadjL.reshape(npy, npx, nt)
xadjP = xadjP.reshape(npy, npx, nt)
xadjH = xadjH.reshape(npy, npx, nt)

# plotting
fig, axs = plt.subplots(2, 4, figsize=(10, 6))
axs[0][0].imshow(
    x[ny // 2].T,
    vmin=-1,
    vmax=1,
    cmap="seismic_r",
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[0][0].set_title("Input model")
axs[0][0].axis("tight")
axs[0][1].imshow(
    yL[ny // 2].T,
    vmin=-1,
    vmax=1,
    cmap="seismic_r",
    extent=(hx[0], hx[-1], t[-1], t[0]),
)
axs[0][1].set_title("Linear data")
axs[0][1].axis("tight")
axs[0][2].imshow(
    yP[ny // 2].T,
    vmin=-1,
    vmax=1,
    cmap="seismic_r",
    extent=(hx[0], hx[-1], t[-1], t[0]),
)
axs[0][2].set_title("Parabolic data")
axs[0][2].axis("tight")
axs[0][3].imshow(
    yH[ny // 2].T,
    vmin=-1,
    vmax=1,
    cmap="seismic_r",
    extent=(hx[0], hx[-1], t[-1], t[0]),
)
axs[0][3].set_title("Hyperbolic data")
axs[0][3].axis("tight")
axs[1][1].imshow(
    xadjL[ny // 2].T,

```

(continues on next page)

(continued from previous page)

```

    vmin=-100,
    vmax=100,
    cmap="seismic_r",
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[1][0].axis("off")
axs[1][1].set_title("Linear adjoint")
axs[1][1].axis("tight")
axs[1][2].imshow(
    xadjP[ny // 2].T,
    vmin=-100,
    vmax=100,
    cmap="seismic_r",
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[1][2].set_title("Parabolic adjoint")
axs[1][2].axis("tight")
axs[1][3].imshow(
    xadjH[ny // 2].T,
    vmin=-100,
    vmax=100,
    cmap="seismic_r",
    extent=(px[0], px[-1], t[-1], t[0]),
)
axs[1][3].set_title("Hyperbolic adjoint")
axs[1][3].axis("tight")
fig.tight_layout()

fig, axs = plt.subplots(2, 4, figsize=(10, 6))
axs[0][0].imshow(
    x[:, npx // 2 - 2].T,
    vmin=-1,
    vmax=1,
    cmap="seismic_r",
    extent=(py[0], py[-1], t[-1], t[0]),
)
axs[0][0].set_title("Input model")
axs[0][0].axis("tight")
axs[0][1].imshow(
    yL[:, nx // 2].T,
    vmin=-1,
    vmax=1,
    cmap="seismic_r",
    extent=(hy[0], hy[-1], t[-1], t[0]),
)
axs[0][1].set_title("Linear data")
axs[0][1].axis("tight")
axs[0][2].imshow(
    yP[:, nx // 2].T,
    vmin=-1,
    vmax=1,
    cmap="seismic_r",
    extent=(hy[0], hy[-1], t[-1], t[0]),
)
axs[0][2].set_title("Parabolic data")
axs[0][2].axis("tight")
axs[0][3].imshow(

```

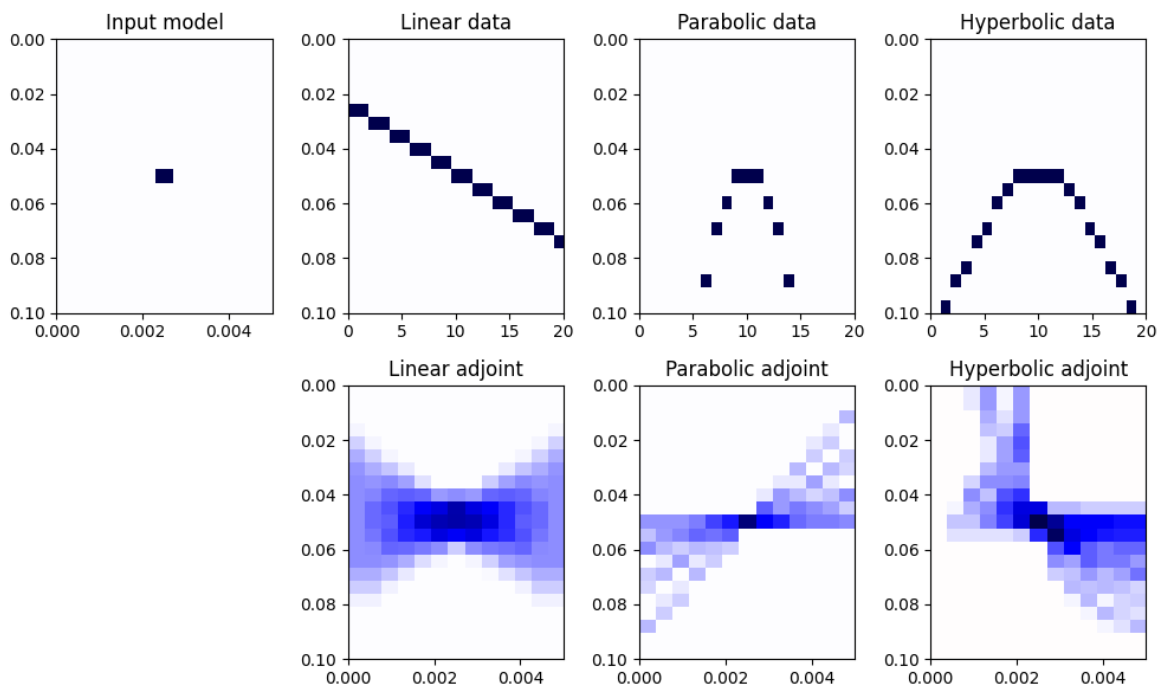
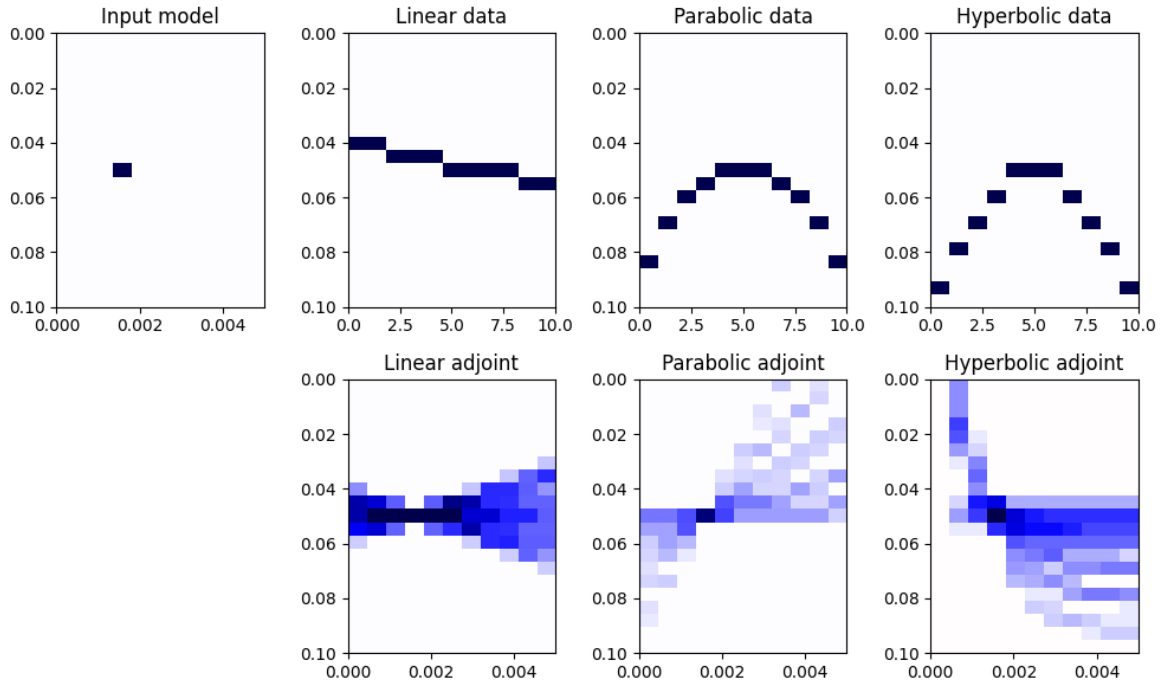
(continues on next page)

(continued from previous page)

```

    yH[:, nx // 2].T,
    vmin=-1,
    vmax=1,
    cmap="seismic_r",
    extent=(hy[0], hy[-1], t[-1], t[0]),
)
axs[0][3].set_title("Hyperbolic data")
axs[0][3].axis("tight")
axs[1][1].imshow(
    xadjL[:, npx // 2 - 5].T,
    vmin=-100,
    vmax=100,
    cmap="seismic_r",
    extent=(py[0], py[-1], t[-1], t[0]),
)
axs[1][0].axis("off")
axs[1][1].set_title("Linear adjoint")
axs[1][1].axis("tight")
axs[1][2].imshow(
    xadjP[:, npx // 2 - 2].T,
    vmin=-100,
    vmax=100,
    cmap="seismic_r",
    extent=(py[0], py[-1], t[-1], t[0]),
)
axs[1][2].set_title("Parabolic adjoint")
axs[1][2].axis("tight")
axs[1][3].imshow(
    xadjH[:, npx // 2 - 2].T,
    vmin=-100,
    vmax=100,
    cmap="seismic_r",
    extent=(py[0], py[-1], t[-1], t[0]),
)
axs[1][3].set_title("Hyperbolic adjoint")
axs[1][3].axis("tight")
fig.tight_layout()

```



Total running time of the script: (0 minutes 2.458 seconds)

3.6.31 Real

This example shows how to use the `pylops.basicoperators.Real` operator. This operator returns the real part of the data in forward and adjoint mode, but the forward output will be a real number, while the adjoint output will be a complex number with a zero-valued imaginary part.

```
import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt
import numpy as np

import pylops

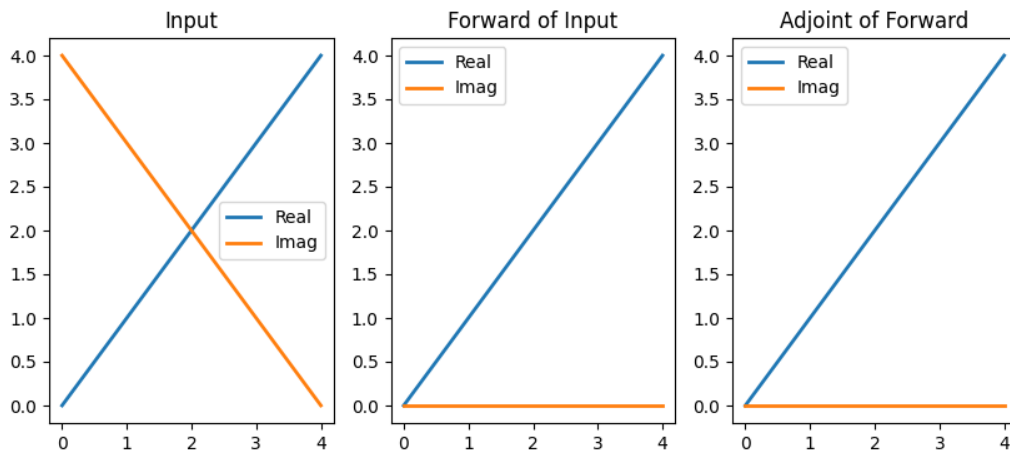
plt.close("all")
```

Let's define a Real operator \Re to extract the real component of the input.

```
M = 5
x = np.arange(M) + 1j * np.arange(M)[:-1]
Rop = pylops.basicoperators.Real(M, dtype="complex128")

y = Rop * x
xadj = Rop.H * y

_, axs = plt.subplots(1, 3, figsize=(10, 4))
axs[0].plot(np.real(x), lw=2, label="Real")
axs[0].plot(np.imag(x), lw=2, label="Imag")
axs[0].legend()
axs[0].set_title("Input")
axs[1].plot(np.real(y), lw=2, label="Real")
axs[1].plot(np.imag(y), lw=2, label="Imag")
axs[1].legend()
axs[1].set_title("Forward of Input")
axs[2].plot(np.real(xadj), lw=2, label="Real")
axs[2].plot(np.imag(xadj), lw=2, label="Imag")
axs[2].legend()
axs[2].set_title("Adjoint of Forward")
```



Out:

```
Text(0.5, 1.0, 'Adjoint of Forward')
```

Total running time of the script: (0 minutes 0.261 seconds)

3.6.32 Restriction and Interpolation

This example shows how to use the `pylops.Restriction` operator to sample a certain input vector at desired locations `iava`. Moreover, we go one step further and use the `pylops.signalprocessing.Interp` operator to show how we can also sample values at locations that are not exactly on the grid of the input vector.

As explained in the [03. Solvers](#) tutorial, such operators can be used as forward model in an inverse problem aimed at interpolate irregularly sampled 1d or 2d signals onto a regular grid.

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

import pylops

plt.close("all")
np.random.seed(10)
```

Let's create a signal of size `nt` and sampling `dt` that is composed of three sinusoids at frequencies `freqs`.

```
nt = 200
dt = 0.004

freqs = [5.0, 3.0, 8.0]

t = np.arange(nt) * dt
x = np.zeros(nt)

for freq in freqs:
    x = x + np.sin(2 * np.pi * freq * t)
```

First of all, we subsample the signal at random locations and we retain 40% of the initial samples.

```
perc_subsampling = 0.4
ntsub = int(np.round(nt * perc_subsampling))

isample = np.arange(nt)
iava = np.sort(np.random.permutation(np.arange(nt))[:ntsub])
```

We then create the restriction and interpolation operators and display the original signal as well as the subsampled signal.

```
Rop = pylops.Restriction(nt, iava, dtype="float64")
NNop, iavann = pylops.signalprocessing.Interp(
    nt, iava + 0.4, kind="nearest", dtype="float64"
)
LIop, iavali = pylops.signalprocessing.Interp(
    nt, iava + 0.4, kind="linear", dtype="float64"
)
SIop, iavasi = pylops.signalprocessing.Interp(
    nt, iava + 0.4, kind="sinc", dtype="float64"
)

y = Rop * x
ynn = NNop * x
yli = LIop * x
ysi = SIop * x
ymask = Rop.mask(x)

# Visualize data
```

(continues on next page)

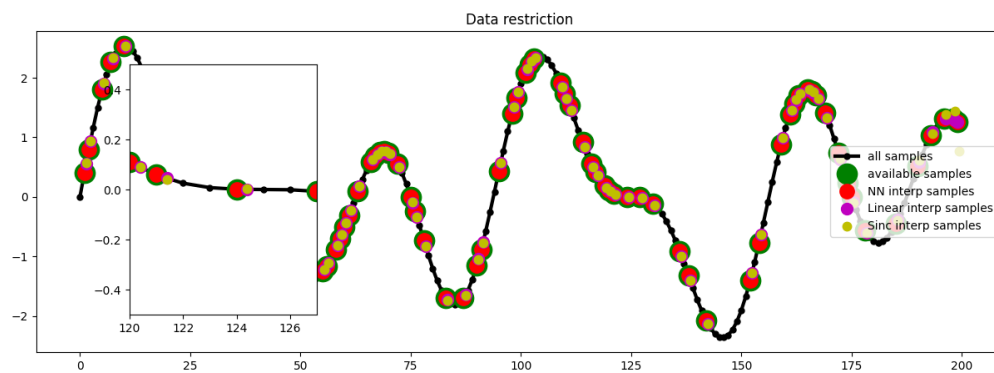
(continued from previous page)

```

fig = plt.figure(figsize=(15, 5))
plt.plot(isample, x, "-k", lw=3, ms=10, label="all samples")
plt.plot(isample, ymask, ".g", ms=35, label="available samples")
plt.plot(iavann, ynn, ".r", ms=25, label="NN interp samples")
plt.plot(iavali, yli, ".m", ms=20, label="Linear interp samples")
plt.plot(iavasi, ysi, ".y", ms=15, label="Sinc interp samples")
plt.legend(loc="right")
plt.title("Data restriction")

subax = fig.add_axes([0.2, 0.2, 0.15, 0.6])
subax.plot(isample, x, "-k", lw=3, ms=10)
subax.plot(isample, ymask, ".g", ms=35)
subax.plot(iavann, ynn, ".r", ms=25)
subax.plot(iavali, yli, ".m", ms=20)
subax.plot(iavasi, ysi, ".y", ms=15)
subax.set_xlim([120, 127])
subax.set_ylim([-0.5, 0.5])

```



Out:

```
(-0.5, 0.5)
```

Finally we show how the `pylops.Restriction` is not limited to one dimensional signals but can be applied to sample locations of a specific axis of a multi-dimensional array. subsampling locations

```

nx, nt = 100, 50

x = np.random.normal(0, 1, (nx, nt))

perc_subsampling = 0.4
nxsub = int(np.round(nx * perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(nx))[:nxsub])

Rop = pylops.Restriction(nx * nt, iava, dims=(nx, nt), dir=0, dtype="float64")
y = (Rop * x.ravel()).reshape(nxsub, nt)
ymask = Rop.mask(x)

fig, axs = plt.subplots(1, 3, figsize=(10, 5))
axs[0].imshow(x.T, cmap="gray")
axs[0].set_title("Model")
axs[0].axis("tight")
axs[1].imshow(y.T, cmap="gray")

```

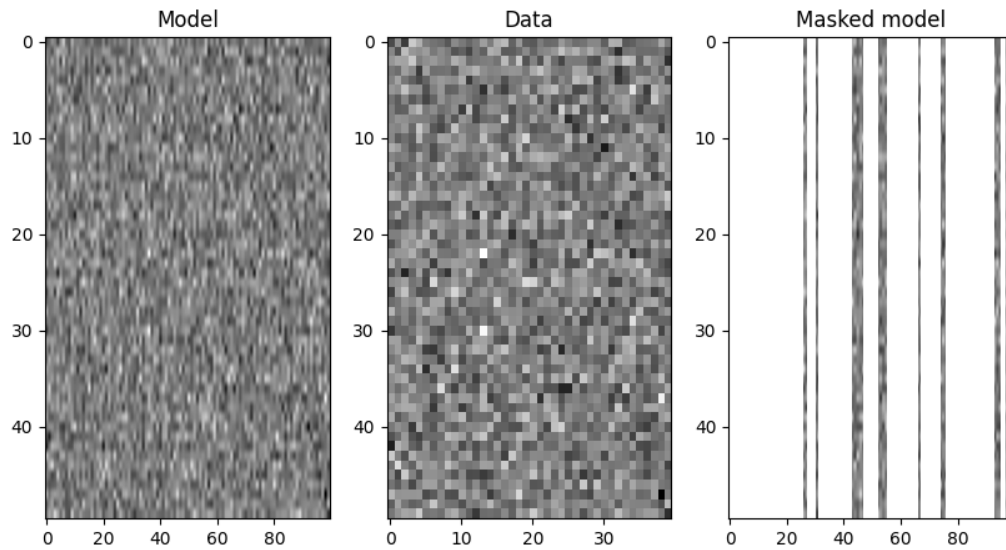
(continues on next page)

(continued from previous page)

```

axs[1].set_title("Data")
axs[1].axis("tight")
axs[2].imshow(ymask.T, cmap="gray")
axs[2].set_title("Masked model")
axs[2].axis("tight")

```



Out:

```
(-0.5, 99.5, 49.5, -0.5)
```

Total running time of the script: (0 minutes 0.572 seconds)

3.6.33 Roll

This example shows how to use the `pylops.Roll` operator.

This operator simply shifts elements of multi-dimensional array along a specified direction a chosen number of samples.

```

import matplotlib.pyplot as plt
import numpy as np

import pylops

plt.close("all")

```

Let's start with a 1d example. We make a signal, shift it by two samples and then shift it back using its adjoint. We can immediately see how the adjoint of this operator is equivalent to its inverse.

```

nx = 10
x = np.arange(nx)

Rop = pylops.Roll(nx, shift=2)

```

(continues on next page)

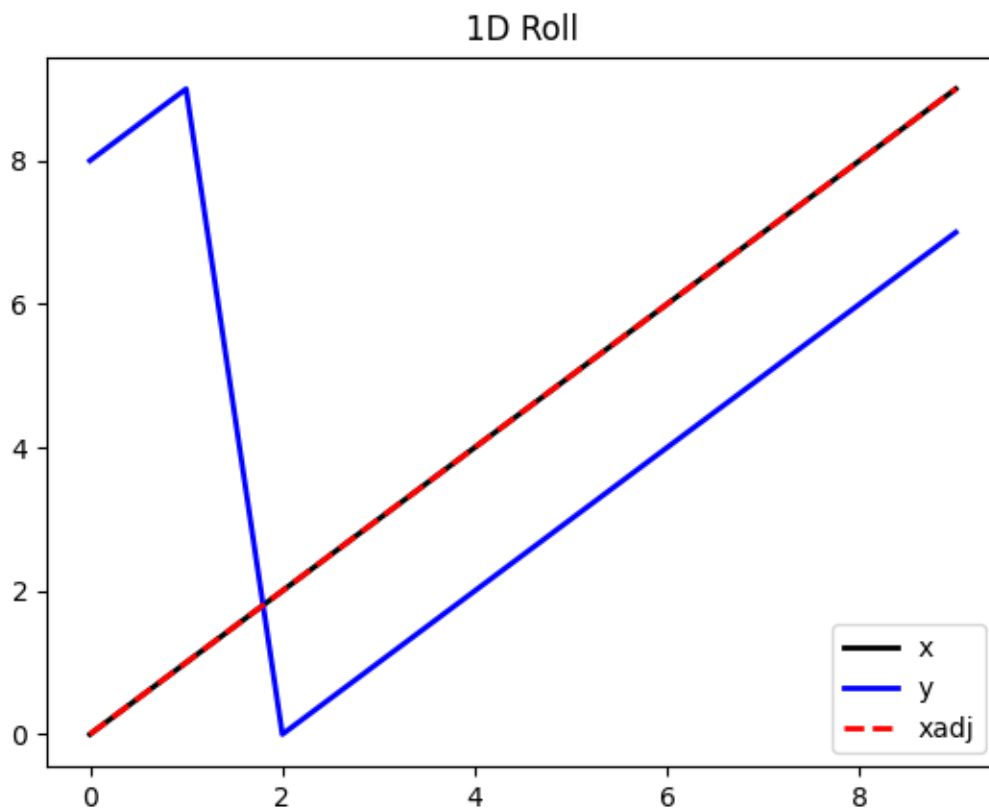
(continued from previous page)

```

y = Rop * x
xadj = Rop.H * y

plt.figure()
plt.plot(x, "k", lw=2, label="x")
plt.plot(y, "b", lw=2, label="y")
plt.plot(xadj, "--r", lw=2, label="xadj")
plt.title("1D Roll")
plt.legend()

```



Out:

```
<matplotlib.legend.Legend object at 0x7f6ae992bb00>
```

We can now do the same with a 2d array.

```

ny, nx = 10, 5
x = np.arange(ny * nx).reshape(ny, nx)

Rop = pylops.Roll(ny * nx, dims=(ny, nx), dir=1, shift=-2)

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

```

(continues on next page)

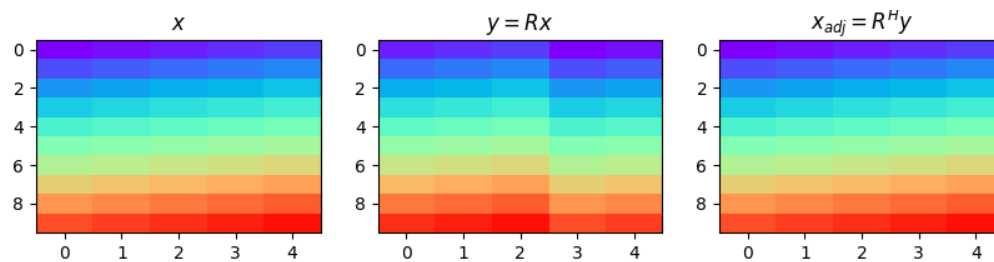
(continued from previous page)

```

y = y.reshape(ny, nx)
xadj = xadj.reshape(ny, nx)

fig, axs = plt.subplots(1, 3, figsize=(10, 2))
fig.suptitle("Roll for 2d data", fontsize=14, fontweight="bold", y=1.15)
axs[0].imshow(x, cmap="rainbow", vmin=0, vmax=50)
axs[0].set_title(r"$x$")
axs[0].axis("tight")
axs[1].imshow(y, cmap="rainbow", vmin=0, vmax=50)
axs[1].set_title(r"$y = R x$")
axs[1].axis("tight")
axs[2].imshow(xadj, cmap="rainbow", vmin=0, vmax=50)
axs[2].set_title(r"$x_{adj} = R^H y$")
axs[2].axis("tight")

```



Out:

```
(-0.5, 4.5, 9.5, -0.5)
```

Total running time of the script: (0 minutes 0.314 seconds)

3.6.34 Seislet transform

This example shows how to use the `pylops.signalprocessing.Seislet` operator. This operator the forward, adjoint and inverse Seislet transform that is a modification of the well-know Wavelet transform where local slopes are used in the prediction and update steps to further improve the prediction of a trace from its previous (or subsequent) one and reduce the amount of information passed to the subsequent scale. While this transform was initially developed in the context of processing and compression of seismic data, it is also suitable to any other oscillatory dataset such as GPR or Acoustic recordings.

```

import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import MaxNLocator
from mpl_toolkits.axes_grid1 import make_axes_locatable

import pylops

plt.close("all")

```

In this example we use the same benchmark `dataset` that was used in the original paper describing the Seislet transform. First, local slopes are estimated using `pylops.utils.signalprocessing.slope_estimate`.

```
inputfile = "../testdata/sigmoid.npz"
```

(continues on next page)

(continued from previous page)

```

d = np.load(inputfile)
d = d["sigmoid"]
nx, nt = d.shape
dx, dt = 0.008, 0.004
x, t = np.arange(nx) * dx, np.arange(nt) * dt

# slope estimation
slope, _ = pylops.utils.signalprocessing.slope_estimate(d.T, dt, dx, smooth=2.5)
slope *= -1 # t-axis points down
# clip slopes above 80°
pmax = np.arctan(80 * np.pi / 180)
slope[slope > pmax] = pmax
slope[slope < -pmax] = -pmax

```

```

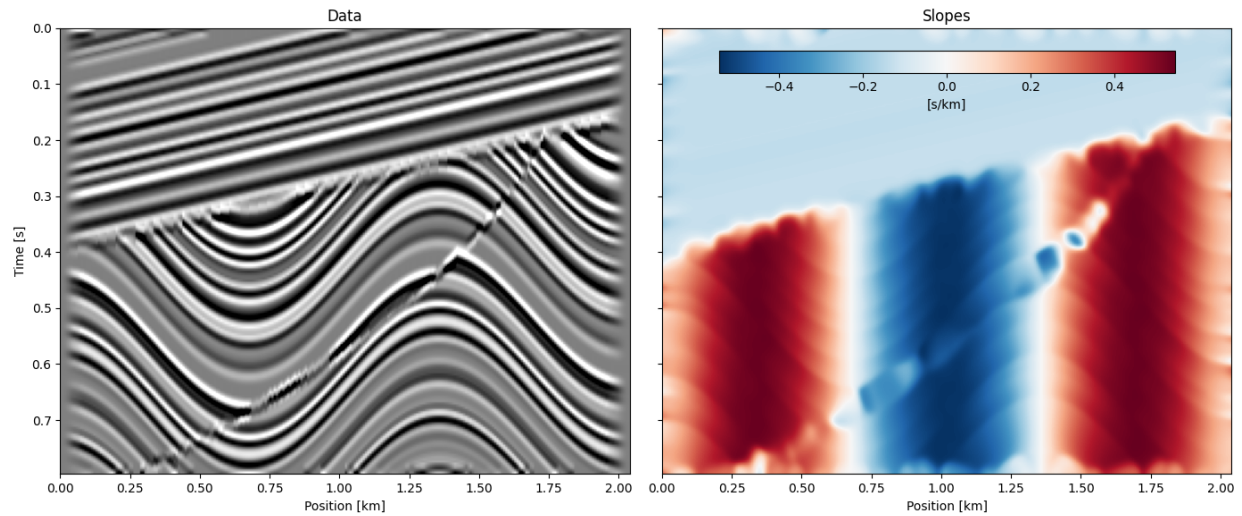
clip = 0.5 * np.max(np.abs(d))
clip_s = min(pmax, np.max(np.abs(slope)))
opts = dict(aspect=2, extent=(x[0], x[-1], t[-1], t[0]))

fig, axs = plt.subplots(1, 2, figsize=(14, 7), sharey=True, sharex=True)
axs[0].imshow(d.T, cmap="gray", vmin=-clip, vmax=clip, **opts)
axs[0].set(xlabel="Position [km]", ylabel="Time [s]", title="Data")

im = axs[1].imshow(slope, cmap="RdBu_r", vmin=-clip_s, vmax=clip_s, **opts)
axs[1].set(xlabel="Position [km]", title="Slopes")
fig.tight_layout()

pos = axs[1].get_position()
cbpos = [
    pos.x0 + 0.1 * pos.width,
    pos.y0 + 0.9 * pos.height,
    0.8 * pos.width,
    0.05 * pos.height,
]
cax = fig.add_axes(cbpos)
cb = fig.colorbar(im, cax=cax, orientation="horizontal")
cb.set_label("[s/km]")

```



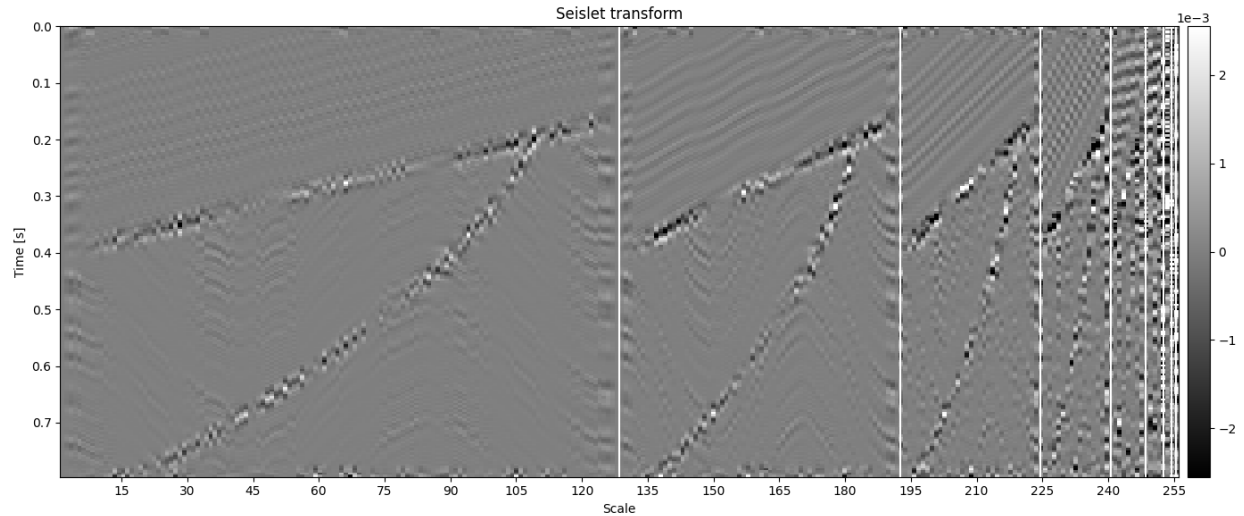
Next the Seislet transform is computed.

```
Sop = pyllops.signalprocessing.Seislet(slope.T, sampling=(dx, dt))

seis = Sop * d.ravel()
seis = seis.reshape(nx, nt)

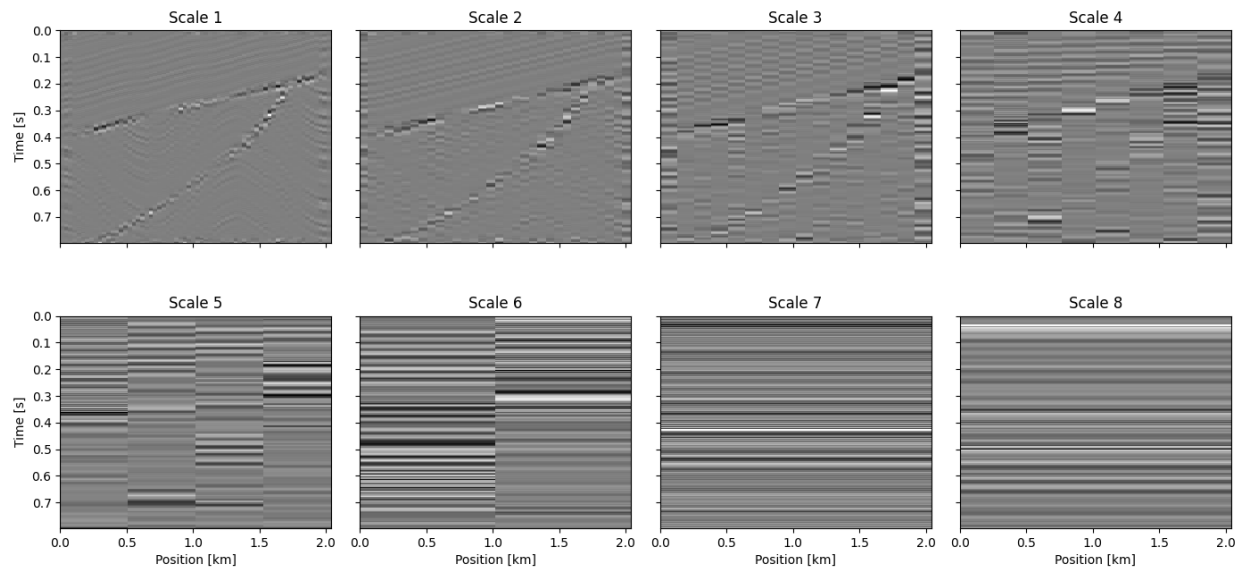
nlevels_max = int(np.log2(nx))
levels_size = np.flip(np.array([2 ** i for i in range(nlevels_max)]))
levels_cum = np.cumsum(levels_size)
```

```
fig, ax = plt.subplots(figsize=(14, 6))
im = ax.imshow(
    seis.T,
    cmap="gray",
    vmin=-clip,
    vmax=clip,
    aspect="auto",
    interpolation="none",
    extent=(1, seis.shape[0], t[-1], t[0]),
)
ax.xaxis.set_major_locator(MaxNLocator(nbins=20, integer=True))
for level in levels_cum:
    ax.axvline(level + 0.5, color="w")
ax.set(xlabel="Scale", ylabel="Time [s]", title="Seislet transform")
cax = make_axes_locatable(ax).append_axes("right", size="2%", pad=0.1)
cb = fig.colorbar(im, cax=cax, orientation="vertical")
cb.formatter.set_powerlimits((0, 0))
fig.tight_layout()
```



We may also stretch the finer scales to be the width of the image

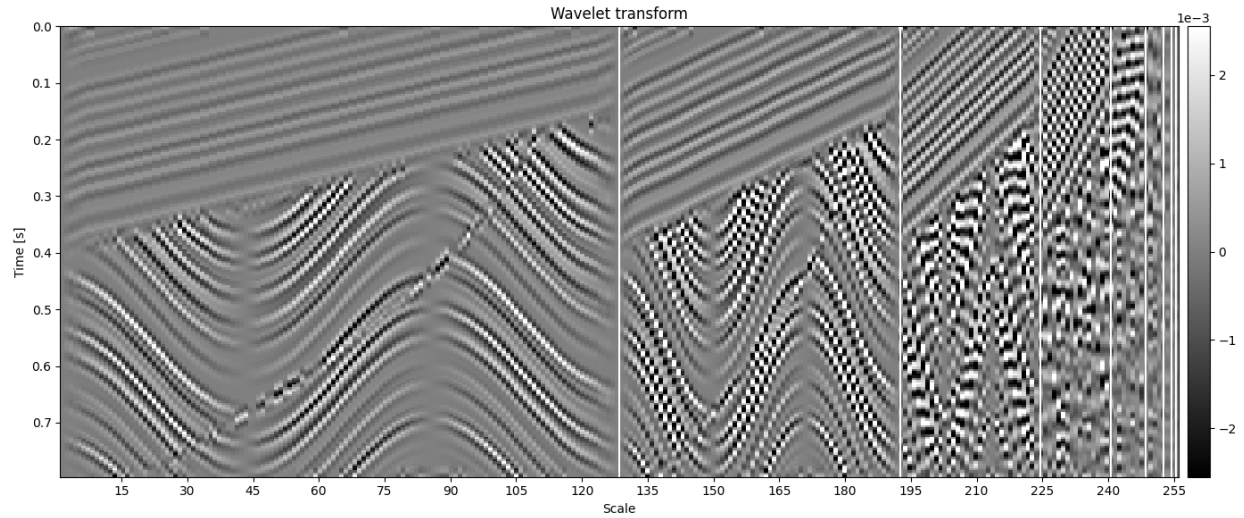
```
fig, axs = plt.subplots(2, nlevels_max // 2, figsize=(14, 7), sharex=True,
    ↳sharey=True)
for i, ax in enumerate(axs.ravel()[:-1]):
    curdata = seis[levels_cum[i] : levels_cum[i + 1], :].T
    vmax = np.max(np.abs(curdata))
    ax.imshow(curdata, vmin=-vmax, vmax=vmax, cmap="gray", interpolation="none",
    ↳**opts)
    ax.set(title=f"Scale {i+1}")
    if i + 1 > nlevels_max // 2:
        ax.set(xlabel="Position [km]")
curdata = seis[levels_cum[-1] :, :].T
vmax = np.max(np.abs(curdata))
axs[-1, -1].imshow(
    curdata, vmin=-vmax, vmax=vmax, cmap="gray", interpolation="none", **opts
)
axs[0, 0].set(ylabel="Time [s]")
axs[1, 0].set(ylabel="Time [s]")
axs[-1, -1].set(xlabel="Position [km]", title=f"Scale {nlevels_max}")
fig.tight_layout()
```

As a comparison we also compute the Seislet transform fixing slopes to zero. This way we turn the Seislet transform into a basic 1D Wavelet transform performed over the spatial axis.

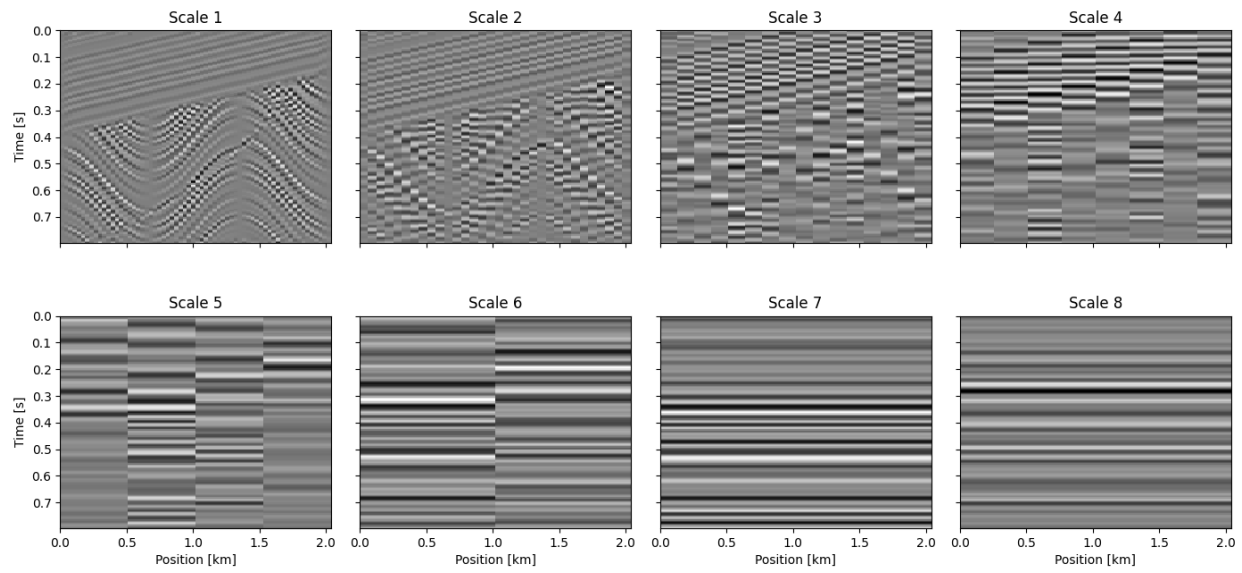
```
Wop = pyllops.signalprocessing.Seislet(np.zeros_like(slope.T), sampling=(dx, dt))
dwt = Wop * d.ravel()
dwt = dwt.reshape(nx, nt)
```

```
fig, ax = plt.subplots(figsize=(14, 6))
im = ax.imshow(
    dwt.T,
    cmap="gray",
    vmin=-clip,
    vmax=clip,
    aspect="auto",
    interpolation="none",
    extent=(1, dwt.shape[0], t[-1], t[0]),
)
ax.xaxis.set_major_locator(MaxNLocator(nbins=20, integer=True))
for level in levels_cum:
    ax.axvline(level + 0.5, color="w")
ax.set(xlabel="Scale", ylabel="Time [s]", title="Wavelet transform")
cax = make_axes_locatable(ax).append_axes("right", size="2%", pad=0.1)
cb = fig.colorbar(im, cax=cax, orientation="vertical")
cb.formatter.set_powerlimits((0, 0))
fig.tight_layout()
```



Again, we may decompress the finer scales

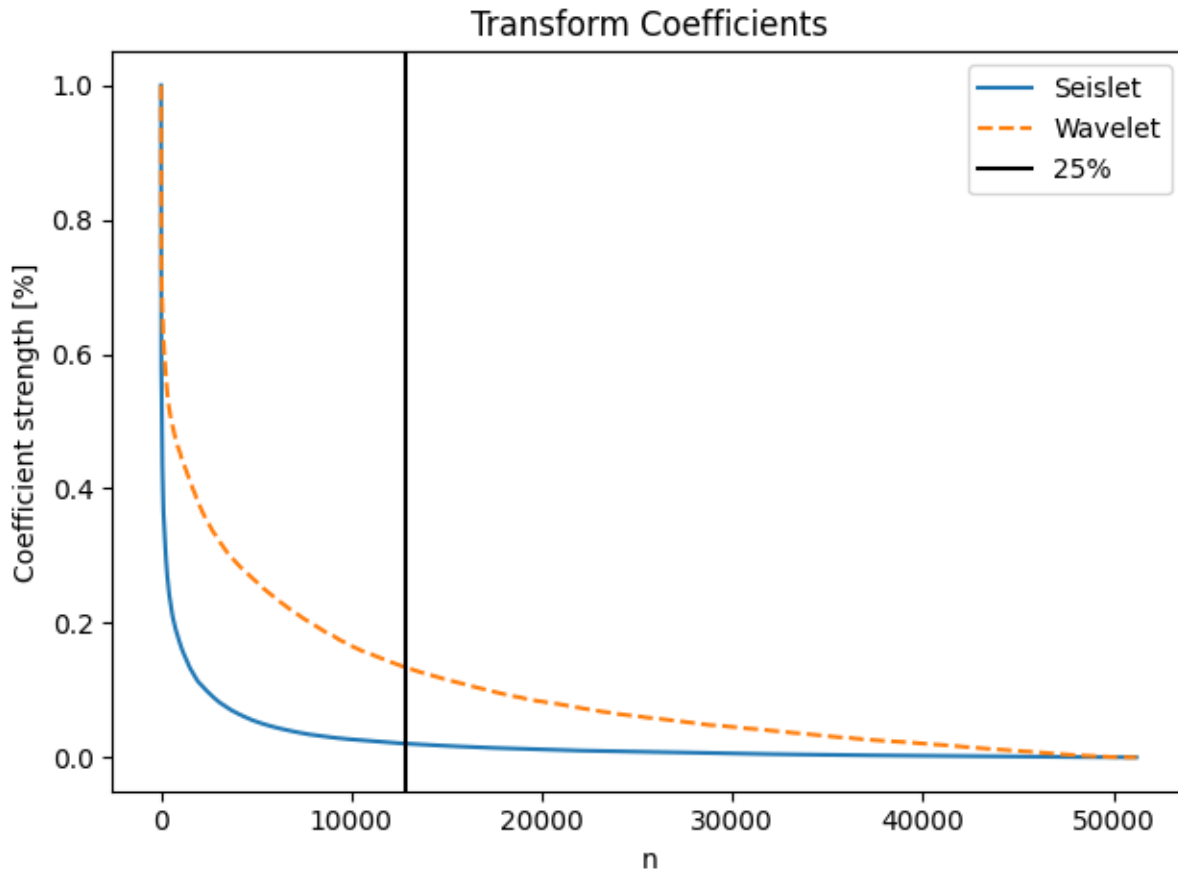
```
fig, axs = plt.subplots(2, nlevels_max // 2, figsize=(14, 7), sharex=True,
    ↳sharey=True)
for i, ax in enumerate(axs.ravel()[:-1]):
    curdata = dwt[levels_cum[i] : levels_cum[i + 1], :].T
    vmax = np.max(np.abs(curdata))
    ax.imshow(curdata, vmin=-vmax, vmax=vmax, cmap="gray", interpolation="none",
    ↳**opts)
    ax.set(title=f"Scale {i+1}")
    if i + 1 > nlevels_max // 2:
        ax.set(xlabel="Position [km]")
curdata = dwt[levels_cum[-1] :, :].T
vmax = np.max(np.abs(curdata))
axs[-1, -1].imshow(
    curdata, vmin=-vmax, vmax=vmax, cmap="gray", interpolation="none", **opts
)
axs[0, 0].set(ylabel="Time [s]")
axs[1, 0].set(ylabel="Time [s]")
axs[-1, -1].set(xlabel="Position [km]", title=f"Scale {nlevels_max}")
fig.tight_layout()
```



Finally we evaluate the compression capabilities of the Seislet transform compared to the 1D Wavelet transform. We zero-out all but the strongest 25% of the components. We perform the inverse transforms and assess the compression error.

```
perc = 0.25
seis_strong_idx = np.argsort(-np.abs(seis.ravel()))
dwt_strong_idx = np.argsort(-np.abs(dwt.ravel()))
seis_strong = np.abs(seis.ravel())[seis_strong_idx]
dwt_strong = np.abs(dwt.ravel())[dwt_strong_idx]
```

```
fig, ax = plt.subplots()
ax.plot(range(1, len(seis_strong) + 1), seis_strong / seis_strong[0], label="Seislet")
ax.plot(
    range(1, len(dwt_strong) + 1), dwt_strong / dwt_strong[0], "--", label="Wavelet"
)
ax.set(xlabel="n", ylabel="Coefficient strength [%]", title="Transform Coefficients")
ax.axvline(np rint(len(seis_strong) * perc), color="k", label=f"{100*perc:.0f}%")
ax.legend()
fig.tight_layout()
```



```
seisl = np.zeros_like(seis.ravel())
seis_strong_idx = seis_strong_idx[: int(np rint(len(seis_strong) * perc))]
seisl[seis_strong_idx] = seis.ravel()[seis_strong_idx]
d_seis = Sop.inverse(seisl)
d_seis = d_seis.reshape(nx, nt)

dwt1 = np.zeros_like(dwt.ravel())
dwt_strong_idx = dwt_strong_idx[: int(np rint(len(dwt_strong) * perc))]
dwt1[dwt_strong_idx] = dwt.ravel()[dwt_strong_idx]
d_dwt = Wop.inverse(dwt1)
d_dwt = d_dwt.reshape(nx, nt)
```

```
opts.update(dict(cmap="gray", vmin=-clip, vmax=clip))
fig, axs = plt.subplots(2, 3, figsize=(14, 7), sharex=True, sharey=True)
axs[0, 0].imshow(d.T, **opts)
axs[0, 0].set(title="Data")
axs[0, 1].imshow(d_seis.T, **opts)
axs[0, 1].set(title=f"Rec. from Seislet ({100*perc:.0f}% of coeffs.)")
axs[0, 2].imshow((d - d_seis).T, **opts)
axs[0, 2].set(title="Error from Seislet Rec.")
axs[1, 0].imshow(d.T, **opts)
axs[1, 0].set(ylabel="Time [s]", title="Data [Repeat]")
axs[1, 1].imshow(d_dwt.T, **opts)
axs[1, 1].set(title=f"Rec. from Wavelet ({100*perc:.0f}% of coeffs.)")
axs[1, 2].imshow((d - d_dwt).T, **opts)
```

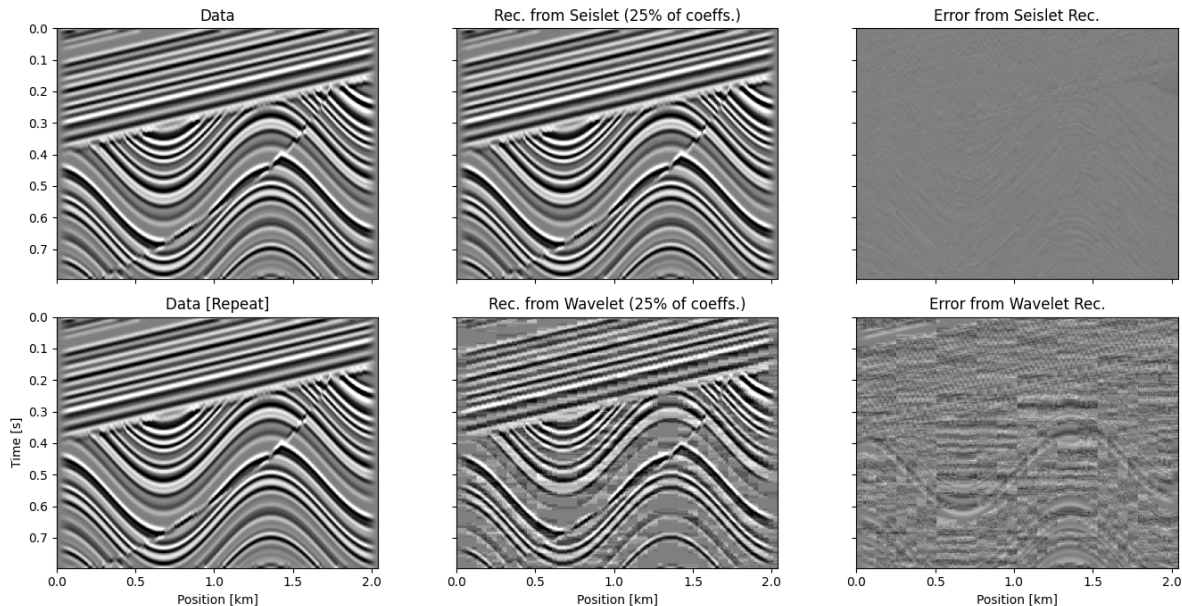
(continues on next page)

(continued from previous page)

```

axs[1, 2].set(title="Error from Wavelet Rec.")
for i in range(3):
    axs[1, i].set(xlabel="Position [km]")
plt.tight_layout()

```



To conclude it is worth noting that the Seislet transform, differently to the Wavelet transform, is not orthogonal: in other words, its adjoint and inverse are not equivalent. While we have used the forward and inverse transformations, when used as linear operator in composition with other operators, the Seislet transform requires the adjoint be defined and that it also passes the dot-test pair that is. As shown below, this is the case when using the implementation in the PyLops package.

```
pylops.utils.dottest(Sop, nt * nx, nt * nx, verb=True)
```

Out:

```

Dot test passed, v^H(Opu)=-84.40995891105774 - u^H(Op^Hv)=-84.40995891105746
True

```

Total running time of the script: (1 minutes 9.030 seconds)

3.6.35 Shift

This example shows how to use the `pylops.signalprocessing.Shift` operator to apply fractional delay to an input signal. Whilst this operator acts on 1D signals it can also be applied on any multi-dimensional signal on a specific direction of it.

```

import matplotlib.pyplot as plt
import numpy as np

import pylops

plt.close("all")

```

Let's start with a 1D example. Define the input parameters: number of samples of input signal (nt), sampling step (dt) as well as the input signal which will be equal to a ricker wavelet:

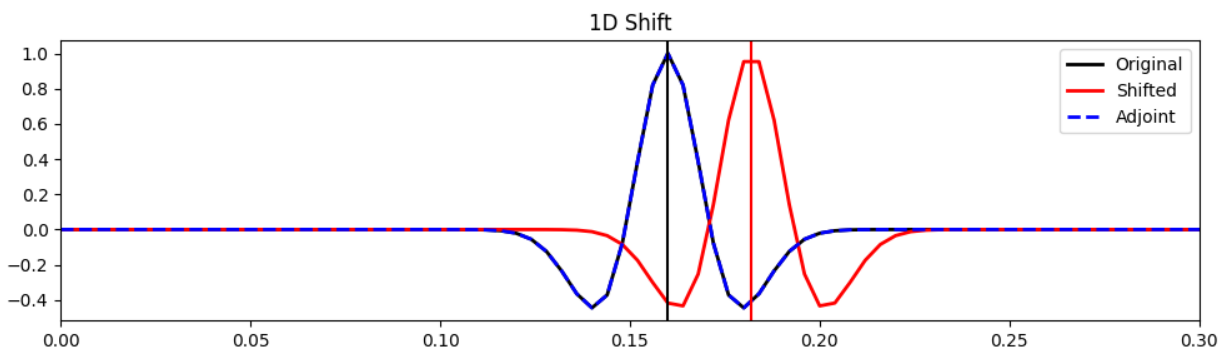
```
nt = 127
dt = 0.004
t = np.arange(nt) * dt
ntwav = 41

wav = pyllops.utils.wavelets.ricker(t[:ntwav], f0=20)[0]
wav = np.pad(wav, [0, nt - len(wav)])
WAV = np.fft.rfft(wav, n=nt)
```

We can shift this wavelet by $5.5 * dt$:

```
shift = 5.5 * dt
Op = pyllops.signalprocessing.Shift(nt, shift, sampling=dt, real=True, dtype=np.
    ↪float64)
wavshift = Op * wav
wavshiftback = Op.H * wavshift

plt.figure(figsize=(10, 3))
plt.plot(t, wav, "k", lw=2, label="Original")
plt.plot(t, wavshift, "r", lw=2, label="Shifted")
plt.plot(t, wavshiftback, "--b", lw=2, label="Adjoint")
plt.axvline(t[ntwav - 1], color="k")
plt.axvline(t[ntwav - 1] + shift, color="r")
plt.xlim(0, 0.3)
plt.legend()
plt.title("1D Shift")
plt.tight_layout()
```



We can repeat the same exercise for a 2D signal and perform the shift along the first and second dimensions.

```
shift = 10.5 * dt

# 1st dir
wav2d = np.outer(wav, np.ones(10))
Op = pyllops.signalprocessing.Shift(
    (nt, 10), shift, dir=0, sampling=dt, real=True, dtype=np.float64
)
wav2dshift = (Op * wav2d.ravel()).reshape(nt, 10)
wav2dshiftback = (Op.H * wav2dshift.ravel()).reshape(nt, 10)

fig, axs = plt.subplots(1, 3, figsize=(10, 3))
```

(continues on next page)

(continued from previous page)

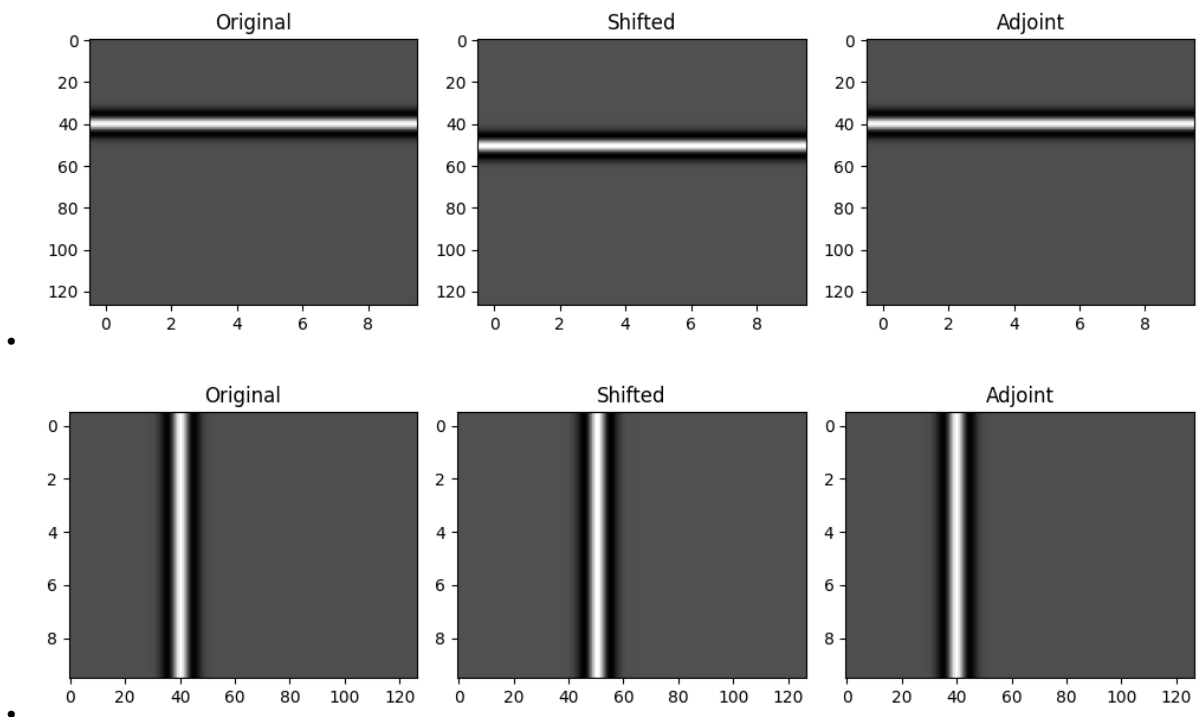
```

axs[0].imshow(wav2d, cmap="gray")
axs[0].axis("tight")
axs[0].set_title("Original")
axs[1].imshow(wav2dshift, cmap="gray")
axs[1].set_title("Shifted")
axs[1].axis("tight")
axs[2].imshow(wav2dshiftback, cmap="gray")
axs[2].set_title("Adjoint")
axs[2].axis("tight")
fig.tight_layout()

# 2nd dir
wav2d = np.outer(wav, np.ones(10)).T
Op = pylops.signalprocessing.Shift(
    (10, nt), shift, dir=1, sampling=dt, real=True, dtype=np.float64
)
wav2dshift = (Op * wav2d.ravel()).reshape(10, nt)
wav2dshiftback = (Op.H * wav2dshift.ravel()).reshape(10, nt)

fig, axs = plt.subplots(1, 3, figsize=(10, 3))
axs[0].imshow(wav2d, cmap="gray")
axs[0].axis("tight")
axs[0].set_title("Original")
axs[1].imshow(wav2dshift, cmap="gray")
axs[1].set_title("Shifted")
axs[1].axis("tight")
axs[2].imshow(wav2dshiftback, cmap="gray")
axs[2].set_title("Adjoint")
axs[2].axis("tight")
fig.tight_layout()

```



Total running time of the script: (0 minutes 0.823 seconds)

3.6.36 Slope estimation via Structure Tensor algorithm

This example shows how to estimate local slopes of a two-dimensional array using `pylops.utils.signalprocessing.slope_estimate`.

Knowing the local slopes of an image (or a seismic data) can be useful for a variety of tasks in image (or geo-physical) processing such as denoising, smoothing, or interpolation. When slopes are used with the `pylops.signalprocessing.Seislet` operator, the input dataset can be compressed and the sparse nature of the Seislet transform can also be used to precondition sparsity-promoting inverse problems.

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.axes_grid1 import make_axes_locatable

import pylops
from pylops.signalprocessing.Seislet import _predict_trace

plt.close("all")
np.random.seed(10)
```

To start we import a 2d image and estimate the local slopes of the image.

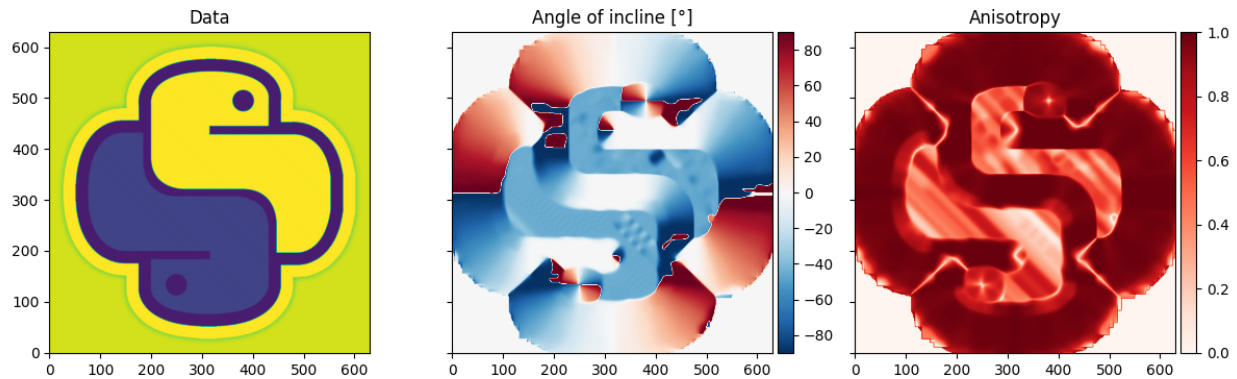
```
im = np.load("../testdata/python.npy")[..., 0]
im = im / 255.0 - 0.5

slopes, anisotropy = pylops.utils.signalprocessing.slope_estimate(im, smooth=7)
angles = -np.rad2deg(np.arctan(slopes))
```

```
fig, axs = plt.subplots(1, 3, figsize=(12, 4), sharex=True, sharey=True)
iax = axs[0].imshow(im, cmap="viridis", origin="lower")
axs[0].set_title("Data")
cax = make_axes_locatable(axs[0]).append_axes("right", size="5%", pad=0.05)
cax.axis("off")

iax = axs[1].imshow(angles, cmap="RdBu_r", origin="lower", vmin=-90, vmax=90)
axs[1].set_title("Angle of incline [°]")
cax = make_axes_locatable(axs[1]).append_axes("right", size="5%", pad=0.05)
cb = fig.colorbar(iax, cax=cax, orientation="vertical")

iax = axs[2].imshow(anisotropy, cmap="Reds", origin="lower", vmin=0, vmax=1)
axs[2].set_title("Anisotropy")
cax = make_axes_locatable(axs[2]).append_axes("right", size="5%", pad=0.05)
cb = fig.colorbar(iax, cax=cax, orientation="vertical")
fig.tight_layout()
```

We can now repeat the same using some seismic data. We will first define a single trace and a slope field, apply such slope field to the trace recursively to create the other traces of the data and finally try to recover the underlying slope field from the data alone.

```
# Reflectivity model
nx, nt = 2 ** 7, 121
dx, dt = 0.01, 0.004
x, t = np.arange(nx) * dx, np.arange(nt) * dt

nspike = nt // 8
refl = np.zeros(nt)
it = np.sort(np.random.permutation(range(10, nt - 20))[:nspike])
refl[it] = np.random.normal(0.0, 1.0, nspike)

# Wavelet
ntwav = 41
f0 = 30
twav = np.arange(ntwav) * dt
wav, *_ = pylops.utils.wavelets.ricker(twav, f0)

# Input trace
trace = np.convolve(refl, wav, mode="same")

# Slopes
theta = np.deg2rad(np.linspace(0, 30, nx))
slope = np.outer(np.ones(nt), np.tan(theta) * dt / dx)

# Model data
d = np.zeros((nt, nx))
tr = trace.copy()
for ix in range(nx):
    tr = _predict_trace(tr, t, dt, dx, slope[:, ix])
    d[:, ix] = tr

# Estimate slopes
slope_est, _ = pylops.utils.signalprocessing.slope_estimate(d, dt, dx, smooth=10)
slope_est *= -1
```

```
fig, axs = plt.subplots(2, 2, figsize=(6, 6), sharex=True, sharey=True)

opts = dict(aspect="auto", extent=(x[0], x[-1], t[-1], t[0]))
iax = axs[0, 0].imshow(d, cmap="gray", vmin=-1, vmax=1, **opts)
```

(continues on next page)

(continued from previous page)

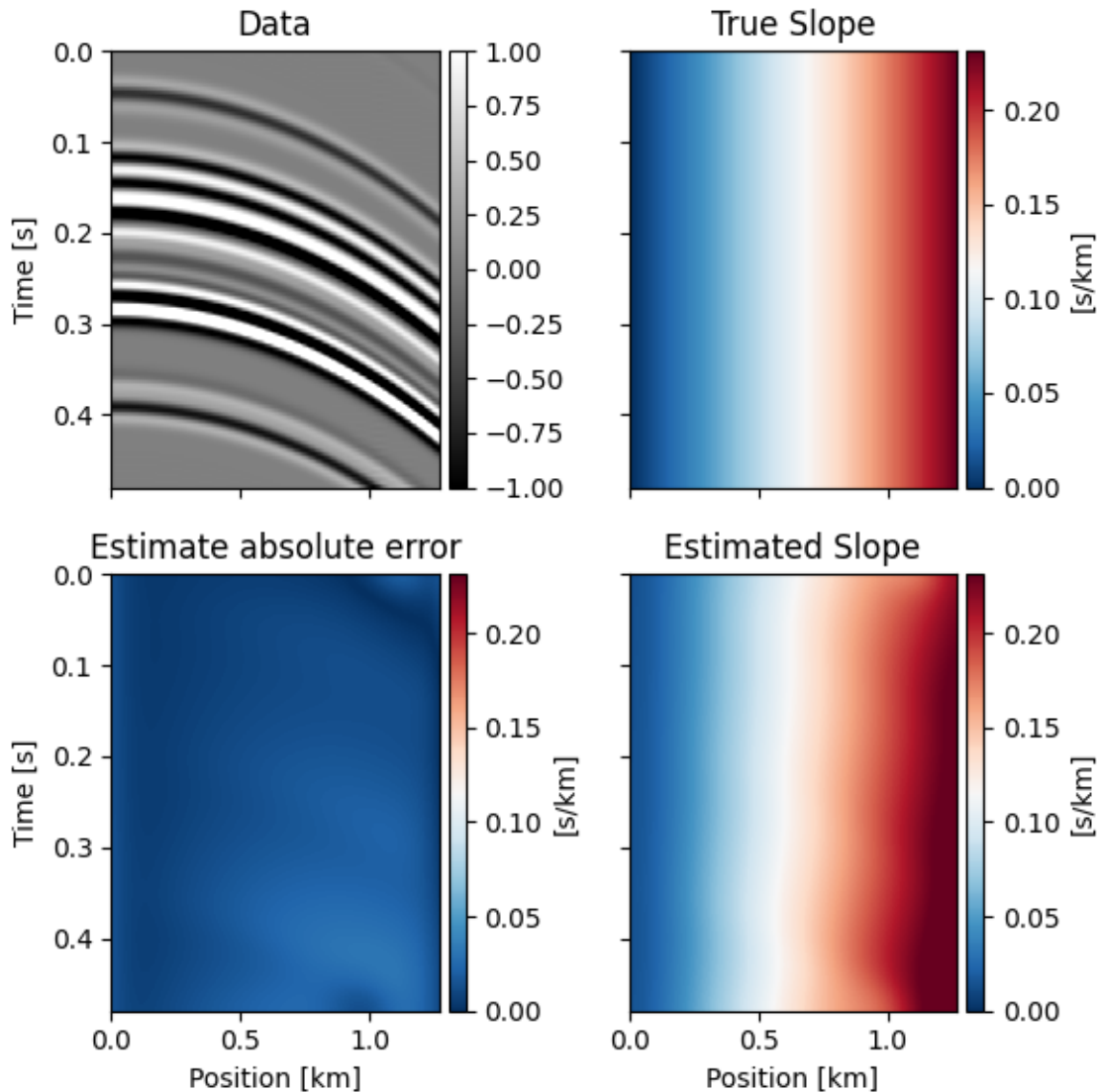
```
axs[0, 0].set(title="Data", ylabel="Time [s]")
cax = make_axes_locatable(axs[0, 0]).append_axes("right", size="5%", pad=0.05)
fig.colorbar(iax, cax=cax, orientation="vertical")

opts.update(dict(cmap="RdBu_r", vmin=np.min(slope), vmax=np.max(slope)))
iax = axs[0, 1].imshow(slope, **opts)
axs[0, 1].set(title="True Slope")
cax = make_axes_locatable(axs[0, 1]).append_axes("right", size="5%", pad=0.05)
fig.colorbar(iax, cax=cax, orientation="vertical")
cax.set_ylabel("[s/km]")

iax = axs[1, 0].imshow(np.abs(slope - slope_est), **opts)
axs[1, 0].set(
    title="Estimate absolute error", ylabel="Time [s]", xlabel="Position [km]"
)
cax = make_axes_locatable(axs[1, 0]).append_axes("right", size="5%", pad=0.05)
fig.colorbar(iax, cax=cax, orientation="vertical")
cax.set_ylabel("[s/km]")

iax = axs[1, 1].imshow(slope_est, **opts)
axs[1, 1].set(title="Estimated Slope", xlabel="Position [km]")
cax = make_axes_locatable(axs[1, 1]).append_axes("right", size="5%", pad=0.05)
fig.colorbar(iax, cax=cax, orientation="vertical")
cax.set_ylabel("[s/km]")

fig.tight_layout()
```



As you can see the Structure Tensor algorithm is a very fast, general purpose algorithm that can be used to estimate local slopes to input datasets of very different nature.

Total running time of the script: (0 minutes 1.368 seconds)

3.6.37 Spread How-to

This example focuses on the `pylops.basicoperators.Spread` operator, which is a highly versatile operator in PyLops to perform spreading/stacking operations in a vectorized manner (or efficiently via Numba-jitted for loops).

The `pylops.basicoperators.Spread` is powerful in its generality, but it may not be obvious for at first how to structure your code to leverage it properly. While it is highly recommended for advanced users to inspect the `pylops.signalprocessing.Radon2D` and `pylops.signalprocessing.Radon3D` operators since they are built using the `pylops.basicoperators.Spread` class, here we provide a simple example on how to get started.

In this example we will recreate a simplified version of the famous linear [Radon operator](#), which stacks data along straight lines with a given intercept and slope.

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

import pylops

plt.close("all")
```

Let's first define the time and space axes as well as some auxiliary input parameters that we will use to create a Ricker wavelet

```
par = {
    "ox": -200,
    "dx": 2,
    "nx": 201,
    "ot": 0,
    "dt": 0.004,
    "nt": 501,
    "f0": 20,
    "nfmax": 210,
}

# Create axis
t, _, x, _ = pylops.utils.seismicevents.makeaxis(par)

# Create centered Ricker wavelet
t_wav = np.arange(41) * par["dt"]
wav, _, _ = pylops.utils.wavelets.ricker(t_wav, f0=par["f0"])
```

We will create a 2d data with a number of crossing linear events, to which we will later apply our Radon transforms. We use the convenience function `pylops.utils.seismicevents.linear2d`.

```
v = 1500 # m/s
t0 = [0.2, 0.7, 1.6] # seconds
theta = [40, 0, -60] # degrees
amp = [1.0, 0.6, -2.0]

mlin, mlinwav = pylops.utils.seismicevents.linear2d(x, t, v, t0, theta, amp, wav)
```

Let's now define the slowness axis and use `pylops.signalprocessing.Radon2D` to implement our benchmark linear Radon. Refer to the documentation of the operator for a more detailed mathematical description of linear Radon. Note that `pxmax` is in s/m, which explains the small value. Its highest value corresponds to the lowest value of velocity in the transform. In this case we choose that to be 1000 m/s.

```
npx, pxmax = 41, 1e-3
px = np.linspace(-pxmax, pxmax, npx)

RLop = pylops.signalprocessing.Radon2D(
    t, x, px, centeredh=False, kind="linear", interp=False, engine="numpy"
)

# Compute adjoint = Radon transform
mlinwavR = RLop.H * mlinwav.ravel()
mlinwavR = mlinwavR.reshape(npx, par["nt"])
```

Now, let's try to reimplement this operator from scratch using `pylops.basicoperators.Spread`. Using the on-the-fly approach, and we need to create a function which takes indices of the model domain, here (p_x, t_0) where p_x is the slope and t_0 is the intercept of the parametric curve $t(x) = t_0 + p_x x$ we wish to spread the model over in the

data domain. The function must return an array of size n_x , containing the indices corresponding to $t(x)$.

The on-the-fly approach is useful when storing the indices in RAM may exhaust resources, especially when computing the indices is fast. When there is enough memory to store the full table of indices (an array of size $n_x \times n_t \times n_{p_x}$) the `pylops.basicoperators.Spread` operator can be used with tables instead. We will see an example of this later.

Returning to our on-the-fly example, we need to create a function which only depends on `ipx` and `it0`, so we create a closure around it with all our other auxiliary variables.

```
def create_radon_fh(xaxis, taxis, pxaxis):
    ot = taxis[0]
    dt = taxis[1] - taxis[0]
    nt = len(taxis)

    def fh(ipx, it0):
        tx = t[it0] + xaxis * pxaxis[ipx]
        it0_frac = (tx - ot) / dt
        itx = np rint(it0_frac)
        # Indices outside time axis set to nan
        itx[np.isin(itx, range(nt), invert=True)] = np.nan
        return itx

    return fh

fRad = create_radon_fh(x, t, px)
ROTFOp = pylops.Spread(npx, par["nt"], (par["nx"], par["nt"]), fh=fRad)

mlinwavROTf = ROTFOp.H * mlinwav.ravel()
mlinwavROTf = mlinwavROTf.reshape(npx, par["nt"])
```

Compare the results between the native Radon transform and the one using our on-the-fly `pylops.basicoperators.Spread`.

```
fig, axs = plt.subplots(1, 3, figsize=(9, 5), sharey=True)
axs[0].imshow(
    mlinwav.T,
    aspect="auto",
    interpolation="nearest",
    vmin=-1,
    vmax=1,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[0].set_title("Linear events", fontsize=12, fontweight="bold")
axs[0].set_xlabel(r"$x$ [m]")
axs[0].set_ylabel(r"$t$ [s]")
axs[1].imshow(
    mlinwavR.T,
    aspect="auto",
    interpolation="nearest",
    vmin=-10,
    vmax=10,
    cmap="gray",
    extent=(px.min(), px.max(), t.max(), t.min()),
)
axs[1].set_title("Native Linear Radon", fontsize=12, fontweight="bold")
```

(continues on next page)

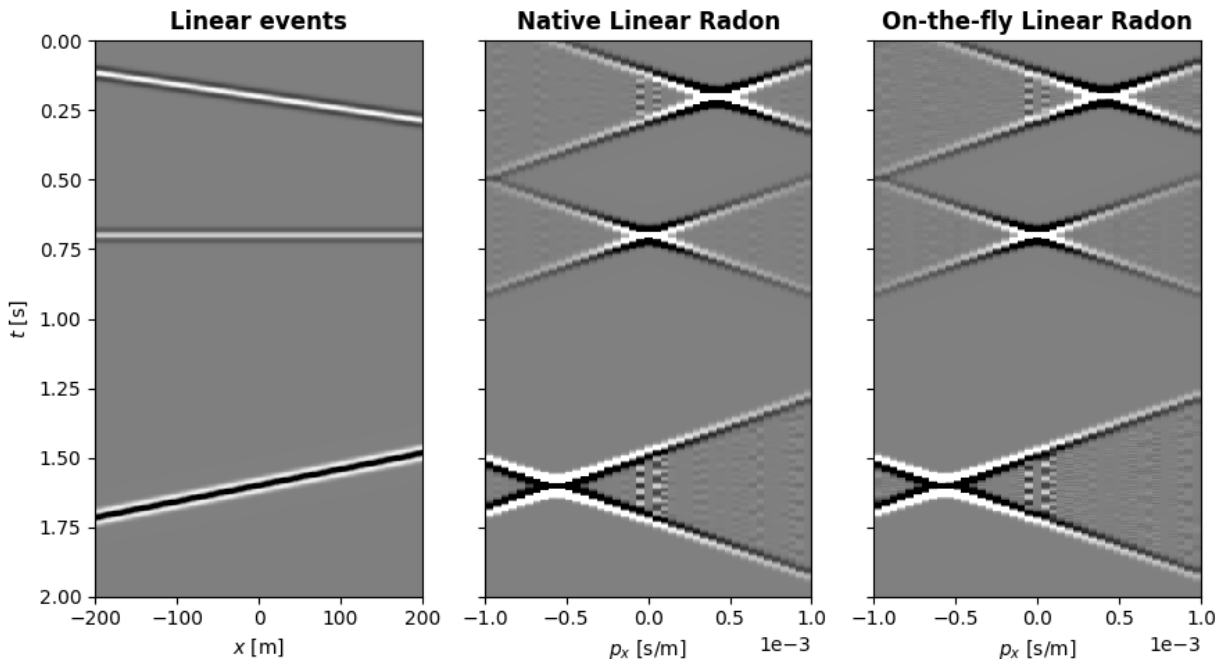
(continued from previous page)

```

axs[1].set_xlabel(r"$p_x$ [s/m]")
axs[1].ticklabel_format(style="sci", axis="x", scilimits=(0, 0))

axs[2].imshow(
    mlinwavROTF.T,
    aspect="auto",
    interpolation="nearest",
    vmin=-10,
    vmax=10,
    cmap="gray",
    extent=(px.min(), px.max(), t.max(), t.min()),
)
axs[2].set_title("On-the-fly Linear Radon", fontsize=12, fontweight="bold")
axs[2].set_xlabel(r"$p_x$ [s/m]")
axs[2].ticklabel_format(style="sci", axis="x", scilimits=(0, 0))
fig.tight_layout()

```



Finally, we will re-implement the example above using pre-computed tables. This is useful when `fh` is expensive to compute, or requires manual edition prior to usage.

Using a table instead of a function is simple, we just need to apply `fh` to all our points and store the results.

```

def create_table(npx, nt, nx):
    table = np.full((npx, nt, nx), fill_value=np.nan)
    for ipx in range(npx):
        for it0 in range(nt):
            table[ipx, it0, :] = fRad(ipx, it0)
    return table

table = create_table(npx, par["nt"], par["nx"])
RPCOp = pylops.Spread((npx, par["nt"]), (par["nx"], par["nt"]), table=table)

```

(continues on next page)

(continued from previous page)

```

mlinwavRPC = RPCOp.H * mlinwav.ravel()
mlinwavRPC = mlinwavRPC.reshape(npix, par["nt"])

```

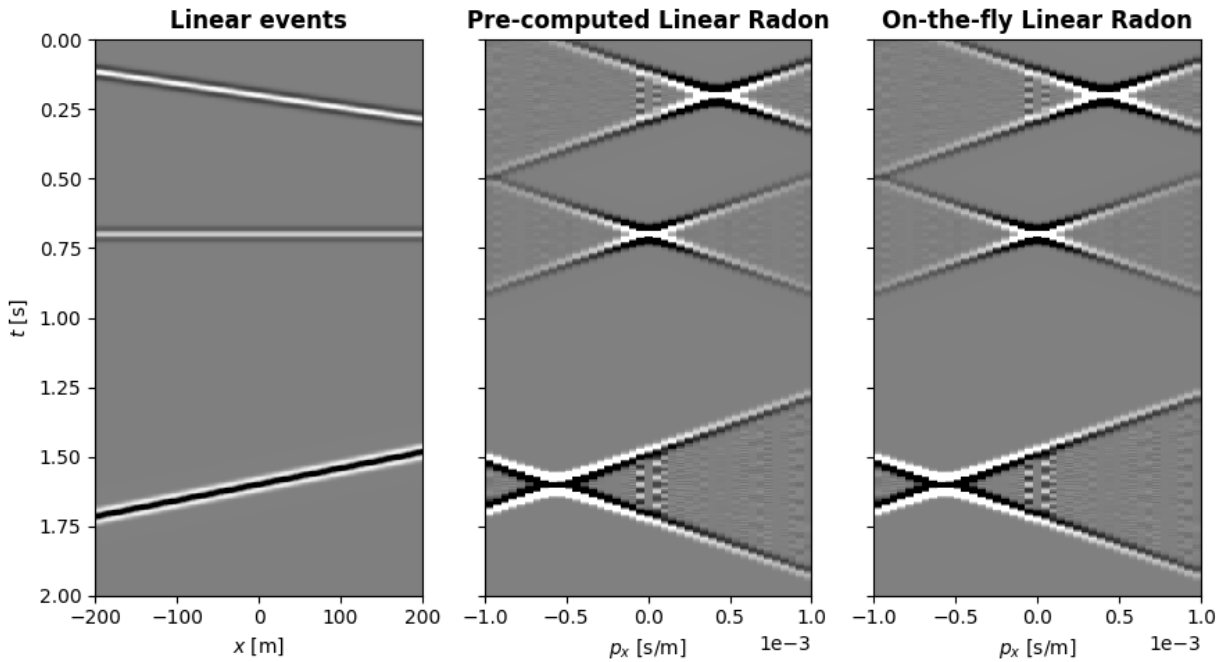
Compare the results between the pre-computed or on-the-fly Radon transforms

```

fig, axs = plt.subplots(1, 3, figsize=(9, 5), sharey=True)
axs[0].imshow(
    mlinwav.T,
    aspect="auto",
    interpolation="nearest",
    vmin=-1,
    vmax=1,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[0].set_title("Linear events", fontsize=12, fontweight="bold")
axs[0].set_xlabel(r"$x$ [m]")
axs[0].set_ylabel(r"$t$ [s]")
axs[1].imshow(
    mlinwavRPC.T,
    aspect="auto",
    interpolation="nearest",
    vmin=-10,
    vmax=10,
    cmap="gray",
    extent=(px.min(), px.max(), t.max(), t.min()),
)
axs[1].set_title("Pre-computed Linear Radon", fontsize=12, fontweight="bold")
axs[1].set_xlabel(r"$p_x$ [s/m]")
axs[1].ticklabel_format(style="sci", axis="x", scilimits=(0, 0))

axs[2].imshow(
    mlinwavROTF.T,
    aspect="auto",
    interpolation="nearest",
    vmin=-10,
    vmax=10,
    cmap="gray",
    extent=(px.min(), px.max(), t.max(), t.min()),
)
axs[2].set_title("On-the-fly Linear Radon", fontsize=12, fontweight="bold")
axs[2].set_xlabel(r"$p_x$ [s/m]")
axs[2].ticklabel_format(style="sci", axis="x", scilimits=(0, 0))
fig.tight_layout()

```



Total running time of the script: (0 minutes 9.345 seconds)

3.6.38 Sum

This example shows how to use the `pylops.Sum` operator to stack values along an axis of a multi-dimensional array

```
import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt
import numpy as np

import pylops

plt.close("all")
```

Let's start by defining a 2-dimensional data

```
ny, nx = 5, 7
x = (np.arange(ny * nx)).reshape(ny, nx)
```

We can now create the operator and perform forward and adjoint

```
Sop = pylops.Sum(dims=(ny, nx), dir=0)

y = Sop * x.ravel()
xadj = Sop.H * y
xadj = xadj.reshape(ny, nx)

gs = pltgs.GridSpec(1, 7)
fig = plt.figure(figsize=(7, 3))
ax = plt.subplot(gs[0, 0:3])
im = ax.imshow(x, cmap="rainbow", vmin=0, vmax=ny * nx)
ax.set_title("x", size=20, fontweight="bold")
```

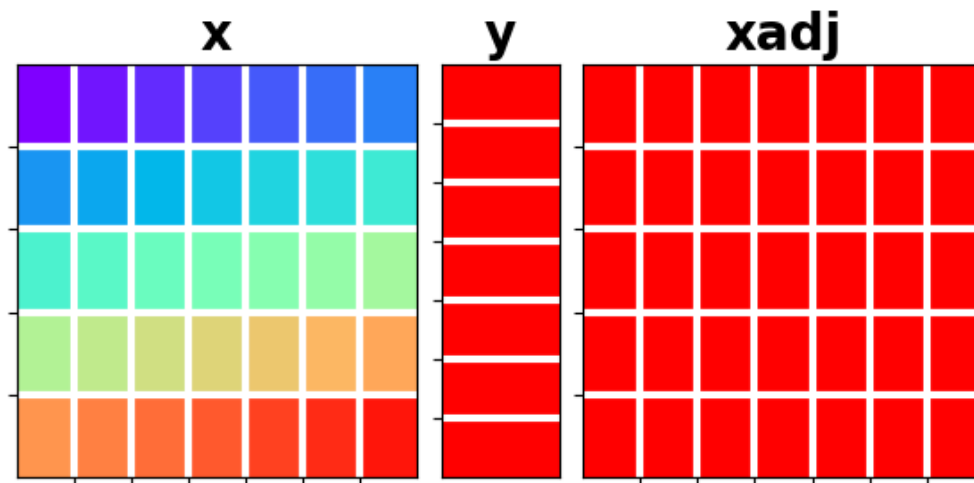
(continues on next page)

(continued from previous page)

```

ax.set_xticks(np.arange(nx - 1) + 0.5)
ax.set_yticks(np.arange(ny - 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, 3])
ax.imshow(y[:, np.newaxis], cmap="rainbow", vmin=0, vmax=ny * nx)
ax.set_title("y", size=20, fontweight="bold")
ax.set_xticks([])
ax.set_yticks(np.arange(nx - 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, 4:])
ax.imshow(xadj, cmap="rainbow", vmin=0, vmax=ny * nx)
ax.set_title("xadj", size=20, fontweight="bold")
ax.set_xticks(np.arange(nx - 1) + 0.5)
ax.set_yticks(np.arange(ny - 1) + 0.5)
ax.grid(linewidth=3, color="white")
ax.xaxis.set_ticklabels([])
ax.yaxis.set_ticklabels([])
ax.axis("tight")

```



Out:

```
(-0.5, 6.5, 4.5, -0.5)
```

Note that since the Sum operator creates an under-determined system of equations (data has always lower dimensionality than the model), an exact inverse is not possible for this operator.

Total running time of the script: (0 minutes 0.137 seconds)

3.6.39 Symmetrize

This example shows how to use the `pylops.Symmetrize` operator which takes an input signal and returns a symmetric signal by pre-pending the input signal in reversed order. Such an operation can be inverted as we will see in this example.

Moreover the `pylops.Symmetrize` can be used as *preconditioning* to any inverse problem where we are after inverting for a signal that we want to ensure is symmetric. Refer to [Wavelet estimation](#) for an example of such a type.

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

import pylops

plt.close("all")
```

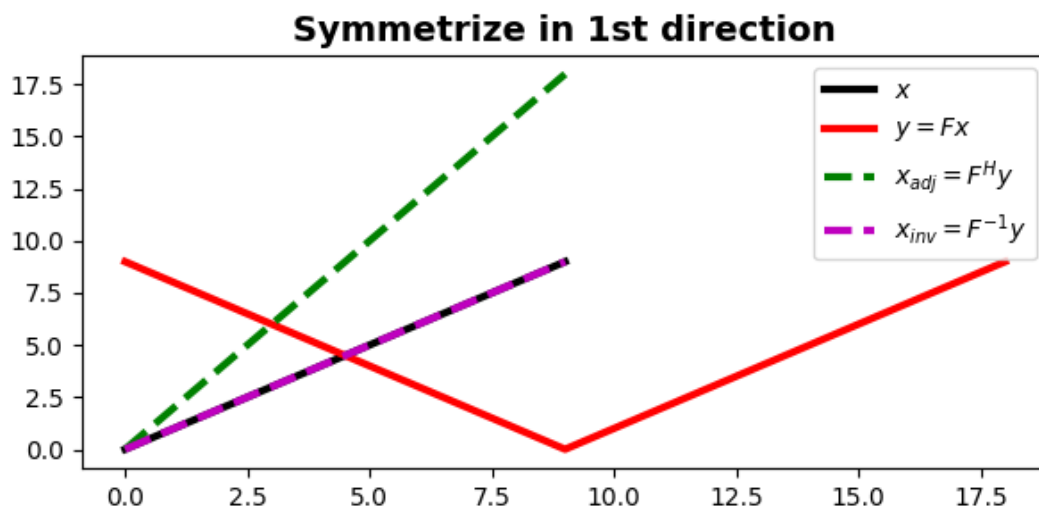
Let's start with a 1D example. Define an input signal composed of `nt` samples

```
nt = 10
x = np.arange(nt)
```

We can now create our flip operator and apply it to the input signal. We can also apply the adjoint to the flipped signal and we can see how for this operator the adjoint is effectively equivalent to the inverse.

```
Sop = pylops.Symmetrize(nt)
y = Sop * x
xadj = Sop.H * y
xinv = Sop / y

plt.figure(figsize=(7, 3))
plt.plot(x, "k", lw=3, label=r"$x$")
plt.plot(y, "r", lw=3, label=r"$y=Fx$")
plt.plot(xadj, "--g", lw=3, label=r"$x_{adj} = F^H y$")
plt.plot(xinv, "--m", lw=3, label=r"$x_{inv} = F^{-1} y$")
plt.title("Symmetrize in 1st direction", fontsize=14, fontweight="bold")
plt.legend()
```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aecdbccc0>
```

Let's now repeat the same exercise on a two dimensional signal. We will first flip the model along the first axis and then along the second axis

```
nt, nx = 10, 6
x = np.outer(np.arange(nt), np.ones(nx))

Sop = pylops.Symmetrize(nt * nx, dims=(nt, nx), dir=0)
y = Sop * x.ravel()
xadj = Sop.H * y.ravel()
xinv = Sop / y
y = y.reshape(2 * nt - 1, nx)
xadj = xadj.reshape(nt, nx)
xinv = xinv.reshape(nt, nx)

fig, axs = plt.subplots(1, 3, figsize=(7, 3))
fig.suptitle(
    "Symmetrize in 2nd direction for 2d data", fontsize=14, fontweight="bold", y=0.95
)
axs[0].imshow(x, cmap="rainbow", vmin=0, vmax=9)
axs[0].set_title(r"$x$")
axs[0].axis("tight")
axs[1].imshow(y, cmap="rainbow", vmin=0, vmax=9)
axs[1].set_title(r"$y=Fx$")
axs[1].axis("tight")
axs[2].imshow(xinv, cmap="rainbow", vmin=0, vmax=9)
axs[2].set_title(r"$x_{\text{adj}}=F^{-1}y$")
axs[2].axis("tight")
plt.tight_layout()
plt.subplots_adjust(top=0.8)

x = np.outer(np.ones(nt), np.arange(nx))
Sop = pylops.Symmetrize(nt * nx, dims=(nt, nx), dir=1)

y = Sop * x.ravel()
xadj = Sop.H * y.ravel()
xinv = Sop / y
y = y.reshape(nt, 2 * nx - 1)
xadj = xadj.reshape(nt, nx)
xinv = xinv.reshape(nt, nx)

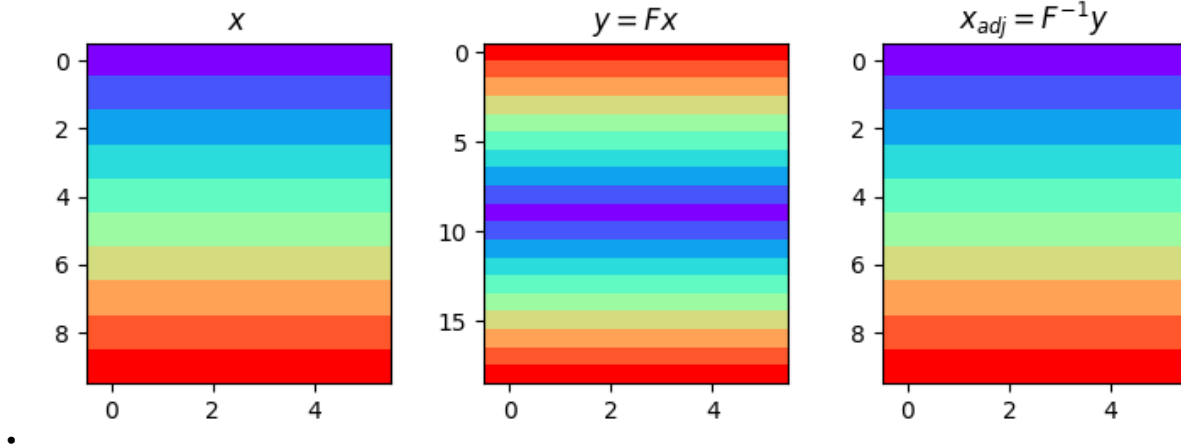
# sphinx_gallery_thumbnail_number = 3
fig, axs = plt.subplots(1, 3, figsize=(7, 3))
fig.suptitle(
    "Symmetrize in 2nd direction for 2d data", fontsize=14, fontweight="bold", y=0.95
)
axs[0].imshow(x, cmap="rainbow", vmin=0, vmax=9)
axs[0].set_title(r"$x$")
axs[0].axis("tight")
axs[1].imshow(y, cmap="rainbow", vmin=0, vmax=9)
axs[1].set_title(r"$y=Fx$")
axs[1].axis("tight")
axs[2].imshow(xinv, cmap="rainbow", vmin=0, vmax=9)
axs[2].set_title(r"$x_{\text{adj}}=F^{-1}y$")
axs[2].axis("tight")
```

(continues on next page)

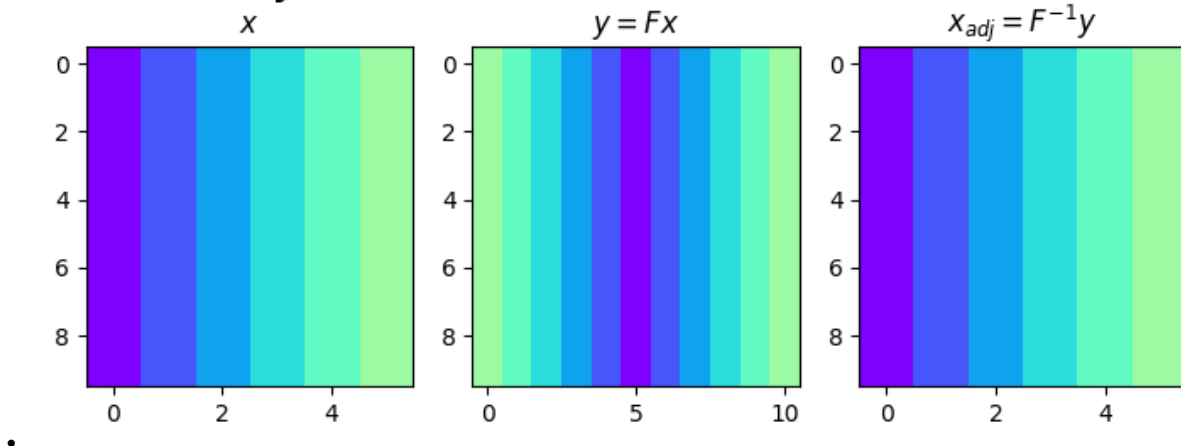
(continued from previous page)

```
plt.tight_layout()
plt.subplots_adjust(top=0.8)
```

Symmetrize in 2nd direction for 2d data



Symmetrize in 2nd direction for 2d data



Total running time of the script: (0 minutes 0.737 seconds)

3.6.40 Synthetic seismic

This example shows how to use the `pylops.utils.seismicevents` module to quickly create synthetic seismic data to be used for toy examples and tests.

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

import pylops

plt.close("all")
```

Let's first define the time and space axes as well as some auxiliary input parameters that we will use to create a Ricker wavelet

```

par = {
    "ox": -200,
    "dx": 2,
    "nx": 201,
    "oy": -100,
    "dy": 2,
    "ny": 101,
    "ot": 0,
    "dt": 0.004,
    "nt": 501,
    "f0": 20,
    "nfmax": 210,
}

# Create axis
t, t2, x, y = pyllops.utils.seismicevents.makeaxis(par)

# Create wavelet
wav = pyllops.utils.wavelets.ricker(np.arange(41) * par["dt"], f0=par["f0"])[0]

```

We want to create a 2d data with a number of crossing linear events using the `pyllops.utils.seismicevents.linear2d` routine.

```

v = 1500
t0 = [0.2, 0.7, 1.6]
theta = [40, 0, -60]
amp = [1.0, 0.6, -2.0]

mlin, mlinwav = pyllops.utils.seismicevents.linear2d(x, t, v, t0, theta, amp, wav)

```

We can also create a 2d data with a number of crossing parabolic events using the `pyllops.utils.seismicevents.parabolic2d` routine.

```

px = [0, 0, 0]
pxx = [1e-5, 5e-6, 1e-6]

mpar, mparwav = pyllops.utils.seismicevents.parabolic2d(x, t, t0, px, pxx, amp, wav)

```

And similarly we can create a 2d data with a number of crossing hyperbolic events using the `pyllops.utils.seismicevents.hyperbolic2d` routine.

```

vrms = [500, 700, 1700]

mhyp, mhypwav = pyllops.utils.seismicevents.hyperbolic2d(x, t, t0, vrms, amp, wav)

```

We can now visualize the different events

```

# sphinx_gallery_thumbnail_number = 2
fig, axs = plt.subplots(1, 3, figsize=(9, 5))
axs[0].imshow(
    mlinwav.T,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),

```

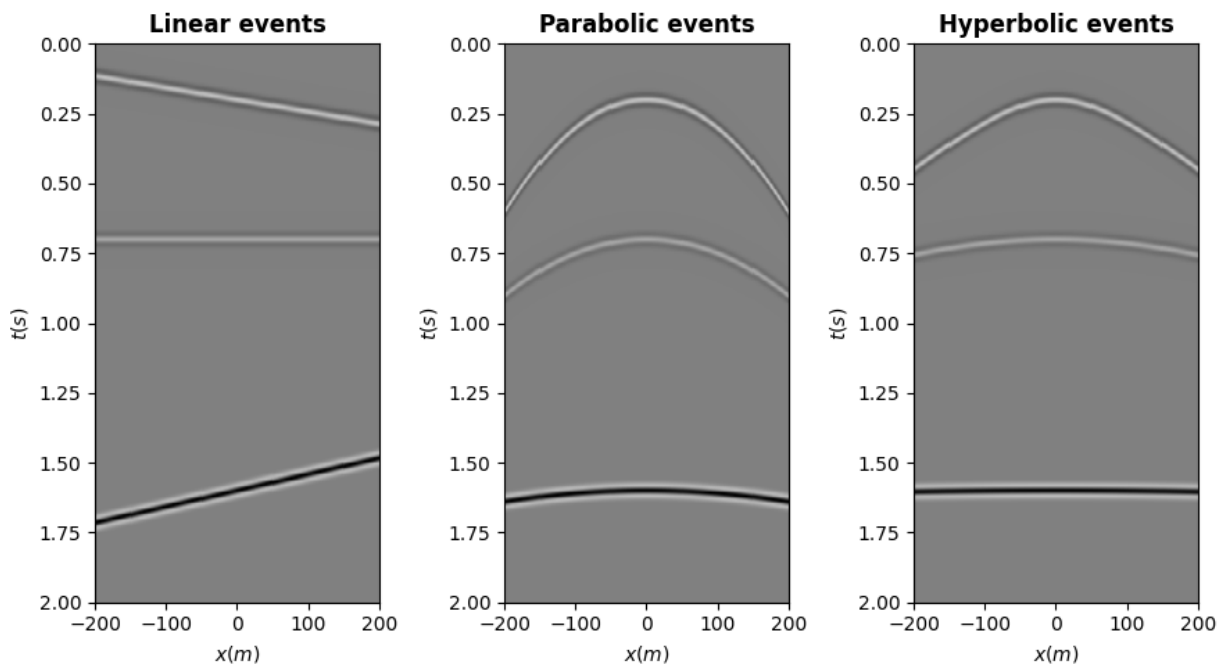
(continues on next page)

(continued from previous page)

```

)
axs[0].set_title("Linear events", fontsize=12, fontweight="bold")
axs[0].set_xlabel(r"$x(m)$")
axs[0].set_ylabel(r"$t(s)$")
axs[1].imshow(
    mparwav.T,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[1].set_title("Parabolic events", fontsize=12, fontweight="bold")
axs[1].set_xlabel(r"$x(m)$")
axs[1].set_ylabel(r"$t(s)$")
axs[2].imshow(
    mhypwav.T,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[2].set_title("Hyperbolic events", fontsize=12, fontweight="bold")
axs[2].set_xlabel(r"$x(m)$")
axs[2].set_ylabel(r"$t(s)$")
plt.tight_layout()

```



Let's finally repeat the same exercise in 3d

```

phi = [20, 0, -10]

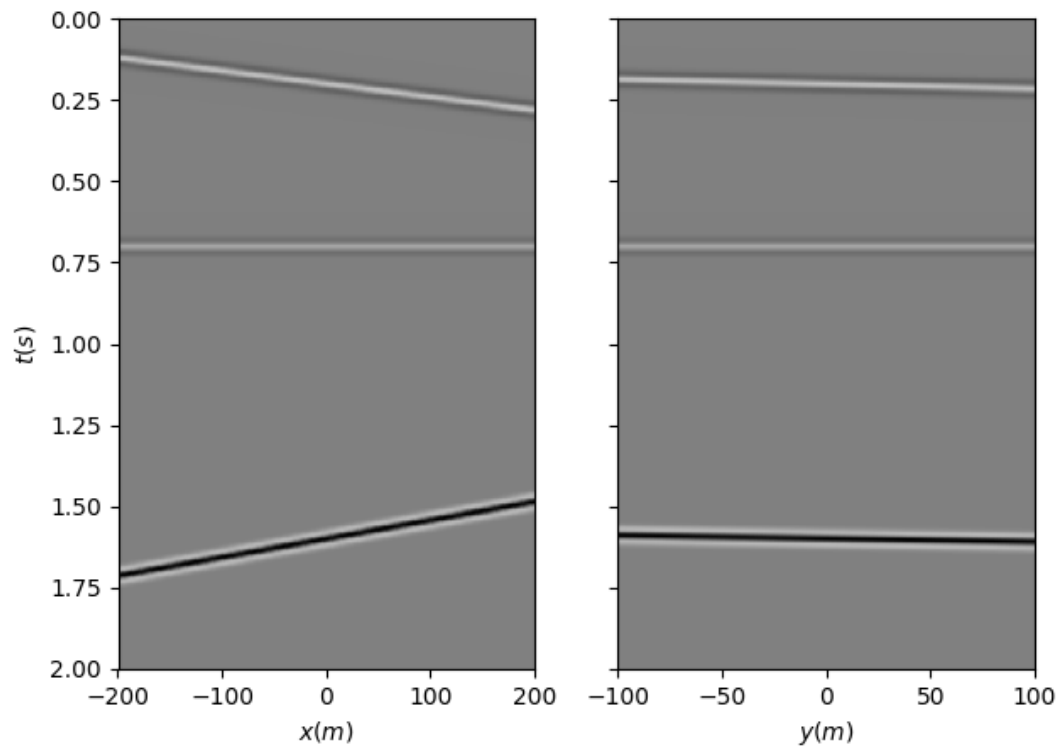
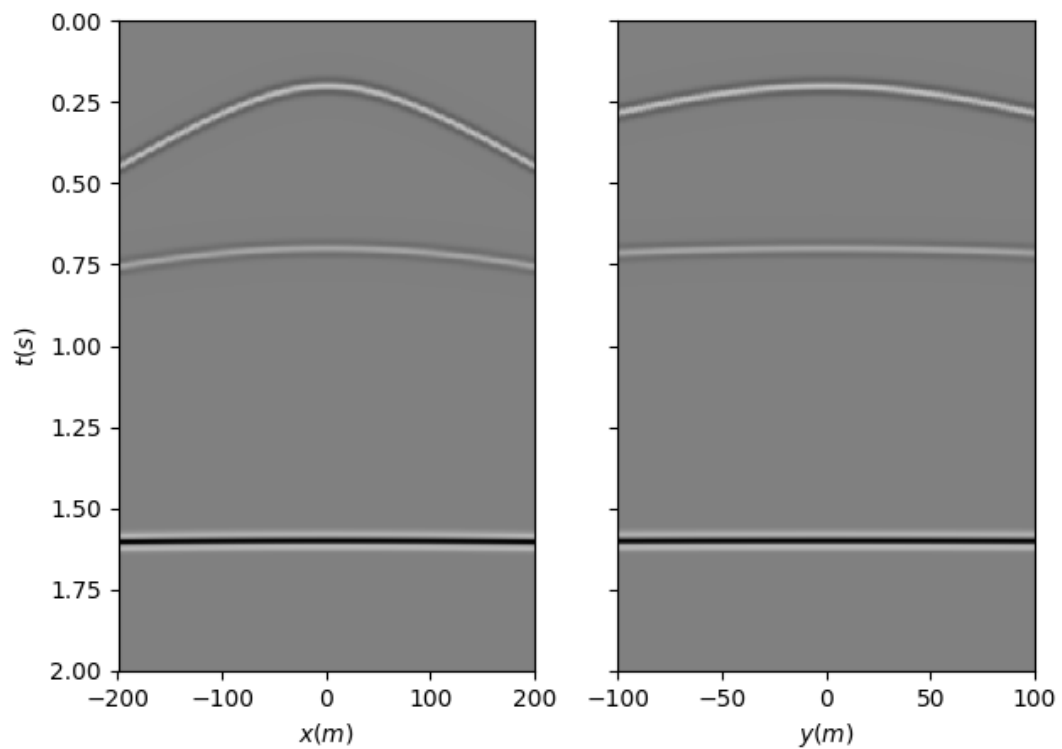
mlin, mlinwav = pyllops.utils.seismicevents.linear3d(
    x, y, t, v, t0, theta, phi, amp, wav
)

fig, axs = plt.subplots(1, 2, figsize=(7, 5), sharey=True)
fig.suptitle("Linear events in 3d", fontsize=12, fontweight="bold", y=0.95)
axs[0].imshow(
    mlinwav[par["ny"] // 2].T,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[0].set_xlabel(r"$x(m)$")
axs[0].set_ylabel(r"$t(s)$")
axs[1].imshow(
    mlinwav[:, par["nx"] // 2].T,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(y.min(), y.max(), t.max(), t.min()),
)
axs[1].set_xlabel(r"$y(m)$")

mhyp, mhypwav = pyllops.utils.seismicevents.hyperbolic3d(
    x, y, t, t0, vrms, vrms, amp, wav
)

fig, axs = plt.subplots(1, 2, figsize=(7, 5), sharey=True)
fig.suptitle("Hyperbolic events in 3d", fontsize=12, fontweight="bold", y=0.95)
axs[0].imshow(
    mhypwav[par["ny"] // 2].T,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(x.min(), x.max(), t.max(), t.min()),
)
axs[0].set_xlabel(r"$x(m)$")
axs[0].set_ylabel(r"$t(s)$")
axs[1].imshow(
    mhypwav[:, par["nx"] // 2].T,
    aspect="auto",
    interpolation="nearest",
    vmin=-2,
    vmax=2,
    cmap="gray",
    extent=(y.min(), y.max(), t.max(), t.min()),
)
axs[1].set_xlabel(r"$y(m)$")

```

Linear events in 3d**Hyperbolic events in 3d**

Out:

```
Text (0.5, 25.72222222222214, '$y(m)$')
```

Total running time of the script: (0 minutes 2.408 seconds)

3.6.41 Tapers

This example shows how to create some basic tapers in 1d, 2d, and 3d using the `pylops.utils.tapers` module.

```
import matplotlib.pyplot as plt

import pylops

plt.close("all")
```

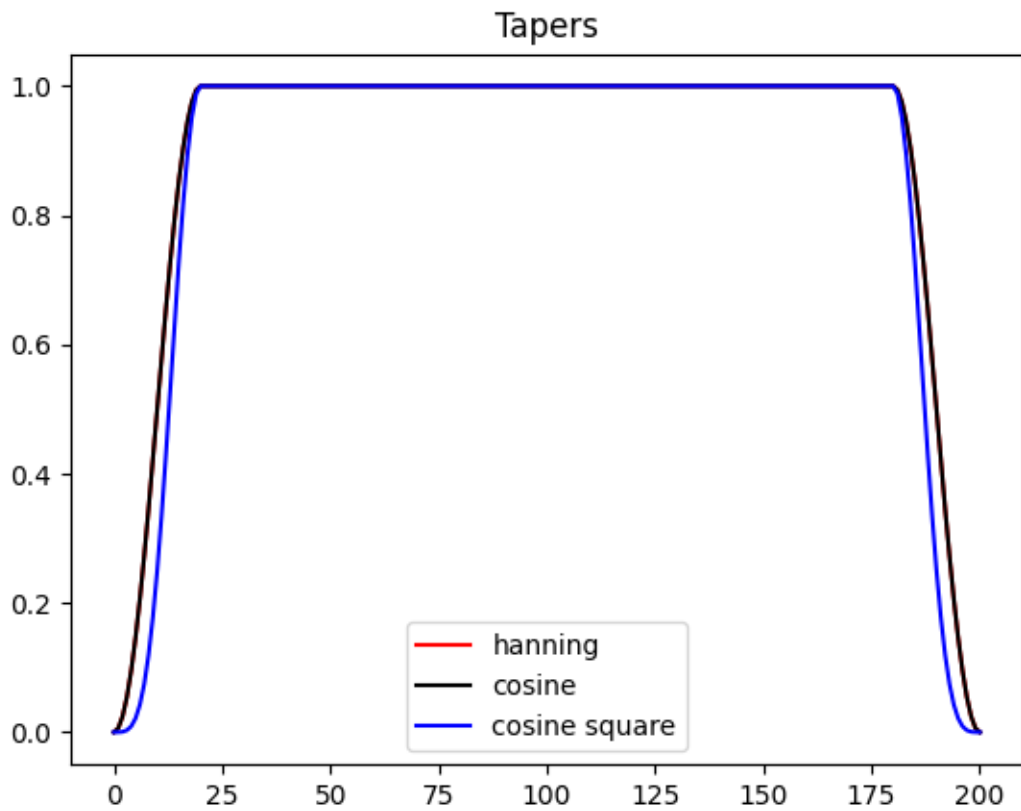
Let's first define the time and space axes

```
par = {
    "ox": -200,
    "dx": 2,
    "nx": 201,
    "oy": -100,
    "dy": 2,
    "ny": 101,
    "ot": 0,
    "dt": 0.004,
    "nt": 501,
    "ntapx": 21,
    "ntapy": 31,
}
```

We can now create tapers in 1d

```
tap_han = pylops.utils.tapers.hanningtaper(par["nx"], par["ntapx"])
tap_cos = pylops.utils.tapers.cosinetaper(par["nx"], par["ntapx"], False)
tap_cos2 = pylops.utils.tapers.cosinetaper(par["nx"], par["ntapx"], True)

plt.figure()
plt.plot(tap_han, "r", label="hanning")
plt.plot(tap_cos, "k", label="cosine")
plt.plot(tap_cos2, "b", label="cosine square")
plt.title("Tapers")
plt.legend()
```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aed6abb38>
```

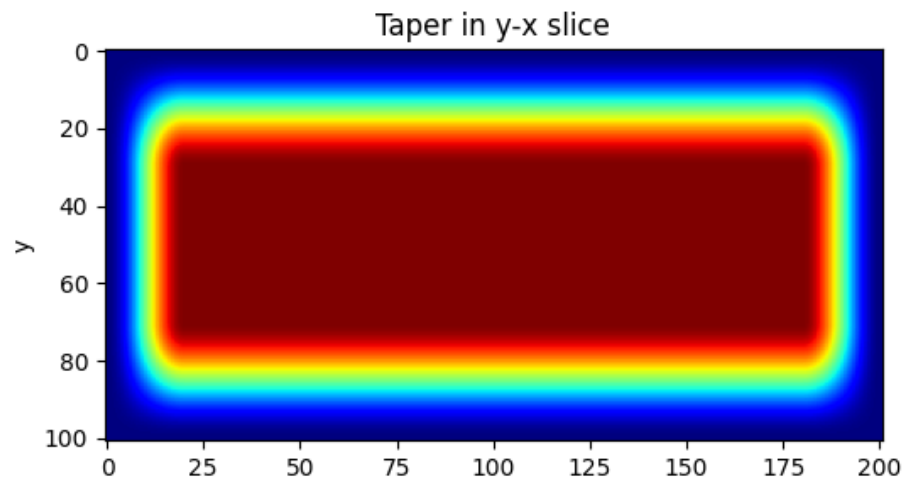
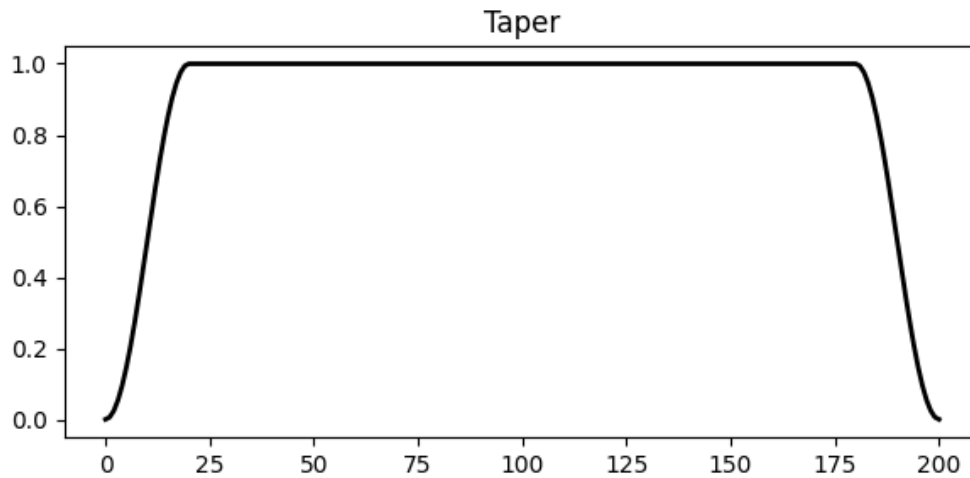
Similarly we can create 2d and 3d tapers with any of the tapers above

```
tap2d = pyllops.utils.tapers.taper2d(par["nt"], par["nx"], par["ntapx"])

plt.figure(figsize=(7, 3))
plt.plot(tap2d[:, par["nt"] // 2], "k", lw=2)
plt.title("Taper")

tap3d = pyllops.utils.tapers.taper3d(
    par["nt"], (par["ny"], par["nx"]), (par["ntapy"], par["ntapx"])
)

plt.figure(figsize=(7, 3))
plt.imshow(tap3d[:, :, par["nt"] // 2], "jet")
plt.title("Taper in y-x slice")
plt.xlabel("x")
plt.ylabel("y")
```



Out:

```
Text (87.24078657865785, 0.5, 'y')
```

Total running time of the script: (0 minutes 0.406 seconds)

3.6.42 Total Variation (TV) Regularization

This set of examples shows how to add Total Variation (TV) regularization to an inverse problem in order to enforce blockiness in the reconstructed model.

To do so we will use the generalized Split Bregman iterations by means of `pylops.optimization.sparsity.SplitBregman` solver.

The first example is concerned with denoising of a piece-wise step function which has been contaminated by noise. The forward model is:

$$\mathbf{y} = \mathbf{x} + \mathbf{n}$$

meaning that we have an identity operator (\mathbf{I}) and inverting for \mathbf{x} from \mathbf{y} is impossible without adding prior information. We will enforce blockiness in the solution by adding a regularization term that enforces sparsity in the first

derivative of the solution:

$$J = \mu/2 \|\mathbf{y} - \mathbf{I}\mathbf{x}\|_2 + \|\nabla \mathbf{x}\|_1$$

```
import matplotlib.pyplot as plt

# sphinx_gallery_thumbnail_number = 5
import numpy as np

import pylops

plt.close("all")
np.random.seed(1)
```

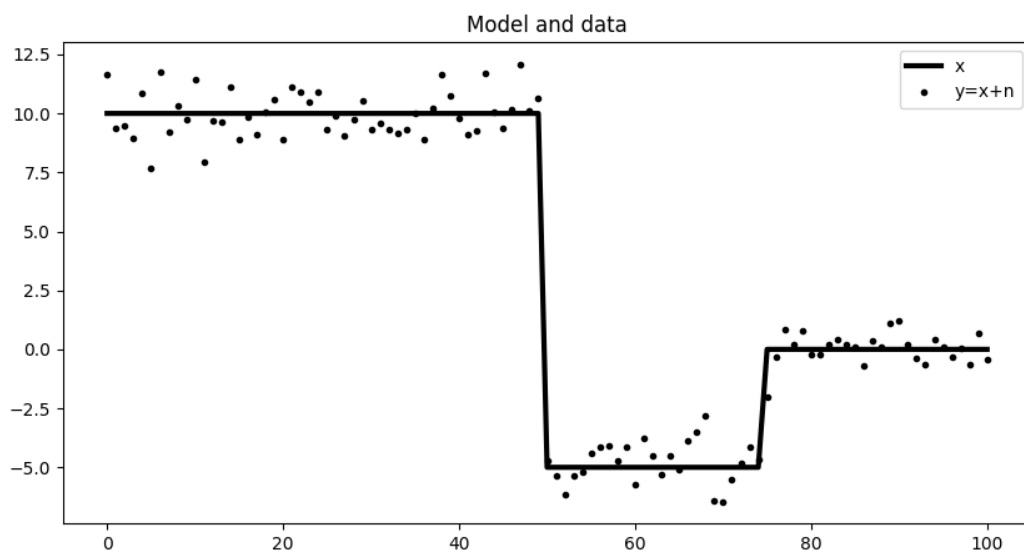
Let's start by creating the model and data

```
nx = 101
x = np.zeros(nx)
x[: nx // 2] = 10
x[nx // 2 : 3 * nx // 4] = -5

Iop = pylops.Identity(nx)

n = np.random.normal(0, 1, nx)
y = Iop * (x + n)

plt.figure(figsize=(10, 5))
plt.plot(x, "k", lw=3, label="x")
plt.plot(y, ".k", label="y=x+n")
plt.legend()
plt.title("Model and data")
```



Out:

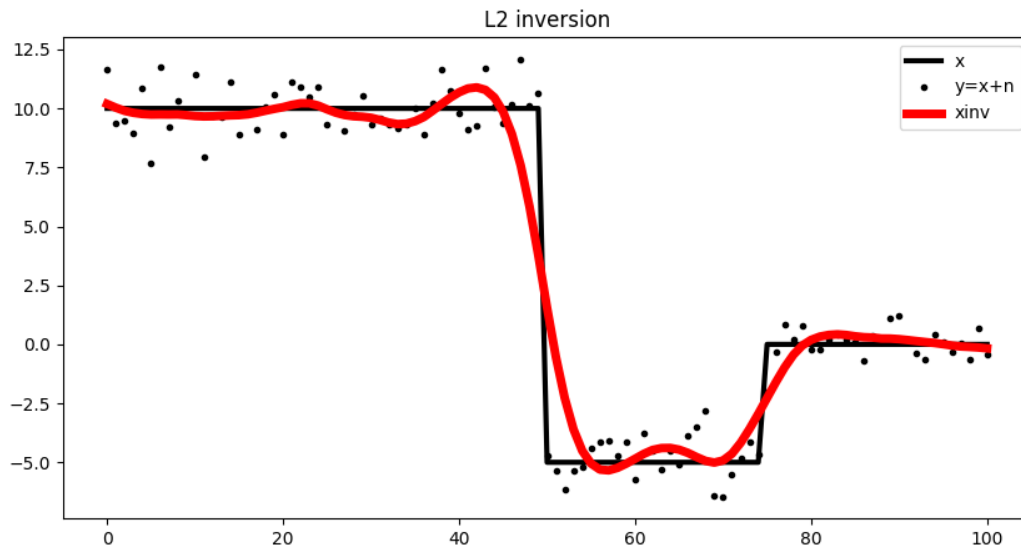
```
Text(0.5, 1.0, 'Model and data')
```

To start we will try to use a simple L2 regularization that enforces smoothness in the solution. We can see how denoising is successfully achieved but the solution is much smoother than we wish for.

```
D2op = pylops.SecondDerivative(nx, edge=True)
lamda = 1e2

xinv = pylops.optimization.leastsquares.RegularizedInversion(
    Iop, [D2op], y, epsRs=[np.sqrt(lamda / 2)], **dict(iter_lim=30)
)

plt.figure(figsize=(10, 5))
plt.plot(x, "k", lw=3, label="x")
plt.plot(y, ".k", label="y=x+n")
plt.plot(xinv, "r", lw=5, label="xinv")
plt.legend()
plt.title("L2 inversion")
```



Out:

```
Text(0.5, 1.0, 'L2 inversion')
```

Now we impose blockiness in the solution using the Split Bregman solver

```
Dop = pylops.FirstDerivative(nx, edge=True, kind="backward")
mu = 0.01
lamda = 0.3
niter_out = 50
niter_in = 3

xinv, niter = pylops.optimization.sparsity.SplitBregman(
    Iop,
    [Dop],
    y,
```

(continues on next page)

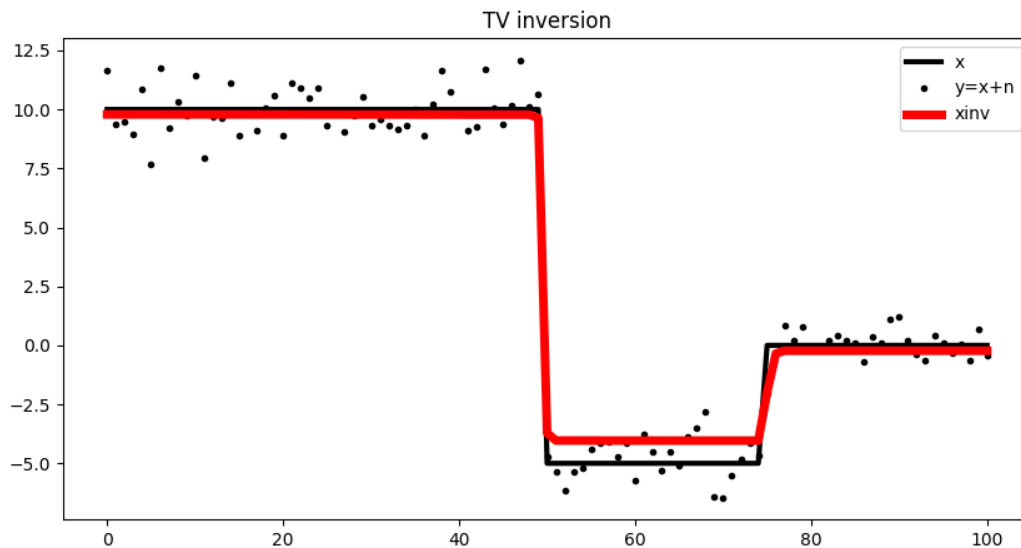
(continued from previous page)

```

niter_out,
niter_in,
mu=mu,
epsRLls=[lamda],
tol=1e-4,
tau=1.0,
**dict(iter_lim=30, damp=1e-10)
)

plt.figure(figsize=(10, 5))
plt.plot(x, "k", lw=3, label="x")
plt.plot(y, ".k", label="y=x+n")
plt.plot(xinv, "r", lw=5, label="xinv")
plt.legend()
plt.title("TV inversion")

```



Out:

```
Text(0.5, 1.0, 'TV inversion')
```

Finally, we repeat the same exercise on a 2-dimensional image. In this case we mock a medical imaging problem: the data is created by applying a 2D Fourier Transform to the input model and by randomly sampling 60% of its values.

```

x = np.load("../testdata/optimization/shepp_logan_phantom.npy")
x = x / x.max()
ny, nx = x.shape

perc_subsampling = 0.6
nxsub = int(np.round(ny * nx * perc_subsampling))
iava = np.sort(np.random.permutation(np.arange(ny * nx))[:nxsub])
Rop = pylops.Restriction(ny * nx, iava, dtype=np.complex128)
Fop = pylops.signalprocessing.FFT2D(dims=(ny, nx))

n = np.random.normal(0, 0.0, (ny, nx))

```

(continues on next page)

(continued from previous page)

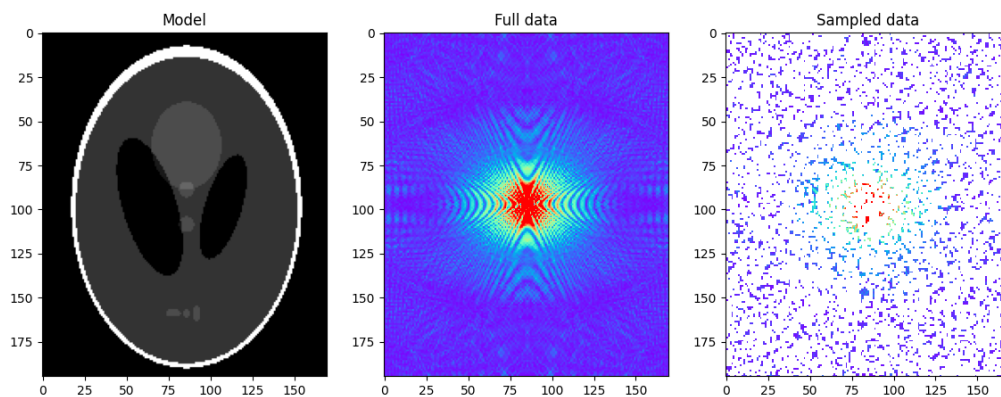
```

y = Rop * Fop * (x.ravel() + n.ravel())
yfft = Fop * (x.ravel() + n.ravel())
yfft = np.fft.fftshift(yfft.reshape(ny, nx))

ymask = Rop.mask(Fop * (x.ravel() + n.ravel()))
ymask = ymask.reshape(ny, nx)
ymask.data[:] = np.fft.fftshift(ymask.data)
ymask.mask[:] = np.fft.fftshift(ymask.mask)

fig, axs = plt.subplots(1, 3, figsize=(14, 5))
axs[0].imshow(x, vmin=0, vmax=1, cmap="gray")
axs[0].set_title("Model")
axs[0].axis("tight")
axs[1].imshow(np.abs(yfft), vmin=0, vmax=1, cmap="rainbow")
axs[1].set_title("Full data")
axs[1].axis("tight")
axs[2].imshow(np.abs(ymask), vmin=0, vmax=1, cmap="rainbow")
axs[2].set_title("Sampled data")
axs[2].axis("tight")

```



Out:

```
(-0.5, 169.5, 194.5, -0.5)
```

Let's attempt now to reconstruct the model using the Split Bregman with anisotropic TV regularization (aka sum of L1 norms of the first derivatives over x and y):

$$J = \mu/2 \|y - \mathbf{R}\mathbf{F}\mathbf{x}\|_2 + \|\nabla_x \mathbf{x}\|_1 + \|\nabla_y \mathbf{x}\|_1$$

```

Dop = [
    pylops.FirstDerivative(
        ny * nx, dims=(ny, nx), dir=0, edge=False, kind="backward", dtype=np.
        ↪complex128
    ),
    pylops.FirstDerivative(
        ny * nx, dims=(ny, nx), dir=1, edge=False, kind="backward", dtype=np.
        ↪complex128
    ),
]

```

(continues on next page)

(continued from previous page)

```

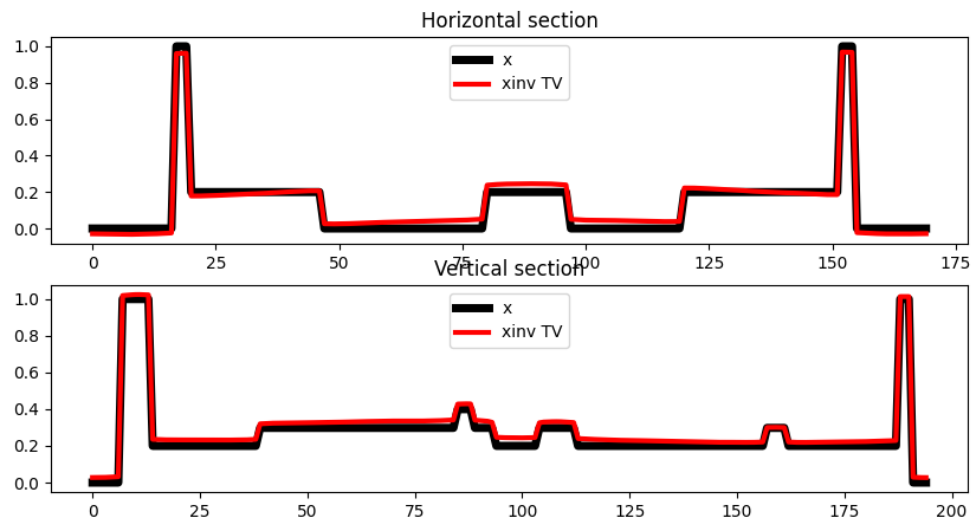
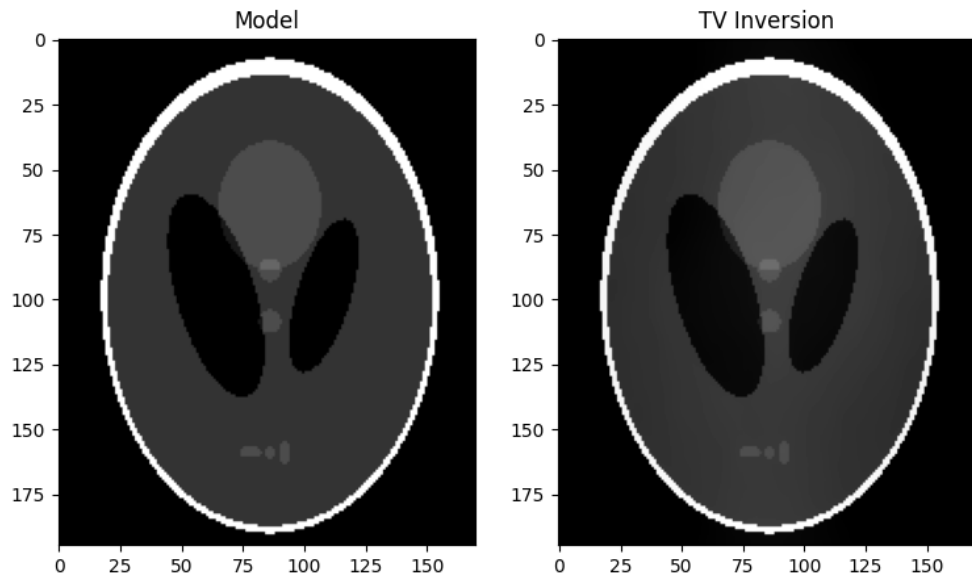
# TV
mu = 1.5
lamda = [0.1, 0.1]
niter = 20
niterinner = 10

xinv, niter = pyllops.optimization.sparsity.SplitBregman(
    Rop * Fop,
    Dop,
    y.ravel(),
    niter,
    niterinner,
    mu=mu,
    epsRLls=lamda,
    tol=1e-4,
    tau=1.0,
    show=False,
    **dict(iter_lim=5, damp=1e-4)
)
xinv = np.real(xinv.reshape(ny, nx))

fig, axs = plt.subplots(1, 2, figsize=(9, 5))
axs[0].imshow(x, vmin=0, vmax=1, cmap="gray")
axs[0].set_title("Model")
axs[0].axis("tight")
axs[1].imshow(xinv, vmin=0, vmax=1, cmap="gray")
axs[1].set_title("TV Inversion")
axs[1].axis("tight")

fig, axs = plt.subplots(2, 1, figsize=(10, 5))
axs[0].plot(x[ny // 2], "k", lw=5, label="x")
axs[0].plot(xinv[ny // 2], "r", lw=3, label="xinv TV")
axs[0].set_title("Horizontal section")
axs[0].legend()
axs[1].plot(x[:, nx // 2], "k", lw=5, label="x")
axs[1].plot(xinv[:, nx // 2], "r", lw=3, label="xinv TV")
axs[1].set_title("Vertical section")
axs[1].legend()

```

Out:

```
<matplotlib.legend.Legend object at 0x7f6ae9949828>
```

Note that more optimized variations of the Split Bregman algorithm have been proposed in the literature for this specific problem, both improving the overall quality of the inversion and the speed of convergence.

In PyLops we however prefer to implement the generalized Split Bergman algorithm as this can be used for any sort of problem where we wish to add any number of L1 and/or L2 regularization terms to the cost function to minimize.

Total running time of the script: (0 minutes 17.984 seconds)

3.6.43 Transpose

This example shows how to use the `pylops.Transpose` operator. For arrays that are 2-dimensional in nature this operator simply transposes rows and columns. For multi-dimensional arrays, this operator can be used to permute dimensions

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

import pylops

plt.close("all")
np.random.seed(0)
```

Let's start by creating a 2-dimensional array

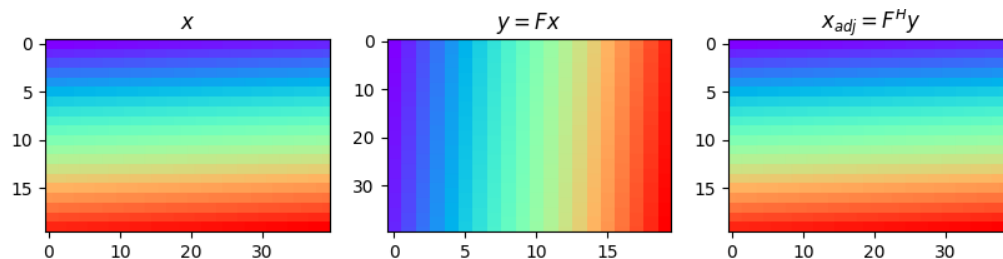
```
dims = (20, 40)
x = np.arange(800).reshape(dims)
```

We use now the `pylops.Transpose` operator to swap the two dimensions. As you will see the adjoint of this operator brings the data back to its original model, or in other words the adjoint operator is equal in this case to the inverse operator.

```
Top = pylops.Transpose(dims=dims, axes=(1, 0))

y = Top * x.ravel()
xadj = Top.H * y
y = y.reshape(Top.dimsd)
xadj = xadj.reshape(Top.dims)

fig, axs = plt.subplots(1, 3, figsize=(10, 2))
fig.suptitle("Transpose for 2d data", fontsize=14, fontweight="bold", y=1.15)
axs[0].imshow(x, cmap="rainbow", vmin=0, vmax=800)
axs[0].set_title(r"$x$")
axs[0].axis("tight")
axs[1].imshow(y, cmap="rainbow", vmin=0, vmax=800)
axs[1].set_title(r"$y = F x$")
axs[1].axis("tight")
axs[2].imshow(xadj, cmap="rainbow", vmin=0, vmax=800)
axs[2].set_title(r"$x_{adj} = F^H y$")
axs[2].axis("tight")
```



Out:

```
(-0.5, 39.5, 19.5, -0.5)
```

A similar approach can of course be taken to swap multiple axes of multi-dimensional arrays for any number of dimensions.

Total running time of the script: (0 minutes 0.198 seconds)

3.6.44 Wavelet estimation

This example shows how to use the `pylops.avo.prestack.PrestackWaveletModelling` to estimate a wavelet from pre-stack seismic data. This problem can be written in mathematical form as:

$$\mathbf{d} = \mathbf{G}\mathbf{w}$$

where \mathbf{G} is an operator that convolves an angle-variant reflectivity series with the wavelet \mathbf{w} that we aim to retrieve.

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

import pylops
from pylops.utils.wavelets import ricker

plt.close("all")
np.random.seed(0)
```

Let's start by creating the input elastic property profiles and wavelet

```
nt0 = 501
dt0 = 0.004
ntheta = 21

t0 = np.arange(nt0) * dt0
thetamin, thetamax = 0, 40
theta = np.linspace(thetamin, thetamax, ntheta)

# Elastic property profiles
vp = 1200 + np.arange(nt0) + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 80, nt0))
vs = 600 + vp / 2 + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 20, nt0))
rho = 1000 + vp + filtfilt(np.ones(5) / 5.0, 1, np.random.normal(0, 30, nt0))
vp[201:] += 500
vs[201:] += 200
rho[201:] += 100

# Wavelet
ntwav = 41
wavoff = 10
wav, twav, wavc = ricker(t0[: ntwav // 2 + 1], 20)
wav_phase = np.hstack((wav[wavoff:], np.zeros(wavoff)))

# vs/vp profile
vsvp = 0.5
vsvp_z = np.linspace(0.4, 0.6, nt0)

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

fig, axs = plt.subplots(1, 3, figsize=(13, 7), sharey=True)
axs[0].plot(vp, t0, "k")
axs[0].set_title("Vp")
axs[0].set_ylabel(r"$t$ (s) $")
```

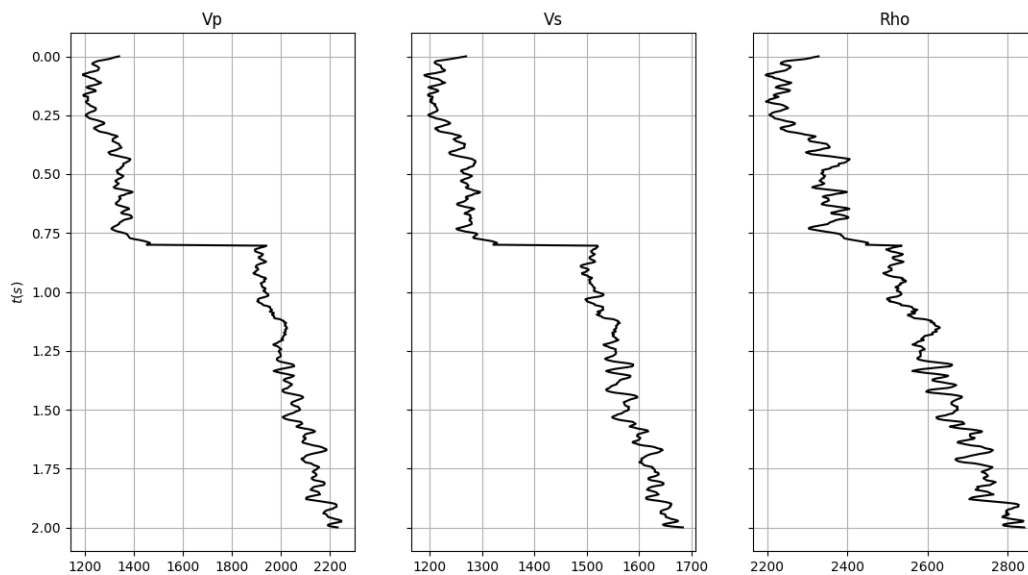
(continues on next page)

(continued from previous page)

```

axs[0].invert_yaxis()
axs[0].grid()
axs[1].plot(vs, t0, "k")
axs[1].set_title("Vs")
axs[1].invert_yaxis()
axs[1].grid()
axs[2].plot(rho, t0, "k")
axs[2].set_title("Rho")
axs[2].invert_yaxis()
axs[2].grid()

```



We create now the operators to model a synthetic pre-stack seismic gather with a zero-phase as well as a mixed phase wavelet.

```

# Create operators
Wavesop = pylops.avo.prestack.PrestackWaveletModelling(
    m, theta, nwav=ntwav, wavc=wavc, vsvp=vsvp, linearization="akirich"
)
Wavesop_phase = pylops.avo.prestack.PrestackWaveletModelling(
    m, theta, nwav=ntwav, wavc=wavc, vsvp=vsvp, linearization="akirich"
)

```

Let's apply those operators to the elastic model and create some synthetic data

```

d = (Wavesop * wav).reshape(ntheta, nt0).T
d_phase = (Wavesop_phase * wav_phase).reshape(ntheta, nt0).T

# add noise
dn = d + np.random.normal(0, 3e-2, d.shape)

fig, axs = plt.subplots(1, 3, figsize=(13, 7), sharey=True)
axs[0].imshow(
    d, cmap="gray", extent=(theta[0], theta[-1], t0[-1], t0[0]), vmin=-0.1, vmax=0.1
)

```

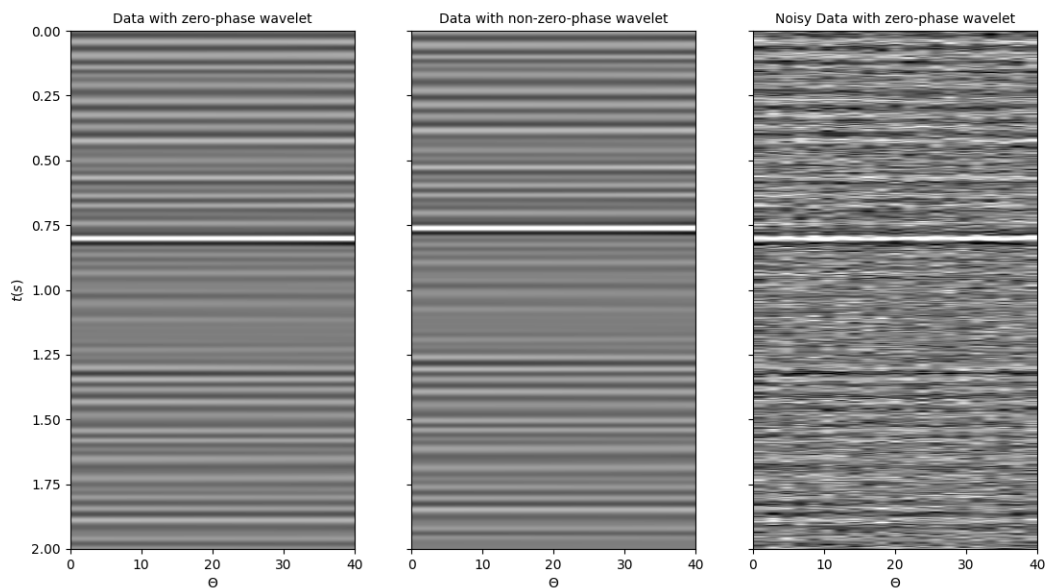
(continues on next page)

(continued from previous page)

```

)
axs[0].axis("tight")
axs[0].set_xlabel(r"$\Theta$")
axs[0].set_ylabel(r"$t(s)$")
axs[0].set_title("Data with zero-phase wavelet", fontsize=10)
axs[1].imshow(
    d_phase,
    cmap="gray",
    extent=(theta[0], theta[-1], t0[-1], t0[0]),
    vmin=-0.1,
    vmax=0.1,
)
axs[1].axis("tight")
axs[1].set_title("Data with non-zero-phase wavelet", fontsize=10)
axs[1].set_xlabel(r"$\Theta$")
axs[2].imshow(
    dn, cmap="gray", extent=(theta[0], theta[-1], t0[-1], t0[0]), vmin=-0.1, vmax=0.1
)
axs[2].axis("tight")
axs[2].set_title("Noisy Data with zero-phase wavelet", fontsize=10)
axs[2].set_xlabel(r"$\Theta$")

```



Out:

```
Text(0.5, 47.72222222222222, '$\Theta$')
```

We can invert the data. First we will consider noise-free data, subsequently we will add some noise and add a regularization terms in the inversion process to obtain a well-behaved wavelet also under noise conditions.

```

wav_est = Wavesop / d.T.ravel()
wav_phase_est = Wavesop_phase / d_phase.T.ravel()
wavn_est = Wavesop / dn.T.ravel()

```

(continues on next page)

(continued from previous page)

```

# Create regularization operator
D2op = pylops.SecondDerivative(ntwav, dtype="float64")

# Invert for wavelet
(
    wavn_reg_est,
    istop,
    itn,
    rlnorm,
    r2norm,
) = pylops.optimization.leastsquares.RegularizedInversion(
    Wavesop,
    [D2op],
    dn.T.ravel(),
    epsRs=[np.sqrt(0.1)],
    returninfo=True,
    **dict(damp=np.sqrt(1e-4), iter_lim=200, show=0)
)

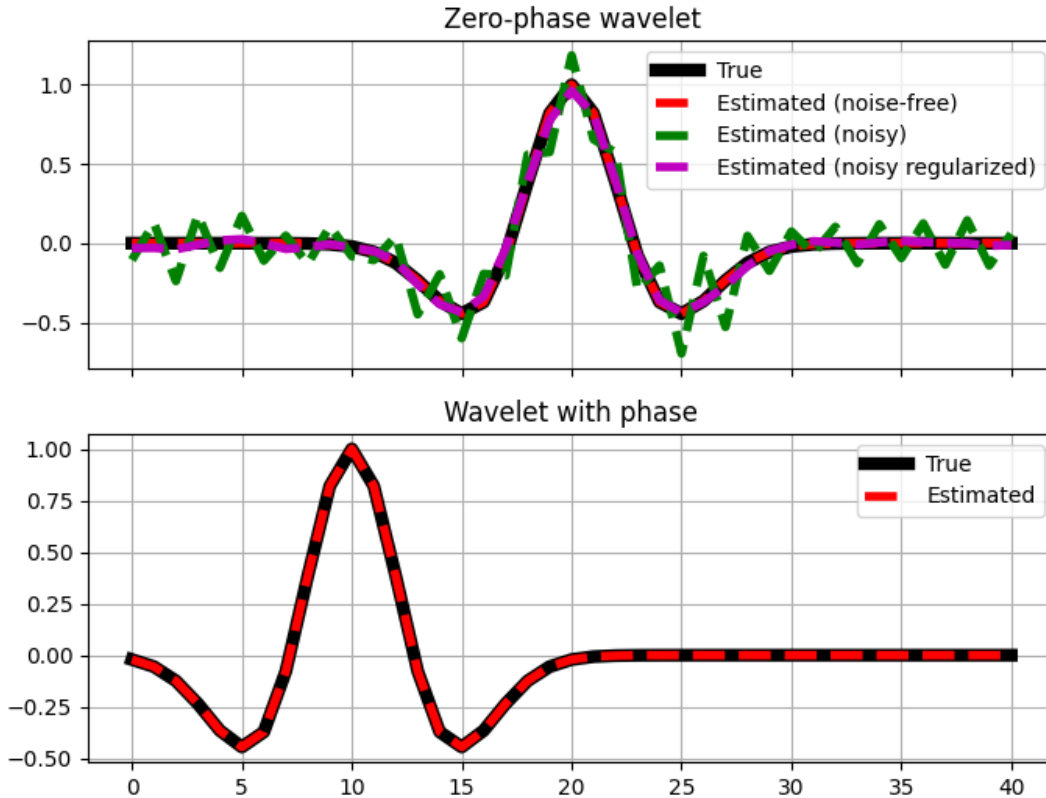
```

As expected, the regularization helps to retrieve a smooth wavelet even under noisy conditions.

```

# sphinx_gallery_thumbnail_number = 3
fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8, 6))
axs[0].plot(wav, "k", lw=6, label="True")
axs[0].plot(wav_est, "--r", lw=4, label="Estimated (noise-free)")
axs[0].plot(wavn_est, "--g", lw=4, label="Estimated (noisy)")
axs[0].plot(wavn_reg_est, "--m", lw=4, label="Estimated (noisy regularized)")
axs[0].set_title("Zero-phase wavelet")
axs[0].grid()
axs[0].legend(loc="upper right")
axs[0].axis("tight")
axs[1].plot(wav_phase, "k", lw=6, label="True")
axs[1].plot(wav_phase_est, "--r", lw=4, label="Estimated")
axs[1].set_title("Wavelet with phase")
axs[1].grid()
axs[1].legend(loc="upper right")
axs[1].axis("tight")

```



Out:

```
(-2.0, 42.0, -0.517181247702497, 1.0722467265486844)
```

Finally we repeat the same exercise, but this time we use a *preconditioner*. Initially, our preconditioner is a `pylops.Symmetrize` operator to ensure that our estimated wavelet is zero-phase. After we chain the `pylops.Symmetrize` and the `pylops.Smoothing1D` operators to also guarantee a smooth wavelet.

```
# Create symmetrize operator
Sop = pylops.Symmetrize((ntwav + 1) // 2)

# Create smoothing operator
Smop = pylops.Smoothing1D(5, dims=((ntwav + 1) // 2,), dtype="float64")

# Invert for wavelet
wavn_prec_est = pylops.optimization.leastsquares.PreconditionedInversion(
    Wavesop,
    Sop,
    dn.T.ravel(),
    returninfo=False,
    **dict(damp=np.sqrt(1e-4), iter_lim=200, show=0)
)

wavn_smooth_est = pylops.optimization.leastsquares.PreconditionedInversion(
    Wavesop,
```

(continues on next page)

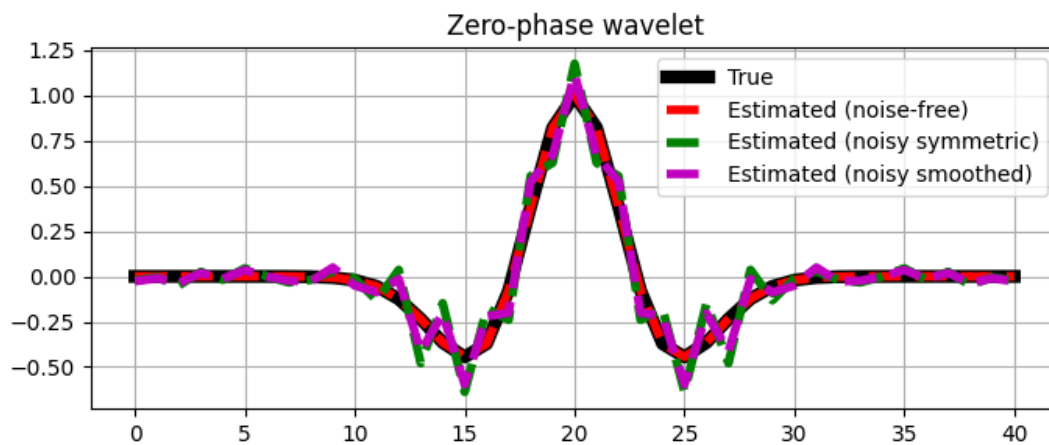
(continued from previous page)

```

Sop * Smop,
dn.T.ravel(),
returninfo=False,
**dict(damp=np.sqrt(1e-4), iter_lim=200, show=0)
)

fig, ax = plt.subplots(1, 1, sharex=True, figsize=(8, 3))
ax.plot(wav, "k", lw=6, label="True")
ax.plot(wav_est, "--r", lw=4, label="Estimated (noise-free)")
ax.plot(wavn_prec_est, "--g", lw=4, label="Estimated (noisy symmetric)")
ax.plot(wavn_smooth_est, "--m", lw=4, label="Estimated (noisy smoothed)")
ax.set_title("Zero-phase wavelet")
ax.grid()
ax.legend(loc="upper right")

```



Out:

```
<matplotlib.legend.Legend object at 0x7f6aece624a8>
```

Total running time of the script: (0 minutes 1.814 seconds)

3.6.45 Wavelet transform

This example shows how to use the `pylops.DWT` and `pylops.DWT2D` operators to perform 1- and 2-dimensional DWT.

```

import matplotlib.pyplot as plt
import numpy as np

import pylops

plt.close("all")

```

Let's start with a 1-dimensional signal. We apply the 1-dimensional wavelet transform, keep only the first 30 coefficients and perform the inverse transform.

```

nt = 200
dt = 0.004

```

(continues on next page)

(continued from previous page)

```

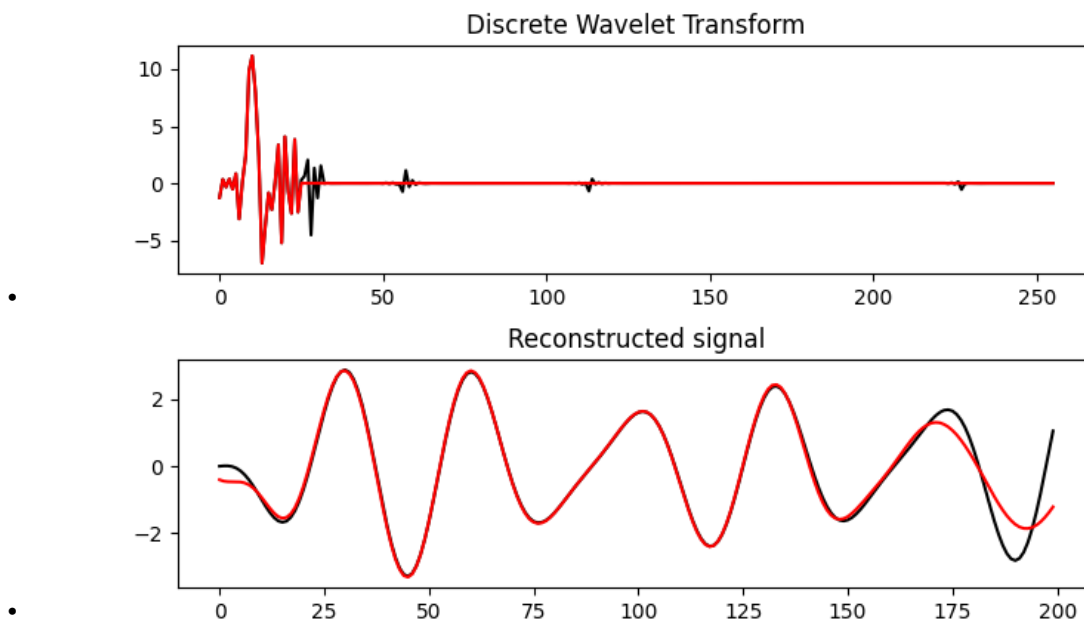
t = np.arange(nt) * dt
freqs = [10, 7, 9]
amps = [1, -2, 0.5]
x = np.sum([amp * np.sin(2 * np.pi * f * t) for (f, amp) in zip(freqs, amps)], axis=0)

Wop = pylops.signalprocessing.DWT(nt, wavelet="dmey", level=5)
y = Wop * x
yf = y.copy()
yf[25:] = 0
xinv = Wop.H * yf

plt.figure(figsize=(8, 2))
plt.plot(y, "k", label="Full")
plt.plot(yf, "r", label="Extracted")
plt.title("Discrete Wavelet Transform")

plt.figure(figsize=(8, 2))
plt.plot(x, "k", label="Original")
plt.plot(xinv, "r", label="Reconstructed")
plt.title("Reconstructed signal")

```



Out:

```
Text(0.5, 1.0, 'Reconstructed signal')
```

We repeat the same procedure with an image. In this case the 2-dimensional DWT will be applied instead. Only a quarter of the coefficients of the DWT will be retained in this case.

```

im = np.load("../testdata/python.npy")[:, :5, :5, 0]

Nz, Nx = im.shape
Wop = pylops.signalprocessing.DWT2D((Nz, Nx), wavelet="haar", level=5)
y = Wop * im.ravel()
yf = y.copy()
yf[len(y) // 4 :] = 0

```

(continues on next page)

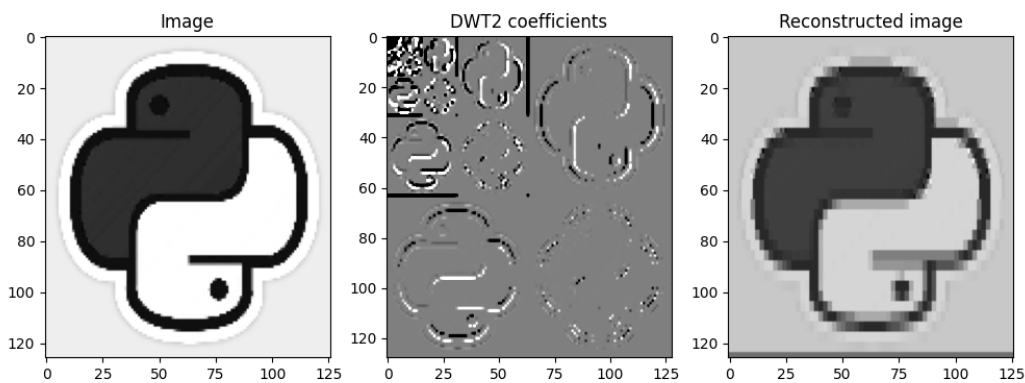
(continued from previous page)

```

iminv = Wop.H * yf
iminv = iminv.reshape(Nz, Nx)

fig, axs = plt.subplots(1, 3, figsize=(12, 4))
axs[0].imshow(im, cmap="gray")
axs[0].set_title("Image")
axs[0].axis("tight")
axs[1].imshow(y.reshape(Wop.dimsd), cmap="gray_r", vmin=-1e2, vmax=1e2)
axs[1].set_title("DWT2 coefficients")
axs[1].axis("tight")
axs[2].imshow(iminv, cmap="gray")
axs[2].set_title("Reconstructed image")
axs[2].axis("tight")

```



Out:

```
(-0.5, 125.5, 125.5, -0.5)
```

Total running time of the script: (0 minutes 0.447 seconds)

3.6.46 Zero

This example shows how to use the `pylops.basicoperators.Zero` operator. This operators simply zeroes the data in forward mode and the model in adjoint mode.

```

import matplotlib.gridspec as pltgs
import matplotlib.pyplot as plt
import numpy as np

import pylops

plt.close("all")

```

Let's define an zero operator $\mathbf{0}$ with same number of elements for data N and model M .

```

N, M = 5, 5
x = np.arange(M)
Zop = pylops.basicoperators.Zero(M, dtype="int")

y = Zop * x

```

(continues on next page)

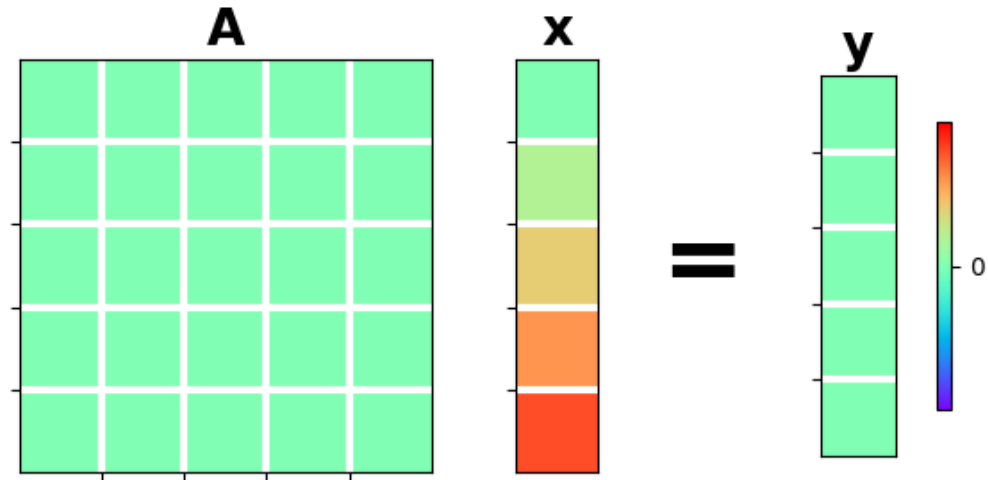
(continued from previous page)

```

xadj = Zop.H * y

gs = pltgs.GridSpec(1, 6)
fig = plt.figure(figsize=(7, 3))
ax = plt.subplot(gs[0, 0:3])
ax.imshow(np.zeros((N, N)), cmap="rainbow", vmin=-M, vmax=M)
ax.set_title("A", 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 = plt.subplot(gs[0, 3])
im = ax.imshow(x[:, np.newaxis], cmap="rainbow", vmin=-M, vmax=M)
ax.set_title("x", size=20, fontweight="bold")
ax.set_xticks([])
ax.set_yticks(np.arange(M - 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(y[:, np.newaxis], cmap="rainbow", vmin=-M, vmax=M)
ax.set_title("y", 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([])
fig.colorbar(im, ax=ax, ticks=[0], pad=0.3, shrink=0.7)

```



Out:

```
<matplotlib.colorbar.Colorbar object at 0x7f6aec99ffd0>
```

Similarly we can consider the case with data bigger than model

```
N, M = 10, 5
x = np.arange(M)
Zop = pylops.Zero(N, M, dtype="int")

y = Zop * x
xadj = Zop.H * y

print("x = %s" % x)
print("0*x = %s" % y)
print("0'*y = %s" % xadj)
```

Out:

```
x = [0 1 2 3 4]
0*x = [0 0 0 0 0 0 0 0 0 0]
0'*y = [0 0 0 0 0]
```

and model bigger than data

```
N, M = 5, 10
x = np.arange(M)
Zop = pylops.Zero(N, M, dtype="int")

y = Zop * x
xadj = Zop.H * y

print("x = %s" % x)
print("0*x = %s" % y)
print("0'*y = %s" % xadj)
```

Out:

```
x = [0 1 2 3 4 5 6 7 8 9]
0*x = [0 0 0 0 0]
0'*y = [0 0 0 0 0 0 0 0 0 0]
```

Note that this operator can be useful in many real-life applications when for example we want to manipulate a subset of the model array and keep intact the rest of the array. For example:

$$\begin{bmatrix} \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \mathbf{A}\mathbf{x}_1$$

Refer to the tutorial on *Optimization* for more details on this.

Total running time of the script: (0 minutes 0.163 seconds)

3.7 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.

2. Can I have an older version of cupy installed in my system (“cupy-cudaXX<8.1.0”)?**** Yes. Nevertheless you need to tell PyLops that you don't want to use its `cupy` backend by setting the environment variable `CUPY_PYLOPS=0`. Failing to do so will lead to an error when you import `pylops` because some of the `cupyx` routines that we use are not available in earlier version of cupy.

3.8 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).

3.8.1 Linear operators

Templates

`LinearOperator([Op, explicit, clinear])`

Common interface for performing matrix-vector products.

Continued on next page

Table 1 – continued from previous page

<code>FunctionOperator(f, *args, **kwargs)</code>	Function Operator.
<code>MemoizeOperator(Op[, max_neval])</code>	Memoize Operator.

pylops.LinearOperator

class `pylops.LinearOperator` (*Op=None, explicit=False, clinear=None*)

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

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)

clinear [`bool`] New in version 1.17.0.

Operator is complex-linear.

Methods

<code>__init__([Op, explicit, clinear])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

matvec (*x*)

Matrix-vector multiplication.

Modified version of scipy matvec which does not consider the case where the input vector is `np.matrix` (the use `np.matrix` is now discouraged in numpy's documentation).

Parameters

x [`numpy.ndarray`] Input array of shape (N,) or (N,1)

Returns

y [`numpy.ndarray`] Output array of shape (M,) or (M,1)

rmatvec (*x*)

Adjoint matrix-vector multiplication.

Modified version of scipy rmatvec which does not consider the case where the input vector is `np.matrix` (the use `np.matrix` is now discouraged in numpy's documentation).

Parameters

y [`numpy.ndarray`] Input array of shape (M,) or (M,1)

Returns

x [`numpy.ndarray`] Output array of shape (N,) or (N,1)

matmat (*X*)

Matrix-matrix multiplication.

Modified version of scipy matmat which does not consider the case where the input vector is `np.matrix` (the use `np.matrix` is now discouraged in numpy's documentation).

Parameters

x [`numpy.ndarray`] Input array of shape (N,K)

Returns

y [`numpy.ndarray`] Output array of shape (M,K)

rmatmat (*X*)

Matrix-matrix multiplication.

Modified version of scipy rmatmat which does not consider the case where the input vector is `np.matrix` (the use `np.matrix` is now discouraged in numpy's documentation).

Parameters

x [`numpy.ndarray`] Input array of shape (M,K)

Returns

y [`numpy.ndarray`] Output array of shape (N,K)

dot (*x*)

Matrix-matrix or matrix-vector multiplication.

Parameters

x [`np.ndarray`] Input array (or matrix)

Returns

y [`np.ndarray`] Output array (or matrix) that represents the result of applying the linear operator on *x*.

div (*y*, *niter*=100, *densesolver*='scipy')

Solve the linear problem $\mathbf{y} = \mathbf{Ax}$.

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)

densesolver [*str*, optional] Use *scipy* (*scipy*) or *numpy* (*numpy*) dense solver

Returns

xest [*np.ndarray*] Estimated model

todense (*backend*='numpy')

Return dense matrix.

The operator is converted into its dense matrix equivalent. In order to do so, square or tall operators are applied to an identity matrix whose number of rows and columns is equivalent to the number of columns of the operator. Conversely, for skinny operators, the transpose operator is applied to an identity matrix whose number of rows and columns is equivalent to the number of rows of the operator and the resulting matrix is transposed (and complex conjugated).

Note that this operation may be costly for operators with large number of rows and columns and it should be used mostly as a way to inspect the structure of the matricial equivalent of the operator.

Parameters

backend [*str*, optional] Backend used to densify matrix (*numpy* or *cupy*). Note that this must be consistent with how the operator has been created.

Returns

matrix [*numpy.ndarray* or *cupy.ndarray*] Dense matrix.

tosparse ()

Return sparse matrix.

The operator is 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

matrix [*scipy.sparse.csr_matrix*] Sparse matrix.

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 $\mathbf{A}^H \mathbf{A}$ 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

eigenvalues [numpy.ndarray] Operator eigenvalues.

Notes

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

$$k = \frac{\lambda_{max}}{\lambda_{min}}$$

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

toreal(forw=True, adj=True)

Real operator

Parameters

forw [*bool*, optional] Apply real to output of forward pass

adj [*bool*, optional] Apply real to output of adjoint pass

Returns

realop [*pylops.LinearOperator*] Real operator

toimag(forw=True, adj=True)

Imag operator

Parameters

forw [*bool*, optional] Apply imag to output of forward pass

adj [*bool*, optional] Apply imag to output of adjoint pass

Returns

imagop [*pylops.LinearOperator*] Imag operator

trace(neval=None, method=None, backend='numpy', **kwargs_trace)

Trace of linear operator.

Returns the trace (or its estimate) of the linear operator.

Parameters

neval [*int*, optional] Maximum number of matrix-vector products compute. Default depends `method`.

method [`str`, optional] Should be one of the following:

- **explicit**: If the operator is not explicit, will convert to dense first.
- **hutchinson**: see `pylops.utils.trace_hutchinson`
- **hutch++**: see `pylops.utils.trace_hutchpp`
- **na-hutch++**: see `pylops.utils.trace_nahutchpp`

Defaults to 'explicit' for explicit operators, and 'Hutch++' for the rest.

backend [`str`, optional] Backend used to densify matrix (numpy or cupy). Note that this must be consistent with how the operator has been created.

****kwargs_trace** Arbitrary keyword arguments passed to `pylops.utils.trace_hutchinson`, `pylops.utils.trace_hutchpp`, or `pylops.utils.trace_nahutchpp`

Returns

trace [`self.dtype`] Operator trace.

Raises

ValueError If the operator has rectangular shape (`shape[0] != shape[1]`)

NotImplementedError If the method is not one of the available methods.

Examples using `pylops.LinearOperator`

- *1D, 2D and 3D Sliding*
- *2D Patching*
- *AVO modelling*
- *Bilinear Interpolation*
- *CGLS and LSQR Solvers*
- *Causal Integration*
- *Chirp Radon Transform*
- *Conj*
- *Convolution*
- *Derivatives*
- *Describe*
- *Diagonal*
- *Flip along an axis*
- *Identity*
- *Imag*
- *Linear Regression*
- *MP, OMP, ISTA and FISTA*
- *Matrix Multiplication*
- *Multi-Dimensional Convolution*

- *Normal Moveout (NMO) Correction*
- *Operators concatenation*
- *Operators with Multiprocessing*
- *Padding*
- *PhaseShift operator*
- *Polynomial Regression*
- *Pre-stack modelling*
- *Real*
- *Restriction and Interpolation*
- *Roll*
- *Seislet transform*
- *Shift*
- *Spread How-to*
- *Sum*
- *Symmetrize*
- *Total Variation (TV) Regularization*
- *Transpose*
- *Wavelet estimation*
- *Wavelet transform*
- *Zero*
- *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*
- *16. CT Scan Imaging*
- *17. Real/Complex Inversion*

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 $f : \mathbb{F}^m \rightarrow \mathbb{F}_c^n$ and $f_c : \mathbb{F}_c^n \rightarrow \mathbb{F}^m$ where \mathbb{F} and \mathbb{F}_c are the underlying fields (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*, *fc*, *n*), and FunctionOperator(*f*, *fc*, *n*, *m*).

The first two methods can only be used for forward modelling and will return 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 *dtype* keyword argument.

Parameters

- f** [callable] Function for forward multiplication.
- fc** [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).
- dtype** [str, optional] Type of elements in input array.

Examples

```
>>> 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

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

Methods

<code>__init__(f, *args, **kwargs)</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.

Continued on next page

Table 3 – continued from previous page

<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{Ax}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

pylops.MemoizeOperator

class `pylops.MemoizeOperator` (*Op*, *max_neval=10*)

Memoize Operator.

This operator can be used to wrap any PyLops operator and add a memoize functionality and stores the last `max_neval` model/data vector pairs

Parameters

Op [`pylops.LinearOperator`] PyLops linear operator

max_neval [`int`, optional] Maximum number of previous evaluations stored, use `np.inf` for infinite memory

Attributes

shape [`tuple`] Operator shape $[n \times m]$

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

Methods

<code>__init__(Op[, max_neval])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{Ax}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator

Continued on next page

Table 4 – continued from previous page

<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.MemoizeOperator`

- 17. *Real/Complex Inversion*

Basic operators

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

`pylops.MatrixMult`

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

Matrix multiplication.

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

Parameters

A [`numpy.ndarray` or `scipy.sparse` matrix] Matrix.

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

<code>__init__(A[, dims, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>inv()</code>	Return the inverse of A .
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

inv()

Return the inverse of **A**.

Returns

Ainv [numpy.ndarray] Inverse matrix.

Examples using `pylops.MatrixMult`

- *CGLS and LSQR Solvers*
- *Describe*
- *MP, OMP, ISTA and FISTA*
- *Matrix Multiplication*
- *Operators concatenation*
- *Operators with Multiprocessing*
- *02. The Dot-Test*
- *03. Solvers*
- *07. Post-stack inversion*
- *08. Pre-stack (AVO) inversion*
- *17. Real/Complex Inversion*

pylops.Identity

class `pylops.Identity` ($N, M=None, \text{dtype}='float64', \text{inplace}=True$)

Identity operator.

Simply move model to data in forward model and viceversa 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 \mathbf{x} to the data \mathbf{y} in forward mode and viceversa in adjoint mode:

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

or in matrix form:

$$\mathbf{y} = \mathbf{I}\mathbf{x} = \mathbf{x}$$

and

$$\mathbf{x} = \mathbf{I}\mathbf{y} = \mathbf{y}$$

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

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

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

$$\begin{aligned} x_i &= y_i \quad \forall i = 1, 2, \dots, M \\ x_i &= 0 \quad \forall i = M + 1, \dots, N \end{aligned}$$

Attributes

shape [`tuple`] Operator shape

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

Methods

<code>__init__(N[, M, dtype, inplace])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator

Continued on next page

Table 7 – continued from previous page

<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{Ax}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Identity`

- *Identity*
- *Operators concatenation*
- *Total Variation (TV) Regularization*
- *04. Bayesian Inversion*

`pylops.Zero`

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 \mathbf{y} in forward mode:

$$\mathbf{0x} = \mathbf{0}_N$$

and a null model vector \mathbf{x} in forward mode:

$$\mathbf{0y} = \mathbf{0}_M$$

Attributes

shape [`tuple`] Operator shape

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

Methods

<code>__init__(N[, M, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Zero`

- [Zero](#)

`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 \mathbf{x} and/or data \mathbf{y} vectors and the array \mathbf{d} can be expressed as

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

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

<code>__init__(diag[, dims, dir, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matrix()</code>	Return diagonal matrix as dense <code>numpy.ndarray</code>
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense()</code>	Fast implementation of <code>todense</code> based on known structure of the operator
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

`matrix()`

Return diagonal matrix as dense `numpy.ndarray`

Returns

densemat `[numpy.ndarray]` Dense matrix.

`todense()`

Fast implementation of `todense` based on known structure of the operator

Returns

densemat `[numpy.ndarray]` Dense matrix.

Examples using `pylops.Diagonal`

- *Describe*
- *Diagonal*
- *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

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

<code>__init__(dims, axes[, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.

Continued on next page

Table 10 – continued from previous page

<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Transpose`

- *Transpose*

`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:

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

where *N* is the lenght 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

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

Continued on next page

Table 11 – continued from previous page

<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Flip`

- *Flip along an axis*

`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 *dir* for a chosen number of samples (*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

<code>__init__</code> (<i>N</i> , <i>dims</i> , <i>dir</i> , <i>shift</i> , <i>dtype</i>)	Initialize this LinearOperator.
<code>adjoint</code> ()	Hermitian adjoint.
<code>apply_columns</code> (<i>cols</i>)	Apply subset of columns of operator
<code>cond</code> (<i>[uselobpcg]</i>)	Condition number of linear operator.
<code>conj</code> ()	Complex conjugate operator
<code>div</code> (<i>y</i> , <i>niter</i> , <i>densesolver</i>)	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot</code> (<i>x</i>)	Matrix-matrix or matrix-vector multiplication.

Continued on next page

Table 12 – continued from previous page

<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Roll`

- [Roll](#)

`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

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:

$$y_i = x_{i-\text{pad}_{\text{in}}} \quad \forall i = \text{pad}_{\text{in}}, \dots, \text{pad}_{\text{in}} + N - 1$$

and $y_i = 0 \quad \forall i = 0, \dots, \text{pad}_{\text{in}} - 1, \text{pad}_{\text{in}} + N - 1, \dots, N + \text{pad}_{\text{in}} + \text{pad}_{\text{end}}$

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

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

Attributes

shape [`tuple`] Operator shape

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

Methods

<code>__init__(dims, pad[, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Pad`

- *Padding*

`pylops.Sum`

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

<code>__init__(dims, dir[, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{Ax}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Sum`

- *Sum*

`pylops.Symmetrize`

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:

$$y[i] = \begin{cases} x[i - N + 1], & i \geq N \\ x[N - 1 - i], & \text{otherwise} \end{cases}$$

for $i = 0, 1, 2, \dots, 2N - 2$, where N is the length of the input model.

In adjoint mode, the Symmetrize operator assigns the sums of the elements in position $N - 1 - i$ and $N - 1 + i$ to position i as follows:

apart from the central sample where $x[0] = y[N - 1]$.

Attributes

shape [tuple] Operator shape

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

Methods

<code>__init__(N[, dims, dir, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Symmetrize`

- *Symmetrize*
- *Wavelet estimation*

`pylops.Restriction`

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 \mathbf{x} of size M can be expressed as:

$$y_i = x_{l_i} \quad \forall i = 0, 1, \dots, N - 1$$

where $\mathbf{l} = [l_0, l_1, \dots, l_{N-1}]$ is a vector containing the indices of the original array at which samples are taken.

Conversely, in adjoint mode the available values in the data vector \mathbf{y} are placed at locations $\mathbf{l} = [l_0, l_1, \dots, l_{M-1}]$ in the model vector:

$$x_{l_i} = y_i \quad \forall i = 0, 1, \dots, N - 1$$

and $x_j = 0$ for $j \neq l_i$ (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

<code>__init__(M, iava[, dims, dir, dtype, inplace])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{Ax}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>mask(x)</code>	Apply mask to input signal returning a signal of same size with values at <code>iava</code> locations and 0 at other locations
<code>matmat(X)</code>	Matrix-matrix multiplication.

Continued on next page

Table 16 – continued from previous page

<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

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` or `cupy.ndarray`] Input array (can be either flattened or not)

Returns

y [`numpy.ma.core.MaskedArray`] Masked array.

Examples using `pylops.Restriction`

- *Restriction and Interpolation*
- *Total Variation (TV) Regularization*
- *03. Solvers*
- *04. Bayesian Inversion*
- *06. 2D Interpolation*
- *12. Seismic regularization*

pylops.Reggression

class `pylops.Reggression` (*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 `taxis` is not `numpy.ndarray`.

See also:

LinearRegression Linear regression**Notes**

The Regression operator solves the following problem:

$$y_i = \sum_{n=0}^{\text{order}} x_n t_i^n \quad \forall i = 0, 1, \dots, N-1$$

where N represents the number of points in `taxis`. We can express this problem in a matrix form

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

where

$$\mathbf{y} = [y_0, y_1, \dots, y_{N-1}]^T, \quad \mathbf{x} = [x_0, x_1, \dots, x_{\text{order}}]^T$$

and

$$\mathbf{A} = \begin{bmatrix} 1 & t_0 & t_0^2 & \dots & t_0^{\text{order}} \\ 1 & t_1 & t_1^2 & \dots & t_1^{\text{order}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_{N-1} & t_{N-1}^2 & \dots & t_{N-1}^{\text{order}} \end{bmatrix}_{N \times \text{order}+1}$$

Attributes

shape [`tuple`] Operator shape

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

Methods

<code>__init__(taxis, order[, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply(t, x)</code>	Return values along y-axis given certain <code>t</code> location(s) along t-axis and regression coefficients <code>x</code>
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

apply (*t*, *x*)

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

Parameters

taxis [`numpy.ndarray`] Elements along the t-axis.

x [`numpy.ndarray`] Regression coefficients

dtype [`str`, optional]

Returns

y [`numpy.ndarray`] Values along y-axis

Examples using `pylops.Regression`

- *Linear Regression*
- *Polynomial Regression*

`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 [`numpy.ndarray`] Elements along the *t*-axis.

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

Raises

TypeError If *taxis* is not `numpy.ndarray`.

See also:

Regression Polynomial regression

Notes

The `LinearRegression` operator solves the following problem:

$$y_i = x_0 + x_1 t_i \quad \forall i = 0, 1, \dots, N - 1$$

We can express this problem in a matrix form

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

where

$$\mathbf{y} = [y_0, y_1, \dots, y_{N-1}]^T, \quad \mathbf{x} = [x_0, x_1]^T$$

and

$$\mathbf{A} = \begin{bmatrix} 1 & t_0 \\ 1 & t_1 \\ \vdots & \vdots \\ 1 & t_{N-1} \end{bmatrix}$$

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`

- *Linear Regression*

`pylops.CausalIntegration`

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

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 [`bool`, optional] Add half of current value (`True`) or the entire value (`False`).
This will be *deprecated* in v2.0.0, use instead `kind=half` to obtain the same behaviour.

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

kind [`str`, optional] Integration kind (`full`, `half`, or `trapezoidal`).

removefirst [`bool`, optional] Remove first sample (`True`) or not (`False`).

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:

$$y(t) = \int_{-\infty}^t x(\tau) d\tau$$

which can be discretised as :

$$y[i] = \sum_{j=0}^i x[j] \Delta t$$

or

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

or

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

where Δt is the sampling interval, and assuming the signal is zero before sample $j = 0$. In our implementation, the choice to add $x[i]$ or $0.5x[i]$ is made by selecting `kind=full` or `kind=half`, respectively. The choice to add $0.5x[i]$ and $0.5x[0]$ instead of made by selecting the `kind=trapezoidal`.

Note that the causal integral of a signal will depend, up to a constant, on causal start of the signal. For example if $x(\tau) = t^2$ the resulting indefinite integration is:

$$y(t) = \int \tau^2 d\tau = \frac{t^3}{3} + C$$

However, if we apply a first derivative to y always 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

<code>__init__(N[, dims, dir, sampling, ...])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.CausalIntegration`

- *Causal Integration*

`pylops.Spread`

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

Spread operator.

Spread values from the input model vector arranged as a 2-dimensional array of size $[n_{x_0} \times n_{t_0}]$ into the data vector of size $[n_x \times n_t]$. Note that the value at each single pair (x_0, t_0) in the input is spread over the entire x axis in the output.

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 $[n_{x_0} \times n_{t_0}]$, where the first dimension is the spreading direction)

dimsd [`tuple`] Dimensions of data vector (vector will be reshaped internal into a two-dimensional array of size $[n_x \times n_t]$, where the first dimension is the stacking direction)

table [`np.ndarray`, optional] Look-up table of indices of size $[n_{x_0} \times n_{t_0} \times n_x]$ (if `None` use function handle *fh*). When *dtable* is not provided, the data will be created as follows

```
data[ix, table[ix0, it0, ix]] += model[ix0, it0]
```

Note: When using *table* without *dtable*, its elements must be between 0 and $n_{t_0} - 1$ (or `numpy.nan`).

dtable [`np.ndarray`, optional] Look-up table of decimals remainders for linear interpolation of size $[n_{x_0} \times n_{t_0} \times n_x]$ (if `None` use function handle *fh*). When provided, the data will be created as follows

```
data[ix, table[ix0, it0, ix]] += (1 - dtable[ix0, it0, ix]) * ↪
↪model[ix0, it0]
data[ix, table[ix0, it0, ix] + 1] +=      dtable[ix0, it0, ix] * ↪
↪model[ix0, it0]
```

Note: When using *table* and *dtable*, the elements of *table* indices must be between 0 and $n_{t_0} - 2$ (or `numpy.nan`).

fh [`callable`, optional] If `None` will use look-up table *table*. When provided, should be a function which takes indices *ix0* and *it0* and returns an array of size n_x containing each respective time index. Alternatively, if linear interpolation is required, it should output in addition to the time indices, a weight for interpolation with linear interpolation, to be used as follows

```
data[ix, index]      += (1 - dindices[ix]) * model[ix0, it0]
data[ix, index + 1] +=      dindices[ix] * model[ix0, it0]
```

where `index` refers to a time index in the first array returned by `fh` and `dindices` refers to the weight in the second array returned by `fh`.

Note: When using `fh` with one output (time indices), the time indices must be between 0 and $n_{t_0} - 1$ (or `numpy.nan`). When using `fh` with two outputs (time indices and weights), they must be within the between 0 and $n_{t_0} - 2$ (or `numpy.nan`).

interp [`bool`, optional] Use only if `engine='numba'`. Apply linear interpolation (`True`) or nearest interpolation (`False`) during stacking/spreading along parametric curve. When using `engine="numpy"`, it will be inferred directly from `fh` or the presence of `dtable`.

engine [`str`, optional] Engine used for spread 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 $[n_{x_0} \times n_{t_0} \times n_x]$

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 $[n_x \times n_t]$:

$$m(x_0, t_0) \rightarrow d(x, t = f(x_0, x, t_0)) \quad \forall x$$

where $f(x_0, x, t)$ is a mapping function that returns a value t given values x_0 , x , and t_0 . Note that for each (x_0, t_0) pair, spreading is done over the entire x axis in the data domain.

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 also possible to perform linear interpolation as part of the spreading/stacking process by providing the decimal part of the mapping function ($t - [t]$) 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

<code>__init__(dims, dimsd[, table, dtable, fh, ...])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Spread`

- *1D, 2D and 3D Sliding*
- *Normal Moveout (NMO) Correction*
- *Radon Transform*
- *Spread How-to*
- *11. Radon filtering*
- *12. Seismic regularization*
- *16. CT Scan Imaging*

`pylops.VStack`

class `pylops.VStack` (*ops, nproc=1, 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.

nproc [*int*, optional] Number of processes used to evaluate the N operators in parallel using multiprocessing. If `nproc=1`, work in serial mode.

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

$$\begin{bmatrix} \mathbf{L}_1 \\ \mathbf{L}_2 \\ \vdots \\ \mathbf{L}_N \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{L}_1 \mathbf{x} \\ \mathbf{L}_2 \mathbf{x} \\ \vdots \\ \mathbf{L}_N \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_N \end{bmatrix}$$

while its application in adjoint mode leads to

$$\begin{bmatrix} \mathbf{L}_1^H & \mathbf{L}_2^H & \dots & \mathbf{L}_N^H \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_N \end{bmatrix} = \mathbf{L}_1^H \mathbf{y}_1 + \mathbf{L}_2^H \mathbf{y}_2 + \dots + \mathbf{L}_N^H \mathbf{y}_N$$

Attributes

shape [tuple] Operator shape

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

Methods

<code>__init__(ops[, nproc, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.VStack`

- *Derivatives*
- *Describe*
- *Operators concatenation*
- *Operators with Multiprocessing*
- *Wavelet estimation*

- 08. Pre-stack (AVO) inversion
- 17. Real/Complex Inversion

pylops.HStack

class `pylops.HStack` (*ops*, *nproc*=1, *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.

nproc [`int`, optional] Number of processes used to evaluate the N operators in parallel using multiprocessing. If `nproc`=1, work in serial mode.

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

$$\begin{bmatrix} \mathbf{L}_1 & \mathbf{L}_2 & \dots & \mathbf{L}_N \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_N \end{bmatrix} = \mathbf{L}_1 \mathbf{x}_1 + \mathbf{L}_2 \mathbf{x}_2 + \dots + \mathbf{L}_N \mathbf{x}_N$$

while its application in adjoint mode leads to

$$\begin{bmatrix} \mathbf{L}_1^H \\ \mathbf{L}_2^H \\ \vdots \\ \mathbf{L}_N^H \end{bmatrix} \mathbf{y} = \begin{bmatrix} \mathbf{L}_1^H \mathbf{y} \\ \mathbf{L}_2^H \mathbf{y} \\ \vdots \\ \mathbf{L}_N^H \mathbf{y} \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_N \end{bmatrix}$$

Attributes

shape [`tuple`] Operator shape

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

Methods

<code>__init__(ops[, nproc, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{Ax}$.

Continued on next page

Table 21 – continued from previous page

<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.HStack`

- *Describe*
- *Operators concatenation*
- *Operators with Multiprocessing*

`pylops.Block`

`pylops.Block(ops, nproc=1, 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.ndarray` or `scipy.sparse` matrices can be passed in place of one or more operators.
- nproc** [`int`, optional] Number of processes used to evaluate the N operators in parallel using multiprocessing. If `nproc=1`, work in serial mode.
- 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

$$\begin{bmatrix} \mathbf{L}_{1,1} & \mathbf{L}_{1,2} & \cdots & \mathbf{L}_{1,M} \\ \mathbf{L}_{2,1} & \mathbf{L}_{2,2} & \cdots & \mathbf{L}_{2,M} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{L}_{N,1} & \mathbf{L}_{N,2} & \cdots & \mathbf{L}_{N,M} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_M \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{1,1}\mathbf{x}_1 + \mathbf{L}_{1,2}\mathbf{x}_2 + \mathbf{L}_{1,M}\mathbf{x}_M \\ \mathbf{L}_{2,1}\mathbf{x}_1 + \mathbf{L}_{2,2}\mathbf{x}_2 + \mathbf{L}_{2,M}\mathbf{x}_M \\ \vdots \\ \mathbf{L}_{N,1}\mathbf{x}_1 + \mathbf{L}_{N,2}\mathbf{x}_2 + \mathbf{L}_{N,M}\mathbf{x}_M \end{bmatrix}$$

while its application in adjoint mode leads to

$$\begin{bmatrix} \mathbf{L}_{1,1}^H & \mathbf{L}_{2,1}^H & \cdots & \mathbf{L}_{N,1}^H \\ \mathbf{L}_{1,2}^H & \mathbf{L}_{2,2}^H & \cdots & \mathbf{L}_{N,2}^H \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{L}_{1,M}^H & \mathbf{L}_{2,M}^H & \cdots & \mathbf{L}_{N,M}^H \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_N \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{1,1}^H\mathbf{y}_1 + \mathbf{L}_{2,1}^H\mathbf{y}_2 + \mathbf{L}_{N,1}^H\mathbf{y}_N \\ \mathbf{L}_{1,2}^H\mathbf{y}_1 + \mathbf{L}_{2,2}^H\mathbf{y}_2 + \mathbf{L}_{N,2}^H\mathbf{y}_N \\ \vdots \\ \mathbf{L}_{1,M}^H\mathbf{y}_1 + \mathbf{L}_{2,M}^H\mathbf{y}_2 + \mathbf{L}_{N,M}^H\mathbf{y}_N \end{bmatrix}$$

Attributes

shape [tuple] Operator shape

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

Examples using `pylops.Block`

- *Describe*
- *Operators concatenation*
- *Operators with Multiprocessing*

pylops.BlockDiag

class `pylops.BlockDiag` (*ops, nproc=1, 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.

nproc [int, optional] Number of processes used to evaluate the N operators in parallel using multiprocessing. If `nproc=1`, work in serial mode.

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

$$\begin{bmatrix} \mathbf{L}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{L}_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{L}_N \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_N \end{bmatrix} = \begin{bmatrix} \mathbf{L}_1 \mathbf{x}_1 \\ \mathbf{L}_2 \mathbf{x}_2 \\ \vdots \\ \mathbf{L}_N \mathbf{x}_N \end{bmatrix}$$

while its application in adjoint mode leads to

$$\begin{bmatrix} \mathbf{L}_1^H & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{L}_2^H & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{L}_N^H \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_N \end{bmatrix} = \begin{bmatrix} \mathbf{L}_1^H \mathbf{y}_1 \\ \mathbf{L}_2^H \mathbf{y}_2 \\ \vdots \\ \mathbf{L}_N^H \mathbf{y}_N \end{bmatrix}$$

Attributes

shape [tuple] Operator shape

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

Methods

<code>__init__(ops[, nproc, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.BlockDiag`

- *Operators concatenation*
- *Operators with Multiprocessing*

`pylops.Kronecker`

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{Op}_1 and \mathbf{Op}_2 of sizes $[n_1 \times m_1]$ and $[n_2 \times m_2]$ respectively, resulting in a block matrix of size $[n_1 n_2 \times m_1 m_2]$.

$$\mathbf{Op}_1 \otimes \mathbf{Op}_2 = \begin{bmatrix} \mathbf{Op}_1^{1,1} \mathbf{Op}_2 & \dots & \mathbf{Op}_1^{1,m_1} \mathbf{Op}_2 \\ \vdots & \ddots & \vdots \\ \mathbf{Op}_1^{n_1,1} \mathbf{Op}_2 & \dots & \mathbf{Op}_1^{n_1,m_1} \mathbf{Op}_2 \end{bmatrix}$$

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{Op}_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

<code>__init__(Op1, Op2[, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{Ax}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Kronecker`

- *Operators concatenation*

`pylops.Real`

class `pylops.Real` (*dims*, *dtype*='complex128')

Real operator.

Return the real component of the input. The adjoint returns a complex number with the same real component as the input and zero imaginary component.

Parameters

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

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

Notes

In forward mode:

$$y_i = \Re\{x_i\} \quad \forall i = 0, \dots, N - 1$$

In adjoint mode:

$$x_i = \Re\{y_i\} + 0i \quad \forall i = 0, \dots, N - 1$$

Attributes

shape [tuple] Operator shape

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

Methods

<code>__init__(dims[, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Real`

- *Real*

`pylops.Imag`

class `pylops.Imag` (*dims*, *dtype*=`'complex128'`)

Imag operator.

Return the imaginary component of the input as a real value. The adjoint returns a complex number with zero real component and the imaginary component set to the real component of the input.

Parameters

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

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

Notes

In forward mode:

$$y_i = \Im\{x_i\} \quad \forall i = 0, \dots, N - 1$$

In adjoint mode:

$$x_i = 0 + i\Re\{y_i\} \quad \forall i = 0, \dots, N - 1$$

Attributes

shape [tuple] Operator shape

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

Methods

<code>__init__(dims[, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Imag`

- *Imag*

`pylops.Conj`

class `pylops.Conj` (*dims*, *dtype*='complex128')

Complex conjugate operator.

Return the complex conjugate of the input. It is self-adjoint.

Parameters

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

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

Notes

In forward mode:

$$y_i = \Re\{x_i\} - i\Im\{x_i\} \quad \forall i = 0, \dots, N-1$$

In adjoint mode:

$$x_i = \Re\{y_i\} - i\Im\{y_i\} \quad \forall i = 0, \dots, N-1$$

Attributes

shape [tuple] Operator shape

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

Methods

<code>__init__(dims[, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.Conj`

- *Conj*

Smoothing and derivatives

<i>Smoothing1D</i> (nsmooth, dims[, dir, dtype])	1D Smoothing.
<i>Smoothing2D</i> (nsmooth, dims[, nodir, dtype])	2D Smoothing.
<i>FirstDerivative</i> (N[, dims, dir, sampling, ...])	First derivative.
<i>SecondDerivative</i> (N[, dims, dir, sampling, ...])	Second derivative.
<i>Laplacian</i> (dims[, dirs, weights, sampling, ...])	Laplacian.
<i>Gradient</i> (dims[, sampling, edge, dtype, kind])	Gradient.
<i>FirstDirectionalDerivative</i> (dims, v[, ...])	First Directional derivative.

Continued on next page

Table 27 – continued from previous page

<i>SecondDirectionalDerivative</i> (dims, v[, ...])	Second Directional derivative.
---	--------------------------------

pylops.Smoothing1D`pylops.Smoothing1D (nsmooth, dims, dir=0, dtype='float64')`

1D Smoothing.

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_{smooth} :

$$\mathbf{f} = [1/n_{\text{smooth}}, 1/n_{\text{smooth}}, \dots, 1/n_{\text{smooth}}]$$

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:

$$y[i, j, k] = 1/n_{\text{smooth}} \sum_{l=-(n_{\text{smooth}}-1)/2}^{(n_{\text{smooth}}-1)/2} x[i, l, k]$$

and the third direction:

$$y[i, j, k] = 1/n_{\text{smooth}} \sum_{l=-(n_{\text{smooth}}-1)/2}^{(n_{\text{smooth}}-1)/2} x[i, j, l]$$

Note that since the filter is symmetrical, the *Smoothing1D* 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.Smoothing1D`

- *1D Smoothing*
- *Causal Integration*
- *Wavelet estimation*
- *03. Solvers*

pylops.Smoothing2D

`pylops.Smoothing2D` (*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 *nodir*.

Parameters

- nsmooth** [`tuple` or `list`] Length of smoothing operator in 1st and 2nd dimensions (must be odd)
- dims** [`tuple`] Number of samples for each dimension
- nodir** [`int`, optional] Direction along which smoothing is **not** applied (set to `None` for 2d arrays)
- dtype** [`str`, optional] Type of elements in input array.

See also:

`pylops.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:

$$y[i, j] = 1/(n_{\text{smooth},1} \cdot n_{\text{smooth},2}) \sum_{l=-(n_{\text{smooth},1}-1)/2}^{(n_{\text{smooth},1}-1)/2} \sum_{m=-(n_{\text{smooth},2}-1)/2}^{(n_{\text{smooth},2}-1)/2} x[l, m]$$

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`

- [2D Smoothing](#)
- [Causal Integration](#)

pylops.FirstDerivative

`class pylops.FirstDerivative` (*N*, *dims=None*, *dir=0*, *sampling=1.0*, *edge=False*, *dtype='float64'*,
kind='centered')

First derivative.

Apply a first derivative using a three-point stencil finite-difference approximation.

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 the derivative is applied.

sampling [`float`, optional] Sampling step Δx .

edge [`bool`, optional] Use reduced order derivative at edges (`True`) or ignore them (`False`) for centered derivative

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:

$$y[i] = (0.5x[i + 1] - 0.5x[i - 1])/\Delta x$$

while the first-order forward stencil is:

$$y[i] = (x[i + 1] - x[i])/\Delta x$$

and the first-order backward stencil is:

$$y[i] = (x[i] - x[i - 1])/\Delta x$$

Attributes

shape [`tuple`] Operator shape

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

Methods

<code>__init__(N[, dims, dir, sampling, edge, ...])</code>	Initialize this <code>LinearOperator</code> .
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator

Continued on next page

Table 28 – continued from previous page

<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.FirstDerivative`

- *Causal Integration*
- *Derivatives*
- *Operators concatenation*
- *Total Variation (TV) Regularization*
- *05. Image deblurring*
- *16. CT Scan Imaging*

`pylops.SecondDerivative`

class `pylops.SecondDerivative` (*N*, *dims=None*, *dir=0*, *sampling=1*, *edge=False*, *dtype='float64'*, *kind='centered'*)

Second derivative.

Apply a second derivative using a three-point stencil finite-difference approximation.

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 the derivative is applied.

sampling [`float`, optional] Sampling step Δx .

edge [`bool`, optional] Use reduced order derivative at edges (`True`) or ignore them (`False`) for centered derivative

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

kind [`str`, optional] Derivative kind (`forward`, `centered`, or `backward`).

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:

$$y[i] = (x[i + 1] - 2x[i] + x[i - 1])/\Delta x^2$$

while the second-order forward stencil is:

$$y[i] = (x[i + 2] - 2x[i + 1] + x[i])/\Delta x^2$$

and the second-order backward stencil is:

$$y[i] = (x[i] - 2x[i - 1] + x[i - 2])/\Delta x^2$$

Attributes

shape [tuple] Operator shape

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

Methods

<code>__init__(N[, dims, dir, sampling, edge, ...])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.SecondDerivative`

- *Causal Integration*
- *Derivatives*
- *Operators concatenation*
- *Total Variation (TV) Regularization*
- *Wavelet estimation*
- *03. Solvers*
- *12. Seismic regularization*

`pylops.Laplacian`

`pylops.Laplacian` (*dims*, *dircs*=(0, 1), *weights*=(1, 1), *sampling*=(1, 1), *edge*=False, *dtype*='float64',
kind='centered')

Laplacian.

Apply second-order centered Laplacian operator to a multi-dimensional array.

Note: At least 2 dimensions are required, use `pylops.SecondDerivative` for 1d arrays.

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`) for centered derivative

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

kind [`str`, optional] Derivative kind (`forward`, `centered`, or `backward`)

Returns

l2op [`pylops.LinearOperator`] Laplacian linear operator

Raises

ValueError If `dirs`, `weights`, and `sampling` do not have the same size

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:

$$y[i, j] = (x[i + 1, j] + x[i - 1, j] + x[i, j - 1] + x[i, j + 1] - 4x[i, j]) / (\Delta x \Delta y)$$

Examples using `pylops.Laplacian`

- *Bilinear Interpolation*
- *Causal Integration*
- *Derivatives*
- *06. 2D Interpolation*
- *16. CT Scan Imaging*

pylops.Gradient

`pylops.Gradient` (`dims`, `sampling=1`, `edge=False`, `dtype='float64'`, `kind='centered'`)
Gradient.

Apply gradient operator to a multi-dimensional array.

Note: At least 2 dimensions are required, use `pylops.FirstDerivative` for 1d arrays.

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

l2op [`pylops.LinearOperator`] Gradient linear operator

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:

$$\mathbf{g}_{i,j,k} = (f_{i+1,j,k} - f_{i-1,j,k})/d_1 \mathbf{i}_1 + (f_{i,j+1,k} - f_{i,j-1,k})/d_2 \mathbf{i}_2 + (f_{i,j,k+1} - f_{i,j,k-1})/d_3 \mathbf{i}_3$$

which is discretized as follows:

$$\mathbf{g} = \begin{bmatrix} \mathbf{df}_1 \\ \mathbf{df}_2 \\ \mathbf{df}_3 \end{bmatrix}$$

In adjoint mode, the adjoints of the first derivatives along different axes are instead summed together.

Examples using `pylops.Gradient`

- *Derivatives*

`pylops.FirstDirectionalDerivative`

`pylops.FirstDirectionalDerivative` (*dims*, *v*, *sampling=1*, *edge=False*, *dtype='float64'*,
kind='centered')

First Directional derivative.

Apply a directional derivative operator to a multi-dimensional array along either a single common direction or different directions for each point of the array.

Note: At least 2 dimensions are required, consider using `pylops.FirstDerivative` for 1d arrays.

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 $[n_{\text{dims}} \times n_{d_0} \times \dots \times n_{d_{n_{\text{dims}}}}]$)

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.

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 \mathbf{v} :

$$df_{\mathbf{v}} = \nabla f \mathbf{v}$$

or along the directions defined by the unitary vectors $\mathbf{v}(x, y)$:

$$df_{\mathbf{v}}(x, y) = \nabla f(x, y) \mathbf{v}(x, y)$$

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 \mathbf{v} along the main diagonal.

Examples using `pylops.FirstDirectionalDerivative`

- *Derivatives*

`pylops.SecondDirectionalDerivative`

`pylops.SecondDirectionalDerivative` (*dims*, *v*, *sampling=1*, *edge=False*, *dtype='float64'*)
Second Directional derivative.

Apply a second directional derivative operator to a multi-dimensional array along either a single common direction or different directions for each point of the array.

Note: At least 2 dimensions are required, consider using *pylops.SecondDerivative* for 1d arrays.

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 $[n_{\text{dims}} \times n_{d_0} \times \dots \times n_{d_{n_{\text{dims}}}}]$)

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 \mathbf{v} :

$$d^2 f_{\mathbf{v}} = -D_{\mathbf{v}}^T [D_{\mathbf{v}} f]$$

where $D_{\mathbf{v}}$ 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`

- *Derivatives*

Signal processing

<code>Convolve1D(N, h[, offset, dims, dir, dtype, ...])</code>	1D convolution operator.
<code>Convolve2D(N, h, dims[, offset, nodir, ...])</code>	2D convolution operator.
<code>ConvolveND(N, h, dims[, offset, dirs, ...])</code>	ND convolution operator.
<code>Interp(M, iava[, dims, dir, kind, dtype])</code>	Interpolation operator.
<code>Bilinear(iava, dims[, dtype])</code>	Bilinear interpolation operator.
<code>FFT(dims[, dir, nfft, sampling, norm, real, ...])</code>	One dimensional Fast-Fourier Transform.
<code>FFT2D(dims[, dirs, nffts, sampling, norm, ...])</code>	Two dimensional Fast-Fourier Transform.
<code>FFTND(dims[, dirs, nffts, sampling, norm, ...])</code>	N-dimensional Fast-Fourier Transform.
<code>Shift(dims, shift[, dir, nfft, sampling, ...])</code>	Shift operator
<code>DWT(dims[, dir, wavelet, level, dtype])</code>	One dimensional Wavelet operator.
<code>DWT2D(dims[, dirs, wavelet, level, dtype])</code>	Two dimensional Wavelet operator.
<code>Seislet(slopes[, sampling, level, kind, ...])</code>	Two dimensional Seislet operator.
<code>Radon2D(taxis, haxis, paxis[, kind, ...])</code>	Two dimensional Radon transform.
<code>Radon3D(taxis, hyaxis, hxaxis, pyaxis, paxis)</code>	Three dimensional Radon transform.
<code>ChirpRadon2D(taxis, haxis, pmax[, dtype])</code>	2D Chirp Radon transform
<code>ChirpRadon3D(taxis, hyaxis, hxaxis, pmax[, ...])</code>	3D Chirp Radon transform
<code>Sliding1D(Op, dim, dimd, nwin, nover[, ...])</code>	1D Sliding transform operator.
<code>Sliding2D(Op, dims, dimsd, nwin, nover[, ...])</code>	2D Sliding transform operator.
<code>Sliding3D(Op, dims, dimsd, nwin, nover, nop)</code>	3D Sliding transform operator.
<code>Patch2D(Op, dims, dimsd, nwin, nover, nop[, ...])</code>	2D Patch transform operator.
<code>Fredholm1(G[, nz, saveGt, usematmul, dtype])</code>	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.

$$Y(f) = F(h(t)) * F(x(t))$$

`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:

$$y(t) = F^{-1}(H(f)^* * X(f))$$

Attributes

shape [`tuple`] Operator shape

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

Methods

<code>__init__(N, h[, offset, dims, dir, dtype, ...])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.signalprocessing.Convolve1D`

- *1D Smoothing*
- *Convolution*
- *MP, OMP, ISTA and FISTA*
- *Wavelet estimation*
- *03. Solvers*
- *04. Bayesian Inversion*

`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] Indices 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:

$$y(t, x) = \iint_{-\infty}^{\infty} h(t - \tau, x - \chi) d(\tau, \chi) d\tau d\chi$$

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.

$$Y(f, k_x) = F(h(t, x)) * F(d(t, x))$$

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:

$$y(t, x) = \iint_{-\infty}^{\infty} h(t + \tau, x + \chi) d(\tau, \chi) d\tau d\chi$$

or in frequency domain:

$$y(t, x) = F^{-1}(H(f, k_x)^* * X(f, k_x))$$

Examples using `pylops.signalprocessing.Convolve2D`

- *Convolution*
- *05. Image deblurring*

`pylops.signalprocessing.ConvolveND`

class `pylops.signalprocessing.ConvolveND` (*N*, *h*, *dims*, *offset=None*, *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 $d(x_1, x_2, \dots, x_N)$ and a compact filter kernel $h(x_1, x_2, \dots, x_N)$ 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

<code>__init__(N, h, dims[, offset, dirs, method, ...])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.signalprocessing.ConvolveND`

- *2D Smoothing*
- *Causal Integration*
- *Convolution*

- 05. Image deblurring

pylops.signalprocessing.Interp

`pylops.signalprocessing.Interp` (*M*, *iava*, *dims=None*, *dir=0*, *kind='linear'*, *dtype='float64'*)

Interpolation operator.

Apply interpolation along direction *dir* from regularly sampled input vector into fractionary positions *iava* using one of the following algorithms:

- *Nearest neighbour* interpolation is a thin wrapper around `pylops.Restriction` at `np.round(iava)` locations.
- *Linear interpolation* extracts values from input vector at locations `np.floor(iava)` and `np.floor(iava)+1` and linearly combines them in forward mode, places weighted versions of the interpolated values at locations `np.floor(iava)` and `np.floor(iava)+1` in an otherwise zero vector in adjoint mode.
- *Sinc interpolation* performs sinc interpolation at locations *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 *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 *linear interpolation* as those values are forced to be just before this element.

Parameters

M [*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 $M-1-\text{eps}$)

Raises

ValueError If the vector *iava* contains repeated values.

NotImplementedError If *kind* is not *nearest*, *linear* or *sinc*

See also:

`pylops.Restriction` Restriction operator

Notes

Linear interpolation of a subset of N values at locations `iava` from an input (or model) vector \mathbf{x} of size M can be expressed as:

$$y_i = (1 - w_i)x_{l_i^l} + w_i x_{l_i^r} \quad \forall i = 1, 2, \dots, N$$

where $\mathbf{l}^l = [\lfloor l_1 \rfloor, \lfloor l_2 \rfloor, \dots, \lfloor l_N \rfloor]$ and $\mathbf{l}^r = [\lfloor l_1 \rfloor + 1, \lfloor l_2 \rfloor + 1, \dots, \lfloor l_N \rfloor + 1]$ are vectors containing the indices of the original array at which samples are taken, and $\mathbf{w} = [l_1 - \lfloor l_1 \rfloor, l_2 - \lfloor l_2 \rfloor, \dots, l_N - \lfloor l_N \rfloor]$ 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 \mathbf{x} of size M can be expressed as:

$$\text{sincy}_i = \sum_{j=0}^M x_j(i - j) \quad \forall i = 1, 2, \dots, N$$

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`

- *Restriction and Interpolation*

`pylops.signalprocessing.Bilinear`

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 $[2 \times n_{\text{ava}}]$ 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 \mathbf{x} of size $[m_1 \times m_2 \times \dots \times m_{\text{ndim}}]$ can be expressed as:

$$y_i = (1 - w_i^0)(1 - w_i^1)x_{l_i^{l,0}, l_i^{l,1}} + w_i^0(1 - w_i^1)x_{l_i^{r,0}, l_i^{l,1}} + (1 - w_i^0)w_i^1x_{l_i^{l,0}, l_i^{r,1}} + w_i^0w_i^1x_{l_i^{r,0}, l_i^{r,1}} \quad \forall i = 1, 2, \dots, M$$

where $\mathbf{l}^{1,0} = [\lfloor l_1^0 \rfloor, \lfloor l_2^0 \rfloor, \dots, \lfloor l_N^0 \rfloor]$, $\mathbf{l}^{1,1} = [\lfloor l_1^1 \rfloor, \lfloor l_2^1 \rfloor, \dots, \lfloor l_N^1 \rfloor]$, $\mathbf{l}^{r,0} = [\lfloor l_1^0 \rfloor + 1, \lfloor l_2^0 \rfloor + 1, \dots, \lfloor l_N^0 \rfloor + 1]$, $\mathbf{l}^{r,1} = [\lfloor l_1^1 \rfloor + 1, \lfloor l_2^1 \rfloor + 1, \dots, \lfloor l_N^1 \rfloor + 1]$, are vectors containing the indices of the original array at which samples are taken, and $\mathbf{w}^j = [l_1^i - \lfloor l_1^i \rfloor, l_2^i - \lfloor l_2^i \rfloor, \dots, l_N^i - \lfloor l_N^i \rfloor]$ ($\forall j = 0, 1$) 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

<code>__init__(iava, dims[, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.signalprocessing.Bilinear`

- *Bilinear Interpolation*

`pylops.signalprocessing.FFT`

`pylops.signalprocessing.FFT(dims, dir=0, nfft=None, sampling=1.0, norm='ortho', real=False, fftshift=None, fftshift_before=None, fftshift_after=False, engine='numpy', dtype='complex128', **kwargs_fft)`

One dimensional Fast-Fourier Transform.

Apply Fast-Fourier Transform (FFT) along a specific direction `dir` of a multi-dimensional array of size `dim`.

Using the default NumPy engine, 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 `numpy.fft.ifft` (or `numpy.fft.irfft` for real models) in adjoint mode, or their CuPy equivalents. When `engine='fftw'` is chosen, the `pyfftw.FFTW` class is used instead. Alternatively, when the SciPy engine is chosen, the overloads are of `scipy.fft.fft` (or `scipy.fft.rfft` for real models) in forward mode, and to `scipy.fft.ifft` (or `scipy.fft.irfft` for real models) in adjoint mode.

When using `real=True`, the result of the forward is also multiplied by $\sqrt{2}$ for all frequency bins except zero and Nyquist, and the input of the adjoint is multiplied by $1/\sqrt{2}$ for the same frequencies.

For a real valued input signal, it is advised to use the flag `real=True` as it stores the values of the Fourier transform at positive frequencies only as values at negative frequencies are simply their complex conjugates.

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`.

norm [`{“ortho”, “none”, “1/n”}`, optional] New in version 1.17.0.

- “ortho”: Scales forward and adjoint FFT transforms with $1/\sqrt{N_F}$, where N_F is the number of samples in the Fourier domain given by `nfft`.
- “none”: Does not scale the forward or the adjoint FFT transforms.
- “1/n”: Scales both the forward and adjoint FFT transforms by $1/N_F$.

Note: For “none” and “1/n”, the operator is not unitary, that is, the adjoint is not the inverse. To invert the operator, simply use `Op \ y`.

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] Deprecated since version 1.17.0: Use `ifftshift_before`.

ifftshift_before [`bool`, optional] New in version 1.17.0.

Apply `ifftshift` (`True`) or not (`False`) to model vector (before FFT). Consider using this option when the model vector’s respective axis is symmetric with respect to the zero value sample. This will shift the zero value sample to coincide with the zero index sample. With such an arrangement, FFT will not introduce a sample-dependent phase-shift when compared to the continuous Fourier Transform. Defaults to not applying `ifftshift`.

fftshift_after [`bool`, optional] New in version 1.17.0.

Apply `fftshift` (`True`) or not (`False`) to data vector (after FFT). Consider using this option when you require frequencies to be arranged naturally, from negative to positive. When not applying `fftshift` after FFT, frequencies are arranged from zero to largest positive, and then from negative Nyquist to the frequency bin before zero.

engine [`str`, optional] Engine used for fft computation (`numpy`, `fftw`, or `scipy`). Choose `numpy` when working with `cupy` arrays.

Note: Since version 1.17.0, accepts “scipy”.

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. In addition, note that neither the NumPy nor the FFTW backends supports returning `dtype` different than `complex128`. As such, when using either backend, arrays will be force-casted to types corresponding to the supplied `dtype`. The SciPy backend supports all precisions natively. Under all backends, when a real `dtype` is supplied, a real result will be enforced on the result of the `rmatvec` and the input of the `matvec`.

****kwargs_fftw** Arbitrary keyword arguments for `pyfftw.FFTW`

Raises**ValueError**

- If `dims` is provided and `dir` is bigger than `len(dims)`.
- If `norm` is not one of “ortho”, “none”, or “1/n”.

NotImplementedError If `engine` is neither `numpy`, `fftw`, nor `scipy`.

See also:

FFT2D Two-dimensional FFT

FFTND N-dimensional FFT

Notes

The FFT operator (using `norm="ortho"`) applies the forward Fourier transform to a signal $d(t)$ in forward mode:

$$D(f) = F(d) = \frac{1}{\sqrt{N_F}} \int_{-\infty}^{\infty} d(t) e^{-j2\pi ft} dt$$

Similarly, the inverse Fourier transform is applied to the Fourier spectrum $D(f)$ in adjoint mode:

$$d(t) = F^{-1}(D) = \frac{1}{\sqrt{N_F}} \int_{-\infty}^{\infty} D(f) e^{j2\pi ft} df$$

where N_F is the number of samples in the Fourier domain `nfft`. Both operators are effectively discretized and solved by a fast iterative algorithm known as Fast Fourier Transform. Note that the FFT operator (using `norm="ortho"`) is a special operator in that the adjoint is also the inverse of the forward mode. For other norms, this does not hold (see `norm` help). However, for any norm, the Fourier transform is Hermitian for real input signals.

Attributes

dims_fft [tuple] Shape of the array after the forward, but before linearization.

For example, `y_reshaped = (Op * x.ravel()).reshape(Op.dims_fft)`.

f [numpy.ndarray] Discrete Fourier Transform sample frequencies

real [bool] When True, uses `rfft/irfft`

rdtype [bool] Expected input type to the forward

cdtype [bool] Output type of the forward. Complex equivalent to `rdtype`.

shape [tuple] Operator shape

cllinear [bool] New in version 1.17.0.

Operator is complex-linear. Is false when either `real=True` or when `dtype` is not a complex type.

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

Examples using `pylops.signalprocessing.FFT`

- *1D, 2D and 3D Sliding*
- *Fourier Transform*
- *02. The Dot-Test*
- *03. Solvers*
- *04. Bayesian Inversion*

`pylops.signalprocessing.FFT2D`

```
pylops.signalprocessing.FFT2D(dims, dirs=(0, 1), nffts=None, sampling=1.0, norm='ortho',
                              real=False, ifftshift_before=False, ifftshift_after=False,
                              dtype='complex128', engine='numpy')
```

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`.

Using the default NumPy engine, the FFT operator is an overload to either the NumPy `numpy.fft.fft2` (or `numpy.fft.rfft2` for real models) in forward mode, and to `numpy.fft.ifft2` (or `numpy.fft.irfft2` for real models) in adjoint mode, or their CuPy equivalents. Alternatively, when the SciPy engine is chosen, the overloads are of `scipy.fft.fft2` (or `scipy.fft.rfft2` for real models) in forward mode, and to `scipy.fft.ifft2` (or `scipy.fft.irfft2` for real models) in adjoint mode.

When using `real=True`, the result of the forward is also multiplied by $\sqrt{2}$ for all frequency bins except zero and Nyquist, and the input of the adjoint is multiplied by $1/\sqrt{2}$ for the same frequencies.

For a real valued input signal, it is advised to use the flag `real=True` as it stores the values of the Fourier transform of the last direction at positive frequencies only as values at negative frequencies are simply their complex conjugates.

Parameters

dims [tuple] Number of samples for each dimension

dirs [tuple, optional] Pair of directions along which FFT2D is applied

nffts [tuple or int, optional] Number of samples in Fourier Transform for each direction. In case only one dimension needs to be specified, use `None` for the other dimension in the tuple. The direction with `None` will use `dims[dir]` as `nfft`. When supplying a tuple, the order must agree with that of `dirs`. When a single value is passed, it will be used for both directions. As such the default is equivalent to `nffts=(None, None)`.

sampling [tuple or float, optional] Sampling steps for each direction. When supplied a single value, it is used for both directions. Unlike `nffts`, any `None` will not be converted to the default value.

norm [{"ortho", "none", "1/n"}, optional] New in version 1.17.0.

- "ortho": Scales forward and adjoint FFT transforms with $1/\sqrt{N_F}$, where N_F is the number of samples in the Fourier domain given by product of all elements of `nffts`.
- "none": Does not scale the forward or the adjoint FFT transforms.
- "1/n": Scales both the forward and adjoint FFT transforms by $1/N_F$.

Note: For “none” and “1/n”, the operator is not unitary, that is, the adjoint is not the inverse. To invert the operator, simply use $Op \setminus y$.

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. Note that the real FFT is applied only to the first dimension to which the FFT2D operator is applied (last element of `dirs`)

ifftshift_before [`tuple` or `bool`, optional] Apply ifftshift (`True`) or not (`False`) to model vector (before FFT). Consider using this option when the model vector’s respective axis is symmetric with respect to the zero value sample. This will shift the zero value sample to coincide with the zero index sample. With such an arrangement, FFT will not introduce a sample-dependent phase-shift when compared to the continuous Fourier Transform. When passing a single value, the shift will be the same for every direction. Pass a tuple to specify which dimensions are shifted.

fftshift_after [`tuple` or `bool`, optional] Apply fftshift (`True`) or not (`False`) to data vector (after FFT). Consider using this option when you require frequencies to be arranged naturally, from negative to positive. When not applying fftshift after FFT, frequencies are arranged from zero to largest positive, and then from negative Nyquist to the frequency bin before zero. When passing a single value, the shift will be the same for every direction. Pass a tuple to specify which dimensions are shifted.

engine [`str`, optional] New in version 1.17.0.

Engine used for fft computation (`numpy` or `scipy`).

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. In addition, note that the NumPy backend does not support returning `dtype` different than `complex128`. As such, when using the NumPy backend, arrays will be force-casted to types corresponding to the supplied `dtype`. The SciPy backend supports all precisions natively. Under both backends, when a real `dtype` is supplied, a real result will be enforced on the result of the `rmatvec` and the input of the `matvec`.

Raises

ValueError

- If `dims` has less than two elements.
- If `dirs` does not have exactly two elements.
- If `nffts` or `sampling` are not either a single value or a tuple with two elements.
- If `norm` is not one of “ortho”, “none”, or “1/n”.

NotImplementedError If `engine` is neither `numpy`, nor `scipy`.

See also:

FFT One-dimensional FFT

FFTND N-dimensional FFT

Notes

The FFT2D operator (using `norm="ortho"`) applies the two-dimensional forward Fourier transform to a signal $d(y, x)$ in forward mode:

$$D(k_y, k_x) = F(d) = \frac{1}{\sqrt{N_F}} \iint_{-\infty}^{\infty} d(y, x) e^{-j2\pi k_y y} e^{-j2\pi k_x x} dy dx$$

Similarly, the two-dimensional inverse Fourier transform is applied to the Fourier spectrum $D(k_y, k_x)$ in adjoint mode:

$$d(y, x) = F^{-1}(D) = \frac{1}{\sqrt{N_F}} \iint_{-\infty}^{\infty} D(k_y, k_x) e^{j2\pi k_y y} e^{j2\pi k_x x} dk_y dk_x$$

where N_F is the number of samples in the Fourier domain given by the product of the element of `nffts`. Both operators are effectively discretized and solved by a fast iterative algorithm known as Fast Fourier Transform. Note that the FFT2D operator (using `norm="ortho"`) is a special operator in that the adjoint is also the inverse of the forward mode. For other norms, this does not hold (see `norm help`). However, for any norm, the 2D Fourier transform is Hermitian for real input signals.

Attributes

dims_fft [`tuple`] Shape of the array after the forward, but before linearization.

For example, `y_reshaped = (Op * x.ravel()).reshape(Op.dims_fft)`.

f1 [`numpy.ndarray`] Discrete Fourier Transform sample frequencies along `dir[0]`

f2 [`numpy.ndarray`] Discrete Fourier Transform sample frequencies along `dir[1]`

real [`bool`] When `True`, uses `rfft2/irfft2`

rdtype [`bool`] Expected input type to the forward

cdtype [`bool`] Output type of the forward. Complex equivalent to `rdtype`.

shape [`tuple`] Operator shape

clinear [`bool`] New in version 1.17.0.

Operator is complex-linear. Is false when either `real=True` or when `dtype` is not a complex type.

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

Examples using `pylops.signalprocessing.FFT2D`

- *2D Patching*
- *Fourier Transform*
- *Total Variation (TV) Regularization*
- *12. Seismic regularization*
- *14. Seismic wavefield decomposition*

pylops.signalprocessing.FFTND

```
pylops.signalprocessing.FFTND (dims, dirs=(0, 1, 2), nffts=None, sampling=1.0, norm='ortho',
                               real=False, ifftshift_before=False, fftshift_after=False,
                               dtype='complex128', engine='scipy')
```

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`.

Using the default NumPy engine, the FFT operator is an overload to either the NumPy `numpy.fft.fftn` (or `numpy.fft.rfftn` for real models) in forward mode, and to `numpy.fft.ifftn` (or `numpy.fft.irfftn` for real models) in adjoint mode, or their CuPy equivalents. Alternatively, when the SciPy engine is chosen, the overloads are of `scipy.fft.fftn` (or `scipy.fft.rfftn` for real models) in forward mode, and to `scipy.fft.ifftn` (or `scipy.fft.irfftn` for real models) in adjoint mode.

When using `real=True`, the result of the forward is also multiplied by $\sqrt{2}$ for all frequency bins except zero and Nyquist along the last direction of `dirs`, and the input of the adjoint is multiplied by $1/\sqrt{2}$ for the same frequencies.

For a real valued input signal, it is advised to use the flag `real=True` as it stores the values of the Fourier transform of the last direction at positive frequencies only as values at negative frequencies are simply their complex conjugates.

Parameters

dims [tuple] Number of samples for each dimension

dirs [tuple or int, optional] Direction(s) along which FFTND is applied

nffts [tuple or int, optional] Number of samples in Fourier Transform for each direction.
In case only one dimension needs to be specified, use `None` for the other dimension in the tuple. The direction with `None` will use `dims[dir]` as `nfft`. When supplying a tuple, the order must agree with that of `dirs`. When a single value is passed, it will be used for both directions. As such the default is equivalent to `nffts=(None, ..., None)`.

sampling [tuple or float, optional] Sampling steps for each direction. When supplied a single value, it is used for all directions. Unlike `nffts`, any `None` will not be converted to the default value.

norm [{"ortho", "none", "1/n"}, optional] New in version 1.17.0.

- “ortho”: Scales forward and adjoint FFT transforms with $1/\sqrt{N_F}$, where N_F is the number of samples in the Fourier domain given by product of all elements of `nffts`.
- “none”: Does not scale the forward or the adjoint FFT transforms.
- “1/n”: Scales both the forward and adjoint FFT transforms by $1/N_F$.

Note: For “none” and “1/n”, the operator is not unitary, that is, the adjoint is not the inverse. To invert the operator, simply use `Op \ y`.

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. Note that the real FFT is applied only to the first dimension to which the FFTND operator is applied (last element of `dirs`)

ifftshift_before [tuple or bool, optional] New in version 1.17.0.

Apply `fftshift` (`True`) or not (`False`) to model vector (before FFT). Consider using this option when the model vector's respective axis is symmetric with respect to the zero value sample. This will shift the zero value sample to coincide with the zero index sample. With such an arrangement, FFT will not introduce a sample-dependent phase-shift when compared to the continuous Fourier Transform. When passing a single value, the shift will be the same for every direction. Pass a tuple to specify which dimensions are shifted.

fftshift_after [`tuple` or `bool`, optional] New in version 1.17.0.

Apply `fftshift` (`True`) or not (`False`) to data vector (after FFT). Consider using this option when you require frequencies to be arranged naturally, from negative to positive. When not applying `fftshift` after FFT, frequencies are arranged from zero to largest positive, and then from negative Nyquist to the frequency bin before zero. When passing a single value, the shift will be the same for every direction. Pass a tuple to specify which dimensions are shifted.

engine [`str`, optional] New in version 1.17.0.

Engine used for fft computation (`numpy` or `scipy`).

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. In addition, note that the NumPy backend does not support returning `dtype` different than `complex128`. As such, when using the NumPy backend, arrays will be force-cast to types corresponding to the supplied `dtype`. The SciPy backend supports all precisions natively. Under both backends, when a real `dtype` is supplied, a real result will be enforced on the result of the `rmatvec` and the input of the `matvec`.

Raises

ValueError

- If `nffts` or `sampling` are not either a single value or tuple with the same dimension `dirs`.
- If `norm` is not one of "ortho", "none", or "1/n".

NotImplementedError If `engine` is neither `numpy`, nor `scipy`.

See also:

FFT One-dimensional FFT

FFT2D Two-dimensional FFT

Notes

The FFTND operator (using `norm="ortho"`) applies the N-dimensional forward Fourier transform to a multi-dimensional array. Considering an N-dimensional signal $d(x_1, \dots, x_N)$. The FFTND in forward mode is:

$$D(k_1, \dots, k_N) = F(d) = \frac{1}{\sqrt{N_F}} \int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} d(x_1, \dots, x_N) e^{-j2\pi k_1 x_1} \dots e^{-j2\pi k_N x_N} dx_1 \dots dx_N$$

Similarly, the three-dimensional inverse Fourier transform is applied to the Fourier spectrum $D(k_z, k_y, k_x)$ in adjoint mode:

$$d(x_1, \dots, x_N) = F^{-1}(D) = \frac{1}{\sqrt{N_F}} \int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} D(k_1, \dots, k_N) e^{-j2\pi k_1 x_1} \dots e^{-j2\pi k_N x_N} dk_1 \dots dk_N$$

where N_F is the number of samples in the Fourier domain given by the product of the element of `nffts`. Both operators are effectively discretized and solved by a fast iterative algorithm known as Fast Fourier Transform. Note that the FFTND operator (using `norm="ortho"`) is a special operator in that the adjoint is also the inverse of the forward mode. For other norms, this does not hold (see `norm` help). However, for any norm, the N-dimensional Fourier transform is Hermitian for real input signals.

Attributes

dims_fft [tuple] Shape of the array after the forward, but before linearization.

For example, `y_reshaped = (Op * x.ravel()).reshape(Op.dims_fft)`.

fs [tuple] Each element of the tuple corresponds to the Discrete Fourier Transform sample frequencies along the respective direction given by `dirs`.

real [bool] When True, uses `rfftn/irfftn`

rdtype [bool] Expected input type to the forward

cdtype [bool] Output type of the forward. Complex equivalent to `rdtype`.

shape [tuple] Operator shape

clinear [bool] New in version 1.17.0.

Operator is complex-linear. Is false when either `real=True` or when `dtype` is not a complex type.

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

Examples using `pylops.signalprocessing.FFTND`

- *Fourier Transform*

`pylops.signalprocessing.Shift`

`pylops.signalprocessing.Shift` (*dims*, *shift*, *dir*=0, *nfft*=None, *sampling*=1.0, *real*=False, *engine*='numpy', *dtype*='complex128', **kwargs_fft)

Shift operator

Apply fractional shift in the frequency domain along a specific direction `dir` of a multi-dimensional array of size `dim`.

Parameters

dims [tuple] Number of samples for each dimension

shift [float] Fractional shift to apply in the same unit as `sampling`.

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 Δt .

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.

engine [str, optional] Engine used for fft computation (numpy, scipy, or fftw). Choose numpy when working with CuPy arrays.

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

****kwargs_fft** Arbitrary keyword arguments for `pyfftw.FFTW`

Raises

ValueError If `dims` is provided and `dir` is bigger than `len(dims)`

NotImplementedError If `engine` is neither `numpy`, `scipy`, nor `fftw`

Notes

The Shift operator applies the forward Fourier transform, an element-wise complex scaling, and inverse fourier transform

$$\mathbf{y} = \mathbf{F}^{-1} \mathbf{S} \mathbf{F} \mathbf{x}$$

Here \mathbf{S} is a diagonal operator that scales the Fourier transformed input by $e^{-j2\pi f t_S}$, where t_S is the chosen shift.

Attributes

shape [`tuple`] Operator shape

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

Examples using `pylops.signalprocessing.Shift`

- *Shift*

`pylops.signalprocessing.DWT`

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

<code>__init__(dims[, dir, wavelet, level, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.signalprocessing.DWT`

- *Wavelet transform*

`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 `dirs` 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 [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

<code>__init__(dims[, dirs, wavelet, level, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.signalprocessing.DWT2D`

- *Wavelet transform*
- *05. Image deblurring*

pylops.signalprocessing.Seislet

```
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 ≥ 0).
- kind** [`str`, optional] Basis function used for predict and update steps: `haar` or `linear`.
- 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:

$$\mathbf{o}^{i+1} = \mathbf{r}^i = \mathbf{o}^i - P(\mathbf{e}^i)$$

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):

$$\mathbf{e}^{i+1} = \mathbf{c}^i = \mathbf{e}^i + U(\mathbf{o}^{i+1})$$

where $U = P^-/2$ is the update operator which performs a slope-based backward prediction. At this point \mathbf{e}^{i+1} becomes the new data and the procedure is repeated *level* times (at maximum until \mathbf{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:

$$\mathbf{e}^i = \mathbf{c}^i - U(\mathbf{r}^i) = \mathbf{e}^{i+1} - U(\mathbf{o}^{i+1})$$

and:

$$\mathbf{o}^i = \mathbf{r}^i + P(\mathbf{e}^i) = \mathbf{o}^{i+1} + P(\mathbf{e}^i)$$

A new data is formed by interleaving \mathbf{e}^i and \mathbf{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:

$$\begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \\ \vdots \\ \mathbf{r}_N \\ \mathbf{c}_1 \\ \mathbf{c}_2 \\ \vdots \\ \mathbf{c}_N \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & \dots & \mathbf{0} & -\mathbf{P} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \dots & \mathbf{0} & \mathbf{0} & -\mathbf{P} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{I} & \mathbf{0} & \mathbf{0} & \dots & -\mathbf{P} \\ \mathbf{U} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I} - \mathbf{UP} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{U} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{I} - \mathbf{UP} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{U} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{I} - \mathbf{UP} \end{bmatrix} \begin{bmatrix} \mathbf{o}_1 \\ \mathbf{o}_2 \\ \vdots \\ \mathbf{o}_N \\ \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_N \end{bmatrix}$$

Transposing the operator leads to:

$$\begin{bmatrix} \mathbf{o}_1 \\ \mathbf{o}_2 \\ \vdots \\ \mathbf{o}_N \\ \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_N \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & \dots & \mathbf{0} & -\mathbf{U}^T & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \dots & \mathbf{0} & \mathbf{0} & -\mathbf{U}^T & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{I} & \mathbf{0} & \mathbf{0} & \dots & -\mathbf{U}^T \\ \mathbf{P}^T & \mathbf{0} & \dots & \mathbf{0} & \mathbf{I} - \mathbf{P}^T \mathbf{U}^T & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{P}^T & \dots & \mathbf{0} & \mathbf{0} & \mathbf{I} - \mathbf{P}^T \mathbf{U}^T & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{P}^T & \mathbf{0} & \mathbf{0} & \dots & \mathbf{I} - \mathbf{P}^T \mathbf{U}^T \end{bmatrix} \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \\ \vdots \\ \mathbf{r}_N \\ \mathbf{c}_1 \\ \mathbf{c}_2 \\ \vdots \\ \mathbf{c}_N \end{bmatrix}$$

which can be written more easily in the following two steps:

$$\mathbf{o} = \mathbf{r} + \mathbf{U}^H \mathbf{c}$$

and:

$$\mathbf{e} = \mathbf{c} - \mathbf{P}^H (\mathbf{r} + \mathbf{U}^H (\mathbf{c})) = \mathbf{c} - \mathbf{P}^H \mathbf{o}$$

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

<code>__init__(slopes[, sampling, level, kind, ...])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.

Continued on next page

Table 36 – continued from previous page

<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>inverse(x)</code>	
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.signalprocessing.Seislet`

- *Seislet transform*
- *Slope estimation via Structure Tensor algorithm*

`pylops.signalprocessing.Radon2D`

`pylops.signalprocessing.Radon2D` (*taxis*, *haxis*, *pxaxis*, *kind*='linear', *centeredh*=True, *interp*=True, *onthe-fly*=False, *engine*='numpy', *dtype*='float64')

Two dimensional Radon transform.

Apply two dimensional Radon forward (and adjoint) transform to a 2-dimensional array of size $[n_{p_x} \times n_t]$ (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, p_x) as input and returns t as output

centeredh [`bool`, optional] Assume centered spatial axis (True) or not (False). If True the original *haxis* is ignored and a new axis is created.

interp [`bool`, optional] Apply linear interpolation (True) or nearest interpolation (False) during stacking/spreading along parametric curve

onthe-fly [`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 $f(p_x, x, t) = t_0 + p_x x$ where $p_x = \sin(\theta)/v$ in linear mode, $f(p_x, x, t) = t_0 + p_x x^2$ in parabolic mode, and $f(p_x, x, t) = \sqrt{t_0^2 + x^2/p_x^2}$ in hyperbolic mode. Note that internally the p_x axis will be normalized by the ratio of the spatial and time axes and used alongside unitless axes. Whilst this makes the linear mode fully unitless, users are required to apply additional scalings to the p_x axis for other relationships:

- p_x should be pre-multiplied by d_x for the parabolic relationship;
- p_x should be pre-multiplied by $(d_t/d_x)^2$ for the hyperbolic relationship.

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*

- *1D, 2D and 3D Sliding*
- *Chirp Radon Transform*
- *Radon Transform*
- *Spread How-to*
- *11. Radon filtering*
- *12. Seismic regularization*
- *16. CT Scan Imaging*

pylops.signalprocessing.Radon3D

pylops.signalprocessing.Radon3D(*taxis*, *hyaxis*, *haxis*, *pyaxis*, *pxaxis*, *kind*='linear', *centerdh*=True, *interp*=True, *ontheftly*=False, *engine*='numpy', *dtype*='float64')

Three dimensional Radon transform.

Apply three dimensional Radon forward (and adjoint) transform to a 3-dimensional array of size $[n_{p_y} \times n_{p_x} \times n_t]$ (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
- haxis** [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). If True the original haxis is ignored and a new axis is created.
- interp** [bool, optional] Apply linear interpolation (True) or nearest interpolation (False) during stacking/spreading along parametric curve
- onthe-fly** [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

- 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 $f(p_y, p_x, y, x, t) = t_0 + p_y y + p_x x$ in linear mode, $f(p_y, p_x, y, x, t) = t_0 + p_y y^2 + p_x x^2$ in parabolic mode, and $f(p_y, p_x, y, x, t) = \sqrt{t_0^2 + y^2/p_y^2 + x^2/p_x^2}$ in hyperbolic mode. Note that internally the p_x and p_y axes will be normalized by the ratio of the spatial and time axes and used alongside unitless axes. Whilst this

makes the linear mode fully unitless, users are required to apply additional scalings to the p_x axis for other relationships

- p_x should be pre-multiplied by d_x/d_y for the parabolic relationship;
- p_x should be pre-multiplied by $(d_t/d_x)^2/(d_t/d_y)^2$ for the hyperbolic relationship.

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`

- *1D, 2D and 3D Sliding*
- *Chirp Radon Transform*
- *Radon Transform*
- *Spread How-to*

`pylops.signalprocessing.ChirpRadon2D`

class `pylops.signalprocessing.ChirpRadon2D` (*taxis*, *haxis*, *pmax*, *dtype*='float64')
2D Chirp Radon transform

Apply Radon forward (and adjoint) transform using Fast Fourier Transform and Chirp functions to a 2-dimensional array of size $[n_x \times n_t]$ (both in forward and adjoint mode).

Note that forward and adjoint are swapped compared to the time-space implementation in `pylops.signalprocessing.Radon2D` and a direct *inverse* method is also available for this implementation.

Parameters

taxis [`np.ndarray`] Time axis

haxis [`np.ndarray`] Spatial axis

pmax [`np.ndarray`] Maximum slope defined as \tan of maximum stacking angle in x direction $p_{\max} = \tan(\alpha_{x,\max})$. If one operates in terms of minimum velocity c_0 , set $p_{x,\max} = c_0 \, dy/dt$.

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

Notes

Refer to [1] for the theoretical and implementation details.

Attributes

shape [`tuple`] Operator shape

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

Methods

<code>__init__(taxis, haxis, pmax[, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $y = Ax$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>inverse(x)</code>	
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.signalprocessing.ChirpRadon2D`

- *Chirp Radon Transform*

`pylops.signalprocessing.ChirpRadon3D`

class `pylops.signalprocessing.ChirpRadon3D` (*taxis, hyaxis, hxaxis, pmax, engine='numpy', dtype='float64', **kwargs_fftw*)

3D Chirp Radon transform

Apply Radon forward (and adjoint) transform using Fast Fourier Transform and Chirp functions to a 3-dimensional array of size $[n_y \times n_x \times n_t]$ (both in forward and adjoint mode).

Note that forward and adjoint are swapped compared to the time-space implementation in `pylops.signalprocessing.Radon3D` and a direct *inverse* method is also available for this implementation.

Parameters

taxis [`np.ndarray`] Time axis

hxaxis [`np.ndarray`] Fast patial axis

hyaxis [`np.ndarray`] Slow spatial axis

pmax [`np.ndarray`] Two element array ($p_{y,\max}, p_{x,\max}$) of tan of maximum stacking angles in y and x directions ($\tan(\alpha_{y,\max}), \tan(\alpha_{x,\max})$). If one operates in terms of minimum velocity c_0 , then $p_{y,\max} = c_0 dy/dt$ and $p_{x,\max} = c_0 dx/dt$

engine [`str`, optional] Engine used for fft computation (numpy or fftw)

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

****kwargs_fftw** Arbitrary keyword arguments for `pyfftw.FFTW` (reccomended: `flags=('FFTW_ESTIMATE',), threads=NTHREADS`)

Notes

Refer to [1] for the theoretical and implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

<code>__init__(taxis, hyaxis, hxaxis, pmax[, ...])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{Ax}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>inverse(x)</code>	
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.signalprocessing.ChirpRadon3D`

- *Chirp Radon Transform*

`pylops.signalprocessing.Sliding1D`

`pylops.signalprocessing.Sliding1D(Op, dim, dimd, nwin, nover, tapertype='hanning', design=False)`

1D Sliding transform operator.

Apply a transform operator `Op` repeatedly to slices of the model vector in forward mode and slices of the data vector in adjoint mode. More specifically, in forward mode the model vector is divided into slices, each slice is transformed, and slices are then recombined in a sliding window fashion.

This operator can be used to perform local, overlapping transforms (e.g., `pylops.signalprocessing.FFT`) on 1-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 `nwin` and `nover` as well as the size of the data, sliding windows may not cover the entire data. 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

dim [`tuple`] Shape of 1-dimensional model.

dimd [`tuple`] Shape of 1-dimensional data

nwin [`int`] Number of samples of window

nover [`int`] Number of samples of overlapping part of window

tapertype [`str`, optional] Type of taper (hanning, cosine, cosinesquare or None)

design [`bool`, optional] Print number of 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.Sliding1D`

- *1D, 2D and 3D Sliding*

`pylops.signalprocessing.Sliding2D`

`pylops.signalprocessing.Sliding2D` (*Op, dims, dimsd, nwin, nover, tapertype='hanning', design=False*)

2D Sliding transform operator.

Apply a transform operator `Op` repeatedly to slices of the model vector in forward mode and slices of the data vector in adjoint mode. More specifically, in forward mode the model vector is divided into slices, each slice is transformed, and slices are then recombined in a sliding window fashion. Both model and data are internally reshaped and interpreted as 2-dimensional arrays: each slice contains 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.FFT2D` or `pylops.signalprocessing.Radon2D`) on 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 *nwin* and *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 `design=True` while defining the best sliding window approach.

Parameters

- Op** [*pylops.LinearOperator*] Transform operator
- dims** [*tuple*] Shape of 2-dimensional model. Note that `dims[0]` should be multiple of the model size of the transform in the first dimension
- dimsd** [*tuple*] Shape of 2-dimensional data
- nwin** [*int*] Number of samples of window
- nover** [*int*] Number of samples of overlapping part of window
- tapertype** [*str*, optional] Type of taper (hanning, cosine, cosinesquare or None)
- design** [*bool*, optional] Print number of 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.Sliding2D`

- *1D, 2D and 3D Sliding*
- *2D Patching*
- *12. Seismic regularization*

`pylops.signalprocessing.Sliding3D`

`pylops.signalprocessing.Sliding3D` (*Op, dims, dimsd, nwin, nover, nop, tapertype='hanning', design=False, nproc=1*)

3D 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 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., `pylops.signalprocessing.FFTND` or `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 *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 *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`

- *1D, 2D and 3D Sliding*

`pylops.signalprocessing.Patch2D`

`pylops.signalprocessing.Patch2D` (*Op, dims, dimsd, nwin, nover, nop, tapertype='hanning', design=False*)

2D Patch 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 recombined together. Both model and data are internally reshaped and interpreted as 2-dimensional arrays: each patch contains a portion of the array in both the first and second dimension.

This operator can be used to perform local, overlapping transforms (e.g., `pylops.signalprocessing.FFT2D` or `pylops.signalprocessing.Radon2D`) on 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 *nwin* and *nover* as well as the size of the data, patches may not cover the entire size of the data. The start and end indices of each window can be displayed using `design=True` while defining the best patching approach.

Parameters

Op [*pylops.LinearOperator*] Transform operator

dims [*tuple*] Shape of 2-dimensional model. Note that `dims[0]` and `dims[1]` should be multiple of the model size of the transform in their respective dimensions

dimstd [*tuple*] Shape of 2-dimensional data

nwin [*tuple*] Number of samples of window

nover [*tuple*] Number of samples of overlapping part of window

nop [*tuple*] Size of model in the transformed domain

tapertype [*str*, optional] Type of taper (hanning, cosine, cosinesquare or None)

design [*bool*, optional] Print number of 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`).

See also:

Sliding2d 2D Sliding transform operator.

Examples using `pylops.signalprocessing.Patch2D`

- [2D Patching](#)

`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 $[n_{\text{slice}} \times n_x \times n_y]$

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

$$d(k, x, z) = \int G(k, x, y) m(k, y, z) dy \quad \forall k = 1, \dots, n_{\text{slice}}$$

on the other hand its adjoint is expressed as

$$m(k, y, z) = \int G^*(k, y, x) d(k, x, z) dx \quad \forall k = 1, \dots, n_{\text{slice}}$$

In discrete form, this operator can be seen as a block-diagonal matrix multiplication:

$$\begin{bmatrix} \mathbf{G}_{k=1} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_{k=2} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{G}_{k=N} \end{bmatrix} \begin{bmatrix} \mathbf{m}_{k=1} \\ \mathbf{m}_{k=2} \\ \vdots \\ \mathbf{m}_{k=N} \end{bmatrix}$$

Attributes

shape [`tuple`] Operator shape

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

Methods

<code>__init__(G[, nz, saveGt, usematmul, dtype])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uselobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uselobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator
<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Wave-Equation processing

<i>PressureToVelocity</i> (nt, nr, dt, dr, rho, vel)	Pressure to Vertical velocity conversion.
<i>UpDownComposition2D</i> (nt, nr, dt, dr, rho, vel)	2D Up-down wavefield composition.
<i>UpDownComposition3D</i> (nt, nr, dt, dr, rho, vel)	3D Up-down wavefield composition.
<i>MDC</i> (G, nt, nv[, dt, dr, twosided, fast, ...])	Multi-dimensional convolution.
<i>PhaseShift</i> (vel, dz, nt, freq, kx[, ky, dtype])	Phase shift operator
<i>Demigration</i> (z, x, t, srcs, recs, vel, wav, ...)	Kirchhoff 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, *backend*='numpy', *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 $([n_{r_y} \times n_{r_x} \times n_t])$.

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 ρ along the receiver array (must be constant)
- vel** [float] Velocity c 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 $\sqrt{k_y^2 + k_x^2} < \frac{\omega}{c}$ 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)
- backend** [str, optional] Backend used for creation of obliquity factor operator (numpy or cupy)
- 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]:

$$\hat{v}_z(k_x, \omega) = \frac{k_z}{\omega \rho} \hat{p}(k_x, \omega)$$

where the vertical wavenumber k_z is defined as $k_z = \sqrt{\frac{\omega^2}{c^2} - k_x^2}$.

Similarly a vertical particle velocity can be converted into an equivalent pressure wavefield by applying the following frequency-wavenumber dependant scaling [1]:

$$\hat{p}(k_x, \omega) = \frac{\omega \rho}{k_z} \hat{v}_z(k_x, \omega)$$

For 3-dimensional applications the only difference is represented by the vertical wavenumber k_z , which is defined as $k_z = \sqrt{\frac{\omega^2}{c^2} - k_x^2 - k_y^2}$.

In both cases, this operator is implemented as a concatenation of a 2 or 3-dimensional forward FFT (`pylops.signalprocessing.FFT2` or `pylops.signalprocessing.FFTN`), a weighting matrix implemented via `pylops.basicprocessing.Diagonal`, and 2 or 3-dimensional inverse FFT.

Examples using `pylops.waveeqprocessing.PressureToVelocity`

- 14. *Seismic wavefield decomposition*

`pylops.waveeqprocessing.UpDownComposition2D`

```
pylops.waveeqprocessing.UpDownComposition2D(nt, nr, dt, dr, rho, vel, nffts=(None, None),
                                             critical=100.0, ntaper=10, scaling=1.0,
                                             backend='numpy', dtype='complex128')
```

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

nt [`int`] Number of samples along the time axis

nr [`int`] Number of samples along the receiver axis

dt [`float`] Sampling along the time axis

dr [`float`] Sampling along the receiver array

rho [`float`] Density ρ along the receiver array (must be constant)

vel [`float`] Velocity c 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 $|k_x| < \frac{f(k_x)}{c}$ will be retained 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)

backend [`str`, optional] Backend used for creation of obliquity factor operator (`numpy` or `cupy`)

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]:

$$\begin{bmatrix} \hat{p} \\ \hat{v}_z \end{bmatrix} (k_x, \omega) = \begin{bmatrix} 1 & 1 \\ \frac{k_z}{\omega\rho} & -\frac{k_z}{\omega\rho} \end{bmatrix} \begin{bmatrix} \hat{p}^+ \\ \hat{p}^- \end{bmatrix} (k_x, \omega)$$

where the vertical wavenumber k_z is defined as $k_z = \sqrt{\frac{\omega^2}{c^2} - k_x^2}$.

We can write the entire composition process in a compact matrix-vector notation as follows:

$$\begin{bmatrix} \mathbf{p} \\ s\mathbf{v}_z \end{bmatrix} = \begin{bmatrix} \mathbf{F} & 0 \\ 0 & s\mathbf{F} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{I} \\ \mathbf{W}^+ & \mathbf{W}^- \end{bmatrix} \begin{bmatrix} \mathbf{F}^H & 0 \\ 0 & \mathbf{F}^H \end{bmatrix} \mathbf{p}^\pm$$

where \mathbf{F} is the 2-dimensional FFT (`pylops.signalprocessing.FFT2`), \mathbf{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.waveeqprocessing.UpDownComposition2D`

- [14. Seismic wavefield decomposition](#)

pylops.waveeqprocessing.UpDownComposition3D

```
pylops.waveeqprocessing.UpDownComposition3D(nt, nr, dt, dr, rho, vel, nffts=(None,
                                         None, None), critical=100.0, nta-
                                         per=10, scaling=1.0, backend='numpy',
                                         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 $[n_{r_y} \times n_{r_x} \times n_t]$ 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 ρ along the receiver array (must be constant)
- vel** [float] Velocity c 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 $\sqrt{k_y^2 + k_x^2} < \frac{\omega}{c}$ 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)
- backend** [str, optional] Backend used for creation of obliquity factor operator (numpy or cupy)
- 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 $k_z = \sqrt{\frac{\omega^2}{c^2} - k_y^2 - k_x^2}$.

pylops.waveeqprocessing.MDC

`pylops.waveeqprocessing.MDC` (*G*, *nt*, *nv*, *dt=1.0*, *dr=1.0*, *twosided=True*, *fast=None*, *dtype=None*, *fftengine='numpy'*, *transpose=True*, *saveGt=True*, *conj=False*, *usematmul=False*, *prescaled=False*)

Multi-dimensional convolution.

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 $[n_r (\times n_{vs}) \times n_t]$ and $[n_s (\times n_{vs}) \times n_t]$ (or $2n_t - 1$ for `twosided=True`), respectively. If `transpose=False`, model and data should be provided after flattening 2- or 3-dimensional arrays of size $[n_t \times n_r (\times n_{vs})]$ and $[n_t \times n_s (\times n_{vs})]$ (or $2n_t - 1$ 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 recommended 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 for model and data (note that this must be equal to $2n_t - 1$ when working with `twosided=True`).
- nv** [`int`] Number of samples along virtual source axis
- dt** [`float`, optional] Sampling of time integration axis Δt
- dr** [`float`, optional] Sampling of receiver integration axis Δr
- 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`, `scipy` 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:

$$y(t, s, v) = \sqrt{n_t} \Delta t \Delta r F^{-1} \left(\sum_{i_r=0}^{n_r} G(f, s, i_r) F(x(t, i_r, v)) \right)$$

where $\sqrt{n_t} \Delta t \Delta r$ is not applied if `prescaled=True`.

This operation can be discretized and performed by means of a linear operator

$$\mathbf{D} = \mathbf{F}^H \mathbf{G} \mathbf{F}$$

where \mathbf{F} is the Fourier transform applied along the time axis and \mathbf{G} is the multi-dimensional convolution kernel.

Examples using `pylops.waveeqprocessing.MDC`

- *Multi-Dimensional Convolution*
- *09. Multi-Dimensional Deconvolution*

`pylops.waveeqprocessing.PhaseShift`

`pylops.waveeqprocessing.PhaseShift` (*vel, dz, nt, freq, kx, ky=None, dtype='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

vel [`float`, optional] Constant propagation velocity

dz [`float`, optional] Depth step

nt [`int`, optional] Number of time samples of model and data

freq [`numpy.ndarray`] Positive frequency axis

kx [`int`, optional] Horizontal wavenumber axis (centered around 0) of size $[n_x \times 1]$.

ky [`int`, optional] Second horizontal wavenumber axis for 3d phase shift (centered around 0) of size $[n_y \times 1]$.

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

Returns

Pop [`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:

$$d(f, k_x, k_y) = m(f, k_x, k_y) e^{-j \Delta z \sqrt{\omega^2/v^2 - k_x^2 - k_y^2}}$$

where v is the constant propagation velocity and Δz is the propagation depth. In adjoint mode, the data is propagated backward using the following transformation:

$$m(f, k_x, k_y) = d(f, k_x, k_y) e^{j \Delta z \sqrt{\omega^2/v^2 - k_x^2 - k_y^2}}$$

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:

$$\mathbf{d} = \mathbf{F}_t^H \mathbf{F}_x^H \mathbf{P} \mathbf{F}_x \mathbf{F}_t \mathbf{m}$$

where \mathbf{P} perfoms the phase-shift as discussed above.

Examples using `pylops.waveeqprocessing.PhaseShift`

- *PhaseShift operator*

`pylops.waveeqprocessing.Demigration`

`pylops.waveeqprocessing.Demigration`(`z`, `x`, `t`, `srcs`, `recs`, `vel`, `wav`, `wavcenter`, `y=None`, `trav=None`, `mode='eikonal'`)

Kirchhoff 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 $[(n_y)n_xn_z \times n_r]$ (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:

$$d(\mathbf{x}_r, \mathbf{x}_s, t) = w(t) * \int_V G(\mathbf{x}_r, \mathbf{x}, t) m(\mathbf{x}) G(\mathbf{x}, \mathbf{x}_s, t) d\mathbf{x}$$

where $m(\mathbf{x})$ is the model and it represents the reflectivity at every location in the subsurface, $G(\mathbf{x}, \mathbf{x}_s, t)$ and $G(\mathbf{x}_r, \mathbf{x}, t)$ 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: traveltimes curves between source to receiver pairs are computed for every subsurface point and Green's functions are implemented from traveltimes look-up tables, placing the reflectivity values at corresponding source-to-receiver time in the data.
- byot: bring your own table. Traveltimes 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

<code>avo.AVOLinearModelling(theta[, vsvp, nt0, ...])</code>	AVO Linearized modelling.
<code>poststack.PoststackLinearModelling(wav, nt0)</code>	Post-stack linearized seismic modelling operator.
<code>prestack.PrestackLinearModelling(wav, theta)</code>	Pre-stack linearized seismic modelling operator.
<code>prestack.PrestackWaveletModelling(m, theta, nwav)</code>	Pre-stack linearized seismic modelling operator for wavelet.

pylops.avo.avo.AVOLinearModelling

class pylops.avo.avo.AVOLinearModelling (*theta, vsvp=0.5, nt0=1, spatdims=None, linearization='akirich', dtype='float64'*)

AVO Linearized modelling.

Create operator to be applied to a combination of elastic parameters for generation of seismic pre-stack reflectivity.

Parameters

theta [np.ndarray] Incident angles in degrees

vsvp [np.ndarray or float] V_S/V_P 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 [{"akirich", "fatti", "PS"}, optional]

- “akirich”: Aki-Richards. See `pylops.avo.avo.akirichards`.
- “fatti”: Fatti. See `pylops.avo.avo.fatti`.
- “PS”: PS. See `pylops.avo.avo.ps`.

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

Raises

NotImplementedError If `linearization` is not an implemented linearization

Notes

The AVO linearized operator performs a linear combination of three (or two) elastic parameters arranged in input vector \mathbf{m} of size $n_{t_0} \times N$ 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 `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

<code>__init__(theta[, vsvp, nt0, spatdims, ...])</code>	Initialize this LinearOperator.
<code>adjoint()</code>	Hermitian adjoint.
<code>apply_columns(cols)</code>	Apply subset of columns of operator
<code>cond([uslobpcg])</code>	Condition number of linear operator.
<code>conj()</code>	Complex conjugate operator
<code>div(y[, niter, densesolver])</code>	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
<code>dot(x)</code>	Matrix-matrix or matrix-vector multiplication.
<code>eigs([neigs, symmetric, niter, uslobpcg])</code>	Most significant eigenvalues of linear operator.
<code>matmat(X)</code>	Matrix-matrix multiplication.
<code>matvec(x)</code>	Matrix-vector multiplication.
<code>rmatmat(X)</code>	Matrix-matrix multiplication.
<code>rmatvec(x)</code>	Adjoint matrix-vector multiplication.
<code>todense([backend])</code>	Return dense matrix.
<code>toimag([forw, adj])</code>	Imag operator
<code>toreal([forw, adj])</code>	Real operator

Continued on next page

Table 42 – continued from previous page

<code>tosparse()</code>	Return sparse matrix.
<code>trace([neval, method, backend])</code>	Trace of linear operator.
<code>transpose()</code>	Transpose this linear operator.

Examples using `pylops.avo.avo.AVOLinearModelling`

- *AVO modelling*

`pylops.avo.poststack.PoststackLinearModelling`

`pylops.avo.poststack.PoststackLinearModelling` (*wav*, *nt0*, *spatdims=None*, *explicit=False*, *sparse=False*, *kind='centered'*)

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 $[n_{t_0} (\times n_x \times n_y)]$.

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 $[n_{t_0} \times n_{\text{wav}}]$ where n_{wav} is the length of each wavelet). Note that the `dtype` of this variable will define that of the operator

nt0 [`int`] Number of samples along time axis

spatdims [`int` or `tuple`, optional] Number of samples along spatial axis (or axes) (`None` if only one dimension is available)

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)

sparse [`bool`, optional] Create a sparse matrix (`True`) or dense (`False`) when `explicit=True`

Returns

Pop [`LinearOperator`] post-stack modelling operator.

Raises

ValueError If *wav* is two dimensional but does not contain *nt0* wavelets

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:

$$d(t, \theta = 0) = w(t) * \frac{d \ln m(t)}{dt}$$

where $m(t)$ is the elastic parameter profile and $w(t)$ is the time domain seismic wavelet. In compact form:

$$\mathbf{d} = \mathbf{W} \mathbf{D} \mathbf{m}$$

In the special case of acoustic impedance ($m(t) = AI(t)$), the modelling operator can be used to create zero-offset data:

$$d(t, \theta = 0) = \frac{1}{2} w(t) * \frac{d \ln m(t)}{dt}$$

where the scaling factor $\frac{1}{2}$ can be easily included in the wavelet.

Examples using `pylops.avo.poststack.PoststackLinearModelling`

- *07. Post-stack inversion*

`pylops.avo.prestack.PrestackLinearModelling`

`pylops.avo.prestack.PrestackLinearModelling(wav, theta, vsvp=0.5, nt0=1, spatdims=None, linearization='akirich', explicit=False, kind='centered')`

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. The input model must be arranged in a vector of size $n_m \times n_{t_0} (\times n_x \times n_y)$ for `explicit=True` and $n_{t_0} \times n_m (\times n_x \times n_y)$ for `explicit=False`. Similarly the output data is arranged in a vector of size $n_\theta \times n_{t_0} (\times n_x \times n_y)$ for `explicit=True` and $n_{t_0} \times n_\theta (\times n_x \times n_y)$ for `explicit=False`.

Parameters

wav [`np.ndarray`] Wavelet in time domain (must had odd number of elements and centered to zero). Note that the `dtype` of this variable will define that of the operator

theta [`np.ndarray`] Incident angles in degrees. Must have same `dtype` of `wav` (or it will be automatically casted to it)

vsvp [`float` or `np.ndarray`] V_S/V_P ratio (constant or time/depth variant)

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 [{`"akirich"`, `"fatti"`, `"PS"`} or `callable`, optional]

- `"akirich"`: Aki-Richards. See `pylops.avo.avo.akirichards`.
- `"fatti"`: Fatti. See `pylops.avo.avo.fatti`.
- `"PS"`: PS. See `pylops.avo.avo.ps`.
- Function with the same signature as `pylops.avo.avo.akirichards`

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)

kind [`str`, optional] Derivative kind (forward or centered).

Returns

Preop [`LinearOperator`] pre-stack modelling operator.

Raises

NotImplementedError If `linearization` is not an implemented linearization

NotImplementedError If kind is not forward nor centered

Notes

Pre-stack seismic modelling is the process of constructing seismic pre-stack data from three (or two) profiles of elastic parameters in time (or depth) domain. This can be easily achieved using the following forward model:

$$d(t, \theta) = w(t) * \sum_{i=1}^{n_m} G_i(t, \theta) m_i(t)$$

where $w(t)$ is the time domain seismic wavelet. In compact form:

$$\mathbf{d} = \mathbf{G}\mathbf{m}$$

On the other hand, pre-stack inversion aims at recovering the different profiles of elastic properties from the band-limited seismic pre-stack data.

Examples using `pylops.avo.prestack.PrestackLinearModelling`

- *Pre-stack modelling*
- *08. Pre-stack (AVO) inversion*

`pylops.avo.prestack.PrestackWaveletModelling`

`pylops.avo.prestack.PrestackWaveletModelling` (*m*, *theta*, *nwav*, *wavc=None*, *vsvp=0.5*, *linearization='akirich'*)

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 $[n_{t_0} \times N]$ where $N = 3, 2$. Note that the dtype of this variable will define that of the operator

theta [`int`] Incident angles in degrees. Must have same dtype of *m* (or it will be automatically cast to it)

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] V_S/V_P ratio

linearization [`{“akirich”, “fatti”, “PS”}` or `callable`, optional]

- “akirich”: Aki-Richards. See `pylops.avo.avo.akirichards`.
- “fatti”: Fatti. See `pylops.avo.avo.fatti`.
- “PS”: PS. See `pylops.avo.avo.ps`.
- Function with the same signature as `pylops.avo.avo.akirichards`

Returns

Mconv [`LinearOperator`] pre-stack modelling operator for wavelet estimation.

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 $n_{t_0} \times N$:

$$d(t, \theta) = \sum_{i=1}^N G_i(t, \theta) m_i(t) * w(t)$$

where $w(t)$ is the time domain seismic wavelet. In compact form:

$$\mathbf{d} = \mathbf{G}\mathbf{w}$$

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`

- *Wavelet estimation*

3.8.2 Solvers**Basic**

<code>solver.cg(Op, y, x0[, niter, damp, tol, ...])</code>	Conjugate gradient
<code>solver.cgls(Op, y, x0[, niter, damp, tol, ...])</code>	Conjugate gradient least squares
<code>solver.lsqr(Op, y, x0[, damp, atol, btol, ...])</code>	LSQR

`pylops.optimization.solver.cg`

`pylops.optimization.solver.cg` (*Op*, *y*, *x0*, *niter*=10, *damp*=0.0, *tol*=0.0001, *show*=False, *callback*=None)

Conjugate gradient

Solve a square system of equations given an operator *Op* and data *y* using conjugate gradient iterations.

Parameters

Op [*pylops.LinearOperator*] Operator to invert of size $[N \times N]$

y [*np.ndarray*] Data of size $[N \times 1]$

x0 [*np.ndarray*, optional] Initial guess

niter [*int*, optional] Number of iterations

damp [*float*, optional] *Deprecated*, will be removed in v2.0.0

tol [*float*, optional] Tolerance on residual norm

show [*bool*, optional] Display iterations log

callback [`callable`, optional] Function with signature (`callback(x)`) to call after each iteration where `x` is the current model vector

Returns

x [`np.ndarray`] Estimated model of size $[N \times 1]$

iit [`int`] Number of executed iterations

cost [`numpy.ndarray`, optional] History of the L2 norm of the residual

Notes

Solve the $y = \mathbf{O}p\mathbf{x}$ problem using conjugate gradient iterations.

pylops.optimization.solver.cgls

`pylops.optimization.solver.cgls` (*Op*, *y*, *x0*, *niter=10*, *damp=0.0*, *tol=0.0001*, *show=False*, *callback=None*)

Conjugate gradient least squares

Solve an overdetermined system of equations given an operator *Op* and data *y* using conjugate gradient iterations.

Parameters

Op [`pylops.LinearOperator`] Operator to invert of size $[N \times M]$

y [`np.ndarray`] Data of size $[N \times 1]$

x0 [`np.ndarray`, optional] Initial guess

niter [`int`, optional] Number of iterations

damp [`float`, optional] Damping coefficient

tol [`float`, optional] Tolerance on residual norm

show [`bool`, optional] Display iterations log

callback [`callable`, optional] Function with signature (`callback(x)`) to call after each iteration where `x` is the current model vector

Returns

x [`np.ndarray`] Estimated model of size $[M \times 1]$

istop [`int`] Gives the reason for termination

1 means `x` is an approximate solution to $\mathbf{d} = \mathbf{O}p\mathbf{x}$

2 means `x` approximately solves the least-squares problem

iit [`int`] Iteration number upon termination

r1norm [`float`] $\|\mathbf{r}\|_2$, where $\mathbf{r} = \mathbf{d} - \mathbf{O}p\mathbf{x}$

r2norm [`float`] $\sqrt{\mathbf{r}^T \mathbf{r} + \epsilon^2 \mathbf{x}^T \mathbf{x}}$. Equal to **r1norm** if $\epsilon = 0$

cost [`numpy.ndarray`, optional] History of **r1norm** through iterations

Notes

Minimize the following functional using conjugate gradient iterations:

$$J = ||\mathbf{y} - \mathbf{Op}\mathbf{x}||_2^2 + \epsilon^2 ||\mathbf{x}||_2^2$$

where ϵ is the damping coefficient.

Examples using `pylops.optimization.solver.cgl`s

- *CGLS and LSQR Solvers*

`pylops.optimization.solver.lsqr`

```
pylops.optimization.solver.lsqr(Op, y, x0, damp=0.0, atol=1e-08, btol=1e-08, con-  
lim=100000000.0, niter=10, calc_var=True, show=False,  
callback=None)
```

LSQR

Solve an overdetermined system of equations given an operator `Op` and data `y` using LSQR iterations.

cond

Parameters

Op [`pylops.LinearOperator`] Operator to invert of size $[N \times M]$

y [`np.ndarray`] Data of size $[N \times 1]$

x0 [`np.ndarray`, optional] Initial guess

damp [`float`, optional] Damping coefficient

atol, btol [`float`, optional] Stopping tolerances. If both are $1.0\text{e-}9$, the final residual norm should be accurate to about 9 digits. (The solution will usually have fewer correct digits, depending on (**Op**) and the size of **damp**.)

conlim [`float`, optional] Stopping tolerance on (**Op**) exceeds **conlim**. For square, **conlim** could be as large as $1.0\text{e+}12$. For least-squares problems, **conlim** should be less than $1.0\text{e+}8$. Maximum precision can be obtained by setting **atol** = **btol** = **conlim** = 0, but the number of iterations may then be excessive.

niter [`int`, optional] Number of iterations

calc_var [`bool`, optional] Estimate diagonals of $(\mathbf{Op}^H \mathbf{Op} + \epsilon^2 \mathbf{I})^{-1}$.

show [`bool`, optional] Display iterations log

callback [`callable`, optional] Function with signature `(callback(x))` to call after each iteration where `x` is the current model vector

Returns

x [`np.ndarray`] Estimated model of size $[M \times 1]$

istop [`int`] Gives the reason for termination

0 means the exact solution is $\mathbf{x} = 0$

1 means `x` is an approximate solution to $\mathbf{y} = \mathbf{Op}\mathbf{x}$

- 2 means \mathbf{x} approximately solves the least-squares problem
- 3 means the estimate of $(\overline{\mathbf{Op}})$ has exceeded `conlim`
- 4 means $\mathbf{y} - \mathbf{Op} \mathbf{x}$ is small enough for this machine
- 5 means the least-squares solution is good enough for this machine
- 6 means $(\overline{\mathbf{Op}})$ seems to be too large for this machine
- 7 means the iteration limit has been reached

r1norm [float] $\|\mathbf{r}\|_2^2$, where $\mathbf{r} = \mathbf{y} - \mathbf{Op} \mathbf{x}$

r2norm [float] $\sqrt{\mathbf{r}^T \mathbf{r} + \epsilon^2 \mathbf{x}^T \mathbf{x}}$. Equal to `r1norm` if $\epsilon = 0$

anorm [float] Estimate of Frobenius norm of $\overline{\mathbf{Op}} = [\mathbf{Op} \ \epsilon \mathbf{I}]$

acond [float] Estimate of $(\overline{\mathbf{Op}})$

arnorm [float] Estimate of norm of $(\mathbf{Op}^H \mathbf{r} - \epsilon^2 \mathbf{x})$

var [float] Diagonals of $(\mathbf{Op}^H \mathbf{Op})^{-1}$ (if `damp=0`) or more generally $(\mathbf{Op}^H \mathbf{Op} + \epsilon^2 \mathbf{I})^{-1}$.

cost [numpy.ndarray, optional] History of `r1norm` through iterations

Notes

Minimize the following functional using LSQR iterations [1]:

$$J = \|\mathbf{y} - \mathbf{Op} \mathbf{x}\|_2^2 + \epsilon^2 \|\mathbf{x}\|_2^2$$

where ϵ is the damping coefficient.

Examples using `pylops.optimization.solver.lsqr`

- *CGLS and LSQR Solvers*

Least-squares

`leastsquares.NormalEquationsInversion(Op)` Inversion of normal equations.
 ...)

`leastsquares.RegularizedInversion(Op, Regs, data)` Regularized inversion.

`leastsquares.PreconditionedInversion(Op, P, data)` Preconditioned inversion.

pylops.optimization.leastsquares.NormalEquationsInversion

```
pylops.optimization.leastsquares.NormalEquationsInversion(Op,      Regs,      data,
                                                           Weight=None,
                                                           dataregs=None,
                                                           epsI=0,   epsRs=None,
                                                           x0=None,                re-
                                                           turninfo=False,
                                                           NRegs=None,
                                                           epsNRs=None,
                                                           **kwargs_solver)
```

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_solver** Arbitrary keyword arguments for chosen solver (*scipy.sparse.linalg.cg* and *pylops.optimization.solver.cg* are used as default for numpy and cupy *data*, respectively)

Note: When user does not supply `atol`, it is set to “legacy”.

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 **Op**, a data weighting operator **W**, a list of regularization terms (**R_i** and/or **N_i**), the data **d** and regularization data **d_{R_i}**, and the damping factors ϵ_I , ϵ_{R_i} and ϵ_{N_i} :

$$(\mathbf{Op}^T \mathbf{W} \mathbf{Op} + \sum_i \epsilon_{R_i}^2 \mathbf{R}_i^T \mathbf{R}_i + \sum_i \epsilon_{N_i}^2 \mathbf{N}_i + \epsilon_I^2 \mathbf{I}) \mathbf{x} = \mathbf{Op}^T \mathbf{W} \mathbf{d} + \sum_i \epsilon_{R_i}^2 \mathbf{R}_i^T \mathbf{d}_{R_i}$$

Note that the data term of the regularizations **N_i** is implicitly assumed to be zero.

Examples using `pylops.optimization.leastsquares.NormalEquationsInversion`

- *Bilinear Interpolation*
- *03. Solvers*
- *05. Image deblurring*
- *06. 2D Interpolation*

`pylops.optimization.leastsquares.RegularizedInversion`

`pylops.optimization.leastsquares.RegularizedInversion` (*Op*, *Regs*, *data*, *Weight=None*, *dataregs=None*, *epsRs=None*, *x0=None*, *returninfo=False*, ***kwargs_solver*)

Regularized inversion.

Solve a system of regularized equations given the operator *Op*, a data weighting operator *Weight*, and a list of regularization terms *Regs*.

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 (if None a zero data will be used for every regularization operator in *Regs*)

epsRs [*list*, optional] Regularization dampings

x0 [*numpy.ndarray*, optional] Initial guess

returninfo [*bool*, optional] Return info of LSQR solver

****kwargs_solver** Arbitrary keyword arguments for chosen solver (`scipy.sparse.linalg.lsqr` and `pylops.optimization.solver.cgls` are used as default for numpy and cupy *data*, respectively)

Returns

xinv [`numpy.ndarray`] Inverted model.

istop [`int`] Gives the reason for termination

1 means \mathbf{x} is an approximate solution to $\mathbf{d} = \mathbf{Op} \mathbf{x}$

2 means \mathbf{x} approximately solves the least-squares problem

itn [`int`] Iteration number upon termination

r1norm [`float`] $\|\mathbf{r}\|_2^2$, where $\mathbf{r} = \mathbf{d} - \mathbf{Op} \mathbf{x}$

r2norm [`float`] $\sqrt{\mathbf{r}^T \mathbf{r} + \epsilon^2 \mathbf{x}^T \mathbf{x}}$. 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 \mathbf{Op} , a data weighting operator $\mathbf{W}^{1/2}$, a list of regularization terms \mathbf{R}_i , the data \mathbf{d} and regularization damping factors ϵ_I : and $\epsilon_{\mathbf{R}_i}$:

$$\begin{bmatrix} \mathbf{W}^{1/2} \mathbf{Op} \\ \epsilon_{\mathbf{R}_1} \mathbf{R}_1 \\ \vdots \\ \epsilon_{\mathbf{R}_N} \mathbf{R}_N \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{W}^{1/2} \mathbf{d} \\ \epsilon_{\mathbf{R}_1} \mathbf{d}_{\mathbf{R}_1} \\ \vdots \\ \epsilon_{\mathbf{R}_N} \mathbf{d}_{\mathbf{R}_N} \end{bmatrix}$$

where the `Weight` provided here is equivalent to the square-root of the weight in `pylops.optimization.leastsquares.NormalEquationsInversion`. Note that this system is solved using the `scipy.sparse.linalg.lsqr` and an initial guess `x0` can be provided to this solver, despite the original solver does not allow so.

Examples using `pylops.optimization.leastsquares.RegularizedInversion`

- *MP, OMP, ISTA and FISTA*
- *Total Variation (TV) Regularization*
- *Wavelet estimation*
- *03. Solvers*
- *06. 2D Interpolation*
- *16. CT Scan Imaging*

pylops.optimization.leastsquares.PreconditionedInversion

```
pylops.optimization.leastsquares.PreconditionedInversion(Op, P, data, x0=None,
                                                         returninfo=False,
                                                         **kwargs_solver)
```

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
x0 [*numpy.ndarray*] Initial guess
returninfo [*bool*] Return info of LSQR solver
****kwargs_solver** Arbitrary keyword arguments for chosen solver (*scipy.sparse.linalg.lsqr* and *pylops.optimization.solver.cgls* are used as default for *numpy* and *cupy* *data*, respectively)

Returns

xinv [*numpy.ndarray*] Inverted model.
istop [*int*] Gives the reason for termination
 1 means \mathbf{x} is an approximate solution to $\mathbf{d} = \text{Op } \mathbf{x}$
 2 means \mathbf{x} approximately solves the least-squares problem
itn [*int*] Iteration number upon termination
r1norm [*float*] $\|\mathbf{r}\|_2^2$, where $\mathbf{r} = \mathbf{d} - \text{Op } \mathbf{x}$
r2norm [*float*] $\sqrt{\mathbf{r}^T \mathbf{r} + \epsilon^2 \mathbf{x}^T \mathbf{x}}$. 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 \mathbf{d}

$$\mathbf{d} = \text{Op } \text{P} \mathbf{m}$$

where \mathbf{m} is the solution in the preconditioned space and $\mathbf{x} = \text{P} \mathbf{m}$ is the solution in the original space.

Examples using `pylops.optimization.leastsquares.PreconditionedInversion`

- *Wavelet estimation*
- *03. Solvers*

Sparsity

<code>sparsity.IRLS(Op, data, nouter[, threshR, ...])</code>	Iteratively reweighted least squares.
<code>sparsity.OMP(Op, data[, niter_outer, ...])</code>	Orthogonal Matching Pursuit (OMP).
<code>sparsity.ISTA(Op, data, niter[, eps, alpha, ...])</code>	Iterative Shrinkage-Thresholding Algorithm (ISTA).
<code>sparsity.FISTA(Op, data, niter[, eps, ...])</code>	Fast Iterative Shrinkage-Thresholding Algorithm (FISTA).
<code>sparsity.SPGL1(Op, data[, SOp, tau, sigma, x0])</code>	Spectral Projected-Gradient for L1 norm.
<code>sparsity.SplitBregman(Op, RegsL1, data[, ...])</code>	Split Bregman for mixed L2-L1 norms.

pylops.optimization.sparsity.IRLS

`pylops.optimization.sparsity.IRLS` (*Op*, *data*, *nouter*, *threshR=False*, *epsR=1e-10*, *epsI=1e-10*, *x0=None*, *tolIRLS=1e-10*, *returnhistory=False*, *kind='data'*, ***kwargs_solver*)

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 *Op* and data *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 L1 norm given less weight to large residuals than L2 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

Op [*pylops.LinearOperator*] Operator to invert

data [*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 when using numpy data (or *pylops.optimization.solver.cg* and *pylops.optimization.solver.cgl*s when using cupy data)

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{Op} and the data \mathbf{d} :

$$J = \|\mathbf{d} - \mathbf{Op} \mathbf{x}\|_1$$

by a set of outer iterations which require to repeatedly solve a weighted least squares problem of the form:

$$*arg \min_{\mathbf{x}} \|\mathbf{d} - \mathbf{Op} \mathbf{x}\|_{2, \mathbf{R}^{(i)}}^2 + \epsilon_{\mathbf{I}}^2 \|\mathbf{x}\|_2^2$$

where $\mathbf{R}^{(i)}$ is a diagonal weight matrix whose diagonal elements at iteration i are equal to the absolute inverses of the residual vector $\mathbf{r}^{(i)} = \mathbf{y} - \mathbf{Op} \mathbf{x}^{(i)}$ at iteration i . More specifically the j -th element of the diagonal of $\mathbf{R}^{(i)}$ is

$$R_{j,j}^{(i)} = \frac{1}{|r_j^{(i)}| + \epsilon_{\mathbf{R}}}$$

or

$$R_{j,j}^{(i)} = \frac{1}{\max\{|r_j^{(i)}|, \epsilon_{\mathbf{R}}\}}$$

depending on the choice `threshR`. In either case, $\epsilon_{\mathbf{R}}$ is the user-defined stabilization/thresholding factor [1].

Similarly *model IRLS* solves the following optimization problem for the operator \mathbf{Op} and the data \mathbf{d} :

$$J = \|\mathbf{x}\|_1 \quad \text{subject to} \quad \mathbf{d} = \mathbf{Op} \mathbf{x}$$

by a set of outer iterations which require to repeatedly solve a weighted least squares problem of the form [2]:

$$\mathbf{x}^{(i+1)} = \arg \min_{\mathbf{x}} \|\mathbf{x}\|_{2, \mathbf{R}^{(i)}}^2 \quad \text{subject to} \quad \mathbf{d} = \mathbf{Op} \mathbf{x}$$

where $\mathbf{R}^{(i)}$ is a diagonal weight matrix whose diagonal elements at iteration i are equal to the absolutes of the model vector $\mathbf{x}^{(i)}$ at iteration i . More specifically the j -th element of the diagonal of $\mathbf{R}^{(i)}$ is

$$R_{j,j}^{(i)} = |x_j^{(i)}|.$$

Examples using `pylops.optimization.sparsity.IRLS`

- *Linear Regression*
- *MP, OMP, ISTA and FISTA*
- *Polynomial Regression*

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 L^0 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 L^2 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

- xinv** [`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 **Op** and the data **d**:

$$\|\mathbf{x}\|_0 \quad \text{subject to} \quad \|\mathbf{Op} \mathbf{x} - \mathbf{b}\|_2^2 \leq \sigma^2,$$

using Orthogonal Matching Pursuit (OMP). This is a very simple iterative algorithm which applies the following step:

$$\begin{aligned} *arg \min * arg \max \Lambda_k &= \Lambda_{k-1} \cup \left\{ j \mid \left| \mathbf{Op}_j^H \mathbf{r}_k \right| \right\} \\ \mathbf{x}_k &= \mathbf{x} \quad \left\| \mathbf{Op}_{\Lambda_k} \mathbf{x} - \mathbf{b} \right\|_2^2 \end{aligned}$$

Note that by choosing `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 \mathbf{x} for the currently selected

basis functions, the vector \mathbf{x} is just updated at the new basis function by taking directly the value from the inner product $\mathbf{Op}_j^H \mathbf{r}_k$.

In this case it is highly recommended 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`

- *MP, OMP, ISTA and FISTA*

`pylops.optimization.sparsity.ISTA`

```
pylops.optimization.sparsity.ISTA(Op, data, niter, eps=0.1, alpha=None, eigsiter=None,
                                  eigstol=0, tol=1e-10, monitorres=False, returninfo=False,
                                  show=False, threshkind='soft', perc=None, callback=None, decay=None, SOP=None, x0=None)
```

Iterative Shrinkage-Thresholding Algorithm (ISTA).

Solve an optimization problem with L^p , $p = 0, 0.5, 1$ regularization, 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** [`int`] Number of iterations
- eps** [`float`, optional] Sparsity damping
- alpha** [`float`, optional] Step size. To guarantee convergence, ensure $\alpha \leq 1/\lambda_{\max}$, where λ_{\max} is the largest eigenvalue of $\mathbf{Op}^H \mathbf{Op}$. If `None`, the maximum eigenvalue is estimated and the optimal step size is chosen as $1/\lambda_{\max}$. If provided, the convergence criterion 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
- decay** [`numpy.ndarray`, optional] Decay factor to be applied to thresholding during iterations

SOp [*pylops.LinearOperator*, optional] Regularization operator (use when solving the analysis problem)

x0: :obj:'numpy.ndarray', optional Initial guess

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 synthesis problem for the operator **Op** and the data **d**:

$$J = \|\mathbf{d} - \mathbf{Op} \mathbf{x}\|_2^2 + \epsilon \|\mathbf{x}\|_p$$

or the analysis problem:

$$J = \|\mathbf{d} - \mathbf{Op} \mathbf{x}\|_2^2 + \epsilon \|\mathbf{SOp}^H \mathbf{x}\|_p$$

if **SOp** is provided. Note that in the first case, **SOp** should be assimilated in the modelling operator (i.e., $\mathbf{Op} = \mathbf{GOp} * \mathbf{SOp}$).

The Iterative Shrinkage-Thresholding Algorithms (ISTA) [1] is used, where $p = 0, 0.5, 1$. This is a very simple iterative algorithm which applies the following step:

$$\mathbf{x}^{(i+1)} = T_{(\epsilon\alpha/2, p)} \left(\mathbf{x}^{(i)} + \alpha \mathbf{Op}^H \left(\mathbf{d} - \mathbf{Op} \mathbf{x}^{(i)} \right) \right)$$

or

$$\mathbf{x}^{(i+1)} = \mathbf{SOp} \left\{ T_{(\epsilon\alpha/2, p)} \mathbf{SOp}^H \left(\mathbf{x}^{(i)} + \alpha \mathbf{Op}^H \left(\mathbf{d} - \mathbf{Op} \mathbf{x}^{(i)} \right) \right) \right\}$$

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 and the threshold changes at every iteration based on the computed percentile.

Examples using `pylops.optimization.sparsity.ISTA`

- *MP, OMP, ISTA and FISTA*
- *03. Solvers*

`pylops.optimization.sparsity.FISTA`

`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, *decay*=None, *SOp*=None, *x0*=None)

Fast Iterative Shrinkage-Thresholding Algorithm (FISTA).

Solve an optimization problem with L^p , $p = 0, 0.5, 1$ regularization, 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** [`int`] Number of iterations
- eps** [`float`, optional] Sparsity damping Step size. To guarantee convergence, ensure $\alpha \leq 1/\lambda_{\max}$, where λ_{\max} is the largest eigenvalue of $\mathbf{Op}^H \mathbf{Op}$. If None, the maximum eigenvalue is estimated and the optimal step size is chosen as $1/\lambda_{\max}$. If provided, the convergence criterion 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
- threshkind** [`str`, optional] Kind of thresholding ('hard', 'soft', 'half', '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
- decay** [`numpy.ndarray`, optional] Decay factor to be applied to thresholding during iterations
- SOp** [`pylops.LinearOperator`, optional] Regularization operator (use when solving the analysis problem)
- x0**: `obj:'numpy.ndarray'`, optional Initial guess

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 synthesis problem for the operator \mathbf{Op} and the data \mathbf{d} :

$$J = \|\mathbf{d} - \mathbf{Op} \mathbf{x}\|_2^2 + \epsilon \|\mathbf{x}\|_p$$

or the analysis problem:

$$J = \|\mathbf{d} - \mathbf{Op} \mathbf{x}\|_2^2 + \epsilon \|\mathbf{SOp}^H \mathbf{x}\|_p$$

if \mathbf{SOp} is provided.

The Fast Iterative Shrinkage-Thresholding Algorithm (FISTA) [1] is used, where $p = 0, 0.5, 1$. 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`

- *MP, OMP, ISTA and FISTA*
- *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 \mathbf{Op} and a sparsifying transform \mathbf{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, final status of solver**
 - 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

$$\|\mathbf{x}\|_1 \quad \text{subject to} \quad \|\mathbf{Op} \mathbf{S}^H \mathbf{x} - \mathbf{b}\|_2^2 \leq \sigma,$$

while the second problem is the *l_1 -regularized least-squares or LASSO* problem and its cost function is

$$\|\mathbf{Op} \mathbf{S}^H \mathbf{x} - \mathbf{b}\|_2^2 \quad \text{subject to} \quad \|\mathbf{x}\|_1 \leq \tau$$

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 L^2 - L^1 norms.

Solve an unconstrained system of equations with mixed L^2 and L^1 regularization terms given the operator `Op`, a list of L^1 regularization terms `RegsL1`, and an optional list of L^2 regularization terms `RegsL2`.

Parameters

Op [`pylops.LinearOperator`] Operator to invert

RegsL1 [`list`] L^1 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 L^2 regularization operators (if `None`, L^2 regularization is not added to the problem)

dataregsL2 [`list`, optional] L^2 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] L^1 Regularization dampings (must have the same number of elements as RegsL1)

epsRL2s [list] L^2 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 (L^2 and L^1) regularization terms $\mathbf{R}_{2,i}$ and $\mathbf{R}_{1,i}$, respectively:

$$J = \frac{\mu}{2} \|\mathbf{d} - \mathbf{Op} \mathbf{x}\|_2^2 + \frac{1}{2} \sum_i \epsilon_{\mathbf{R}_{2,i}} \|\mathbf{d}_{\mathbf{R}_{2,i}} - \mathbf{R}_{2,i} \mathbf{x}\|_2^2 + \sum_i \epsilon_{\mathbf{R}_{1,i}} \|\mathbf{R}_{1,i} \mathbf{x}\|_1$$

where μ is the reconstruction damping, $\epsilon_{\mathbf{R}_{2,i}}$ are the damping factors used to weight the different L^2 regularization terms of the cost function and $\epsilon_{\mathbf{R}_{1,i}}$ are the damping factors used to weight the different L^1 regularization terms of the cost function.

The generalized Split-Bergman algorithm [1] 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:

$$J = \frac{\mu}{2} \|\mathbf{d} - \mathbf{Op} \mathbf{x}\|_2^2 + \frac{1}{2} \sum_i \epsilon_{\mathbf{R}_{2,i}} \|\mathbf{d}_{\mathbf{R}_{2,i}} - \mathbf{R}_{2,i} \mathbf{x}\|_2^2 + \sum_i \|\mathbf{y}_i\|_1 \quad \text{subject to} \quad \mathbf{y}_i = \mathbf{R}_{1,i} \mathbf{x} \quad \forall i$$

and solved as follows:

$$\begin{aligned}
 (\mathbf{x}^{k+1}, \mathbf{y}_i^{k+1}) =_{\mathbf{x}, \mathbf{y}_i} & \|\mathbf{d} - \mathbf{Op} \mathbf{x}\|_2^2 + \frac{1}{2} \sum_i \epsilon_{\mathbf{R}_{2,i}} \|\mathbf{d}_{\mathbf{R}_{2,i}} - \mathbf{R}_{2,i} \mathbf{x}\|_2^2 \\
 *argmin & + \frac{1}{2} \sum_i \epsilon_{\mathbf{R}_{1,i}} \|\mathbf{y}_i - \mathbf{R}_{1,i} \mathbf{x} - \mathbf{b}_i^k\|_2^2 \\
 & + \sum_i \|\mathbf{b}_i^k\|_1 \\
 \mathbf{b}_i^{k+1} = \mathbf{b}_i^k & + (\mathbf{R}_{1,i} \mathbf{x}^{k+1} - \mathbf{y}_i^{k+1})
 \end{aligned} \tag{3.1}$$

The `scipy.sparse.linalg.lsqr` solver and a fast shrinkage algorithm are used within an inner loop to solve the first step. The entire procedure is repeated `niter_outer` times until convergence.

Examples using `pylops.optimization.sparsity.SplitBregman`

- *Total Variation (TV) Regularization*
- *05. Image deblurring*
- *16. CT Scan Imaging*

3.8.3 Applications

Wave-Equation processing

<code>SeismicInterpolation(data, nrec, iava[, ...])</code>	Seismic interpolation (or regularization).
<code>Deghosting(p, nt, nr, dt, dr, vel, zrec[, ...])</code>	Wavefield deghosting.
<code>WavefieldDecomposition(p, vz, nt, nr, dt, ...)</code>	Up-down wavefield decomposition.
<code>MDD(G, d[, dt, dr, nfm, wav, twosided, ...])</code>	Multi-dimensional deconvolution.
<code>Marchenko(R[, R1, dt, nt, dr, nfm, wav, ...])</code>	Marchenko redatuming
<code>LSM(z, x, t, srcs, recs, vel, wav, wavcenter)</code>	Least-squares Migration (LSM).

`pylops.waveeqprocessing.SeismicInterpolation`

```
pylops.waveeqprocessing.SeismicInterpolation(data, nrec, iava, iava1=None,
                                                kind='fk', nffts=None, sampling=None,
                                                spataxis=None, spatlaxis=None,
                                                taxis=None, paxis=None, plaxis=None,
                                                centeredh=True, nwins=None,
                                                nwin=None, nover=None, design=False,
                                                engine='numba', dottest=False,
                                                **kwargs_solver)
```

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 $[n_{ry} (\times n_{rx} \times n_t)]$
- nrec** [`int` or `tuple`] Number of elements in the regularly sampled (reconstructed) spatial array, n_{ry} for 2-dimensional data and (n_{ry}, n_{rx}) 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`, `chirpradon-linear`, `radon-parabolic`, `radon-hyperbolic`, `sliding`, or `chirp-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-linear'`

spataxis [`np.ndarray`, optional] First spatial axis. Required for `kind='radon-linear'`, `kind='chirpradon-linear'`, `kind='radon-parabolic'`, `kind='radon-hyperbolic'`, can also be provided instead of sampling for `kind='fk'`

spat1axis [`np.ndarray`, optional] Second spatial axis. Required for `kind='radon-linear'`, `kind='chirpradon-linear'`, `kind='radon-parabolic'`, `kind='radon-hyperbolic'`, can also be provided instead of sampling for `kind='fk'`

taxis [`np.ndarray`, optional] Time axis. Required for `kind='radon-linear'`, `kind='chirpradon-linear'`, `kind='radon-parabolic'`, `kind='radon-hyperbolic'`, can also be provided instead of sampling for `kind='fk'`

paxis [`np.ndarray`, optional] First Radon axis. Required for `kind='radon-linear'`, `kind='chirpradon-linear'`, `kind='radon-parabolic'`, `kind='radon-hyperbolic'`, `kind='sliding'`, and `kind='chirp-sliding'`

plaxis [`np.ndarray`, optional] Second Radon axis. Required for `kind='radon-linear'`, `kind='chirpradon-linear'`, `kind='radon-parabolic'`, `kind='radon-hyperbolic'`, `kind='sliding'`, and `kind='chirp-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'` and `kind='chirp-sliding'`

nwin [`int` or `tuple`, optional] Number of samples of window. Required for `kind='sliding'` and `kind='chirp-sliding'`

nover [`int` or `tuple`, optional] Number of samples of overlapping part of window. Required for `kind='sliding'` and `kind='chirp-sliding'`

design [`bool`, optional] Print number of sliding window (True) or not (False) when using `kind='sliding'` and `kind='chirp-sliding'`

engine [`str`, optional] Engine used for Radon computations (`numpy/numba` for `Radon2D` and `Radon3D` or `numpy/fftw` for `ChirpRadon2D` and `ChirpRadon3D`)

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

reccdata [`np.ndarray`] Reconstructed data of size $[n_{R_y} (\times n_{R_x} \times n_t)]$

recrec [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

$$\mathbf{y} = \mathbf{R}\mathbf{x}$$

where a restriction or interpolation operator is applied along the spatial direction(s). Here $\mathbf{y} = [\mathbf{y}_{R1}^T, \mathbf{y}_{R2}^T, \dots, \mathbf{y}_{RNT}^T]^T$ where each vector \mathbf{y}_{Ri} contains all time samples recorded in the seismic data at the specific receiver R_i . Similarly, $\mathbf{x} = [\mathbf{x}_{r1}^T, \mathbf{x}_{r2}^T, \dots, \mathbf{x}_{rM}^T]$, 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 $J = \|\mathbf{y} - \mathbf{R}\mathbf{x}\|_2 + \epsilon \nabla \|\mathbf{x}\|_2$ where ∇ 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 $J = \|\mathbf{y} - \mathbf{R}\mathbf{F}\mathbf{x}\|_2$ where \mathbf{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 \mathbf{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 \mathbf{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, *sptransf*=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 $[n_{rx} (\times n_{ry}) \times n_t]$ (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)

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 constituents

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*

win [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 angle

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. If None, directly inferred from *p*

****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 [1] 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 $p_d(x, t)$):

$$p^+ - p_d = e^{-jk_z 2z_{\text{rec}}} p^-$$

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:

$$\mathbf{p} - \mathbf{p}_d = (\mathbf{I} + \Phi) \mathbf{p}^-$$

where Φ is one-way propagator implemented via the `pylops.waveeqprocessing.PhaseShift` operator.

Examples using `pylops.waveeqprocessing.Deghosting`

- 13. *Deghosting*

`pylops.waveeqprocessing.WavefieldDecomposition`

`pylops.waveeqprocessing.WavefieldDecomposition` (*p*, *vz*, *nt*, *nr*, *dt*, *dr*, *rho*, *vel*, *nffts*=(None, None, None), *critical*=100.0, *ntaper*=10, *scaling*=1.0, *kind*='inverse', *restriction*=None, *sptransf*=None, *solver*=<function lsqr>, *dotest*=False, *dtype*='complex128', **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 $[n_{r_x} (\times n_{r_y}) \times n_t]$ (or $[n_{r_{x,\text{sub}}} (\times n_{r_{y,\text{sub}}}) \times n_t]$ in case a *restriction* operator is provided. Note that $n_{r_{x,\text{sub}}}$ (and $n_{r_{y,\text{sub}}}$) 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 ρ along the receiver array (must be constant)

vel [float] Velocity c 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 $\frac{f(k_x)}{c}$ 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 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

Raises

KeyError If 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]:

$$\begin{bmatrix} \hat{p}^+ \\ \hat{p}^- \end{bmatrix} (k_x, \omega) = \frac{1}{2} \begin{bmatrix} 1 & \frac{\omega \rho}{k_z} \\ 1 & -\frac{\omega \rho}{k_z} \end{bmatrix} \begin{bmatrix} \hat{p} \\ \hat{v}_z \end{bmatrix} (k_x, \omega)$$

if kind='analytical' or alternatively by solving the equation in `ptcpy.waveeqprocessing.UpDownComposition2D` as an inverse problem, if kind='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:

$$\begin{bmatrix} \mathbf{p} \\ s\mathbf{v}_z \end{bmatrix} = \begin{bmatrix} \mathbf{R}\mathbf{F} & 0 \\ 0 & s\mathbf{R}\mathbf{F} \end{bmatrix} \mathbf{W} \begin{bmatrix} \mathbf{F}^H \mathbf{S} & 0 \\ 0 & \mathbf{F}^H \mathbf{S} \end{bmatrix} \mathbf{p}^\pm$$

where \mathbf{R} is a `ptcpy.basicoperators.Restriction` operator and \mathbf{S} is sparsifying transform operator (e.g., `ptcpy.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, nfmaz=None, wav=None,
                             twosided=True, causality_precond=False, adjoint=False, psf=False,
                             dtype='float64', dottest=False, saveGt=True, add_negative=True,
                             smooth_precond=0, fftengine='numpy', **kwargs_solver)
```

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 `twosided=False` or `twosided=True` and `add_negative=True` (with only positive times) or size $[n_s \times n_r \times 2n_t - 1]$ for `twosided=True` and `add_negative=False` (with both positive and negative times)
- d** [`numpy.ndarray`] Data in time domain $[n_s (\times n_{vs}) \times n_t]$ if `twosided=False` or `twosided=True` and `add_negative=True` (with only positive times) or size $[n_s (\times n_{vs}) \times 2n_t - 1]$ if `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 `twosided=True`.
- causality_precond** [`bool`, optional] Apply causality mask (`True`) or not (`False`)
- smooth_precond** [`int`, optional] Length of smoothing to apply to causality preconditioner
- 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
- fftengine** [`str`, optional] Engine used for fft computation (`numpy`, `scipy` or `fftw`)
- **kwargs_solver** Arbitrary keyword arguments for chosen solver (`scipy.sparse.linalg.cg` and `pylops.optimization.solver.cg` are used as default for `numpy` and `cupy` *data*, respectively)

Returns

- minv** [`numpy.ndarray`] Inverted model of size $[n_r (\times n_{vs}) \times n_t]$ for `twosided=False` or $[n_r (\times n_{vs}) \times 2n_t - 1]$ for `twosided=True`
- madj** [`numpy.ndarray`] Adjoint model of size $[n_r (\times n_{vs}) \times n_t]$ for `twosided=False` or $[n_r (\times n_r) \times 2n_t - 1]$ for `twosided=True`
- psfinv** [`numpy.ndarray`] Inverted psf of size $[n_r \times n_r \times n_t]$ for `twosided=False` or $[n_r \times n_r \times 2n_t - 1]$ for `twosided=True`
- psfadj** [`numpy.ndarray`] Adjoint psf of size $[n_r \times n_r \times n_t]$ for `twosided=False` or $[n_r \times n_r \times 2n_t - 1]$ 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

$$\mathbf{d} = \mathbf{D}\mathbf{m}$$

or, equivalently, by means of its normal equation

$$\mathbf{m} = (\mathbf{D}^H \mathbf{D})^{-1} \mathbf{D}^H \mathbf{d}$$

where $\mathbf{D}^H \mathbf{D}$ is the PSF.

Examples using `pylops.waveeqprocessing.MDD`

- [09. Multi-Dimensional Deconvolution](#)

`pylops.waveeqprocessing.Marchenko`

```
class pylops.waveeqprocessing.Marchenko (R, Rl=None, dt=0.004, nt=None, dr=1.0, nfm  

max=None, wav=None, toff=0.0, nsmooth=10,  

dtype='float64', saveRt=True, prescaled=False,  

fftengine='numpy')
```

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 $[n_s \times n_r \times n_t(n_{f_{\max}})]$. 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

nfm [`int`, optional] Index of max frequency to include in deconvolution process

wav [`numpy.ndarray`, optional] Wavelet to apply to direct arrival when created using `trav`

toff [`float`, optional] Time-offset to apply to traveltimes

nsmooth [`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 $R.H$ to speed up the computation of adjoint of `pylops.signalprocessing.Fredholm1` (`True`) or create $R.H$ 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 `pylops.waveeqprocessing.MDC` operator. In case `prescaled=True`, the R is assumed to have been pre-scaled by the user.

fftengine [`str`, optional] New in version 1.17.0.

Engine used for fft computation (`numpy`, `scipy` or `fftw`)

Raises

TypeError If `t` is not `numpy.ndarray`.

See also:

MDC Multi-dimensional convolution

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:

$$\begin{bmatrix} \Theta R f_d^+ \\ 0 \end{bmatrix} = I - \begin{bmatrix} 0 & \Theta R \\ \Theta R^* & 0 \end{bmatrix} \begin{bmatrix} f^- \\ f_m^+ \end{bmatrix}$$

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

<code>__init__(R[, Rl, dt, nt, dr, nfm, wav, ...])</code>	Initialize self.
<code>apply_multiplepoints(trav[, G0, nfft, rtm, ...])</code>	Marchenko redatuming for multiple points
<code>apply_onepoint(trav[, G0, nfft, rtm, ...])</code>	Marchenko redatuming for one point

apply_onepoint (*trav*, *G0=None*, *nfft=None*, *rtm=False*, *greens=False*, *dottest=False*, *fast=None*, *usematmul=False*, ***kwargs_solver*)

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 $[n_r \times 1]$

G0 [`numpy.ndarray`, optional] Direct arrival in time domain of size $[n_r \times n_t]$ (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

fast [`bool`] *Deprecated*, will be removed in v2.0.0

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.

****kwargs_solver** Arbitrary keyword arguments for chosen solver (`scipy.sparse.linalg.lsqr` and `pylops.optimization.solver.cgls` are used as default for numpy and cupy *data*, respectively)

Returns

f1_inv_minus [`numpy.ndarray`] Inverted upgoing focusing function of size $[n_r \times n_t]$

f1_inv_plus [`numpy.ndarray`] Inverted downgoing focusing function of size $[n_r \times n_t]$

p0_minus [`numpy.ndarray`] Single-scattering standard redatuming upgoing Green's function of size $[n_r \times n_t]$

g_inv_minus [`numpy.ndarray`] Inverted upgoing Green's function of size $[n_r \times n_t]$

g_inv_plus [`numpy.ndarray`] Inverted downgoing Green's function of size $[n_r \times n_t]$

apply_multiplepoints (*trav*, *G0=None*, *nfft=None*, *rtm=False*, *greens=False*, *dottest=False*, *usematmul=False*, ***kwargs_solver*)

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

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

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.

****kwargs_solver** Arbitrary keyword arguments for chosen solver (`scipy.sparse.linalg.lsqr` and `pylops.optimization.solver.cgls` are used as default for numpy and cupy data, respectively)

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

<code>__init__(z, x, t, srcs, recs, vel, wav, ...)</code>	Initialize self.
<code>solve(d[, solver])</code>	Solve least-squares migration equations with chosen solver

solve (*d*, *solver*=<function lsqr>, ***kwargs_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

<code>poststack.PoststackInversion(data, wav[, ...])</code>	Post-stack linearized seismic inversion.
<code>prestack.PrestackInversion(data, theta, wav)</code>	Pre-stack linearized seismic inversion.

pylops.avo.poststack.PoststackInversion

`pylops.avo.poststack.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 $[n_{t_0} (\times n_x \times n_y)]$

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 $[n_{t_0} \times n_h]$ use as non-stationary wavelet

m0 [np.ndarray, optional] Background model of size $[n_{t_0} (\times n_x \times n_y)]$

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 $[n_{t_0} (\times n_x \times n_y)]$

datar [np.ndarray] Residual data (i.e., data - background data) of size $[n_{t_0} (\times n_x \times n_y)]$

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 $\mathbf{W}^T \mathbf{d} = (\mathbf{W}^T \mathbf{W} + \epsilon_I^2 \mathbf{I}) \mathbf{A} \mathbf{I}$ 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, kind='centered', vsvp=0.5,
                                     **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 $[(n_{\text{ins}} \times) n_{t_0} \times n_{\theta} (\times n_x \times n_y)]$

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 $[n_{t_0} \times n_m (\times n_x \times n_y)]$

linearization [{"akirich", "fatti", "PS"} or list, optional]

- "akirich": Aki-Richards. See `pylops.avo.avo.akirichards`.
- "fatti": Fatti. See `pylops.avo.avo.fatti`.
- "PS": PS. See `pylops.avo.avo.ps`.
- List which is a combination of previous options (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

dottest [`bool`, optional] Apply dot-test

returnres [`bool`, optional] Return residuals

epsRL1 [`float`, optional] Damping factor for additional blockiness regularization term

kind [`str`, optional] Derivative kind (`forward` or `centered`).

vsvp [`float` or `np.ndarray`] V_S/V_P ratio (constant or time/depth variant)

****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 $[n_{t_0} \times n_m (\times n_x \times n_y)]$

data [`np.ndarray`] Residual data (i.e., data - background data) of size $[n_{t_0} \times n_\theta (\times n_x \times n_y)]$

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.avo.poststack.PoststackInversion` for more details.

Examples using `pylops.avo.prestack.PrestackInversion`

- 08. *Pre-stack (AVO) inversion*

3.9 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.

3.9.1 Dot-test

<code>dottest(Op[, nr, nc, rtol, complexflag, ...])</code>	Dot test.
--	-----------

pylops.utils.dottest

```
pylops.utils.dottest(Op, nr=None, nc=None, rtol=1e-06, complexflag=0, raiseerror=True,
                    verb=False, backend='numpy', atol=1e-21, tol=None)
```

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)

rtol [*float*, optional] Relative dottest tolerance .. versionadded:: 1.18.1

complexflag [*bool*, optional] Generate random vectors with

- 0: Real entries for model and data
- 1: Complex entries for model and real entries for data
- 2: Real entries for model and complex entries for data
- 3: Complex entries for model and data

raiseerror [*bool*, optional] Raise error or simply return `False` when dottest fails

verb [*bool*, optional] Verbosity

backend [*str*, optional] Backend used for dot test computations (numpy or cupy). This parameter will be used to choose how to create the random vectors.

atol [*float*, optional] Absolute dottest tolerance .. versionadded:: 1.18.1

tol [*float*, optional] Dottest tolerance .. deprecated:: 2.0.0

Use `rtol` instead.

Raises

ValueError If dot-test is not verified within chosen tolerances.

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:

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

Examples using `pylops.utils.dottest`

- *Normal Moveout (NMO) Correction*
- *Seislet transform*
- *02. The Dot-Test*

3.9.2 Describe

<code>describe(Op)</code>	Describe a PyLops operator
---------------------------	----------------------------

`pylops.utils.describe.describe`

`pylops.utils.describe.describe(Op)`

Describe a PyLops operator

New in version 1.17.0.

Convert a PyLops operator into a `sympy` mathematical formula. This routine is useful both for debugging and educational purposes.

Note that users can add a name to each operator prior to running the `describe` method, i.e. `Op.name = 'Opname'`. Alternatively, each of the PyLops operator that composes the operator `Op` is automatically assigned a name. Moreover, note that the symbols T and \dagger are used in the mathematical expressions to indicate transposed and adjoint operators, respectively.

Parameters

Op [`pylops.LinearOperator`] Linear Operator to describe

Examples using `pylops.utils.describe.describe`

- [Describe](#)

3.9.3 Estimators

<code>trace_hutchinson(Op[, neval, batch_size, ...])</code>	Trace of linear operator using the Hutchinson method.
<code>trace_hutchpp(Op[, neval, sampler, backend])</code>	Trace of linear operator using the Hutch++ method.
<code>trace_nahutchpp(Op[, neval, sampler, c1, ...])</code>	Trace of linear operator using the NA-Hutch++ method.

`pylops.utils.trace_hutchinson`

`pylops.utils.trace_hutchinson(Op, neval=None, batch_size=None, sampler='rademacher', backend='numpy')`

Trace of linear operator using the Hutchinson method.

Returns an estimate of the trace of a linear operator using the Hutchinson method [1].

Parameters

neval [`int`, optional] Maximum number of matrix-vector products compute. Defaults to 10% of `shape[1]`.

batch_size [`int`, optional] Vectorize computations by sampling sketching matrices instead of vectors. Set this value to as high as permitted by memory, but there is no guarantee of speedup. Coerced to never exceed `neval`. When using “unitvector” as sampler, is coerced to not exceed `shape[1]`. Defaults to 100 or `neval`.

sampler [`str`, optional] Sample sketching matrices from the following distributions:

- “gaussian”: Mean zero, unit variance Gaussian.

- “rayleigh”: Sample from mean zero, unit variance Gaussian and normalize the columns.
- “rademacher”: Random sign.
- “unitvector”: Samples from the unit vectors \mathbf{e}_i without replacement.

backend [`str`, optional] Backend used to densify matrix (`numpy` or `cupy`). Note that this must be consistent with how the operator has been created.

Returns

trace [`self.dtype`] Operator trace.

Raises

ValueError If `neval` is smaller than 3.

NotImplementedError If the `sampler` is not one of the available samplers.

Notes

Let $m = \text{shape}[1]$ and $k = \text{neval}$. This algorithm estimates the trace via

$$\frac{1}{k} \sum_{i=1}^k \mathbf{z}_i^T \mathbf{O} \mathbf{p} \mathbf{z}_i$$

where vectors \mathbf{z}_i are sampled according to the sampling function. See [2] for a description of the variance and ϵ -approximation of different samplers.

Prefer the Rademacher sampler if the goal is to minimize variance, but the Gaussian for a better probability of approximating the correct value. Use the Unit Vector approach if you are sampling a large number of `neval` (compared to `shape[1]`), especially if the operator is highly-structured.

pylops.utils.trace_hutchpp

`pylops.utils.trace_hutchpp` (*Op*, *neval=None*, *sampler='rademacher'*, *backend='numpy'*)

Trace of linear operator using the Hutch++ method.

Returns an estimate of the trace of a linear operator using the Hutch++ method [1].

Parameters

neval [`int`, optional] Maximum number of matrix-vector products compute. Defaults to 10% of `shape[1]`.

sampler [`str`, optional] Sample sketching matrices from the following distributions:

- “gaussian”: Mean zero, unit variance Gaussian.
- “rayleigh”: Sample from mean zero, unit variance Gaussian and normalize the columns.
- “rademacher”: Random sign.

backend [`str`, optional] Backend used to densify matrix (`numpy` or `cupy`). Note that this must be consistent with how the operator has been created.

Returns

trace [`self.dtype`] Operator trace.

Raises

ValueError If `neval` is smaller than 3.

NotImplementedError If the `sampler` is not one of the available samplers.

Notes

This function follows Algorithm 1 of [1]. Let $m = \text{shape}[1]$ and $k = \text{neval}$.

1. Sample sketching matrices $\mathbf{S} \in \mathbb{R}^{m \times \lfloor k/3 \rfloor}$, and $\mathbf{G} \in \mathbb{R}^{m \times \lfloor k/3 \rfloor}$, from sub-Gaussian distributions.
2. Compute reduced QR decomposition of $\mathbf{O} \mathbf{p} \mathbf{S}$, retaining only \mathbf{Q} .
3. Return $\text{tr}(\mathbf{Q}^T \mathbf{O} \mathbf{p} \mathbf{Q}) + \frac{1}{\lfloor k/3 \rfloor} \text{tr}(\mathbf{G}^T (\mathbf{I} - \mathbf{Q} \mathbf{Q}^T) \mathbf{O} \mathbf{p} (\mathbf{I} - \mathbf{Q} \mathbf{Q}^T) \mathbf{G})$

Use the Rademacher sampler unless you know what you are doing.

`pylops.utils.trace_nahutchpp`

```
pylops.utils.trace_nahutchpp(Op, neval=None, sampler='rademacher',
                             c1=0.16666666666666666, c2=0.3333333333333333, back-
                             end='numpy')
```

Trace of linear operator using the NA-Hutch++ method.

Returns an estimate of the trace of a linear operator using the Non-Adaptive variant of Hutch++ method [1].

Parameters

neval [`int`, optional] Maximum number of matrix-vector products compute. Defaults to 10% of `shape[1]`.

sampler [`str`, optional] Sample sketching matrices from the following distributions:

- “gaussian”: Mean zero, unit variance Gaussian.
- “rayleigh”: Sample from mean zero, unit variance Gaussian and normalize the columns.
- “rademacher”: Random sign.

c1 [`float`, optional] Fraction of `neval` for sketching matrix \mathbf{S} .

c2 [`float`, optional] Fraction of `neval` for sketching matrix \mathbf{R} . Must be larger than `c1`, ideally by a factor of at least 2.

backend [`str`, optional] Backend used to densify matrix (`numpy` or `cupy`). Note that this must be consistent with how the operator has been created.

Returns

trace [`self.dtype`] Operator trace.

Raises

ValueError If `neval` not large enough to accomodate `c1` and `c2`.

NotImplementedError If the `sampler` is not one of the available samplers.

Notes

This function follows Algorithm 2 of [1]. Let $m = \text{shape}[1]$ and $k = \text{neval}$.

1. Fix constants c_1, c_2, c_3 such that $c_1 < c_2$ and $c_1 + c_2 + c_3 = 1$.

2. Sample sketching matrices $\mathbf{S} \in \mathbb{R}^{m \times c_1 k}$, $\mathbf{R} \in \mathbb{R}^{m \times c_2 k}$, and $\mathbf{G} \in \mathbb{R}^{m \times c_3 k}$ from sub-Gaussian distributions.
3. Compute $\mathbf{Z} = \mathbf{Op} \mathbf{R}$, $\mathbf{W} = \mathbf{Op} \mathbf{S}$, and $\mathbf{Y} = (\mathbf{S}^T \mathbf{Z})^+$, where $+$ denotes the Moore–Penrose inverse.
4. Return $\text{tr}(\mathbf{Y} \mathbf{W}^T \mathbf{Z}) + \frac{1}{c_3 k} [\text{tr}(\mathbf{G}^T \mathbf{Op} \mathbf{G}) - \text{tr}(\mathbf{G}^T \mathbf{Z} \mathbf{Y} \mathbf{W}^T \mathbf{G})]$

The default values for c_1 and c_2 are set to $1/6$ and $1/3$, respectively, but [1] suggests $1/4$ and $1/2$.

Use the Rademacher sampler unless you know what you are doing.

3.9.4 Scalability test

<code>scalability_test(Op, x[, workers, forward])</code>	Scalability test.
--	-------------------

pylops.utils.scalability_test

`pylops.utils.scalability_test (Op, x, workers=[1, 2, 4], forward=True)`

Scalability test.

Small auxiliary routine to test the performance of operators using multiprocessing. This helps identifying the maximum number of workers beyond which no performance gain is observed.

Parameters

Op [`pylops.LinearOperator`] Operator to test. It must allow for multiprocessing.

x [`numpy.ndarray`, optional] Input vector.

workers [`list`, optional] Number of workers to test out.

forward [`bool`, optional] Apply forward (`True`) or adjoint (`False`)

Returns

compute_times [`list`] Compute times as function of workers

speedup [`list`] Speedup as function of workers

Examples using `pylops.utils.scalability_test`

- *Operators with Multiprocessing*

3.9.5 Synthetics

<code>seismicevents.makeaxis(par)</code>	Create axes t, x, and y axes
<code>seismicevents.linear2d(x, t, v, t0, theta, ...)</code>	Linear 2D events
<code>seismicevents.parabolic2d(x, t, t0, px, pxx, ...)</code>	Parabolic 2D events
<code>seismicevents.hyperbolic2d(x, t, t0, vrms, ...)</code>	Hyperbolic 2D events
<code>seismicevents.linear3d(x, y, t, v, t0, ...)</code>	Linear 3D events
<code>seismicevents.hyperbolic3d(x, y, t, t0, ...)</code>	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

t2 [numpy.ndarray] Symmetric time axis

x [numpy.ndarray] x axis

y [numpy.ndarray] y axis (None, if *oy*, *dy* or *ny* are not provided)

Examples

```
>>> 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`

- *1D, 2D and 3D Sliding*
- *2D Patching*
- *Chirp Radon Transform*
- *Multi-Dimensional Convolution*
- *Normal Moveout (NMO) Correction*
- *PhaseShift operator*
- *Spread How-to*
- *Synthetic seismic*
- *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 $[n_x \times n_t]$
dwav [`numpy.ndarray`] data with wavelet of size $[n_x \times n_t]$

Notes

Each event is created using the following relation:

$$t_i(x) = t_{0,i} + p_{x,i}x$$

where $p_{x,i} = \sin(\theta_i)/v$

Examples using `pylops.utils.seismicevents.linear2d`

- *Chirp Radon Transform*
- *Spread How-to*
- *Synthetic seismic*
- *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`

- *1D, 2D and 3D Sliding*
- *2D Patching*
- *Synthetic seismic*
- *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:

$$t_i(x) = \sqrt{t_{0,i}^2 + \frac{x^2}{v_{\text{rms},i}^2}}$$

Examples using `pylops.utils.seismicevents.hyperbolic2d`

- *Multi-Dimensional Convolution*
- *Normal Moveout (NMO) Correction*
- *PhaseShift operator*
- *Synthetic seismic*
- *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 $p_{x,i} = \frac{1}{v} \sin(\theta_i) \cos(\phi_i)$ and $p_{y,i} = \frac{1}{v} \sin(\theta_i) \sin(\phi_i)$.

Examples using `pylops.utils.seismicevents.linear3d`

- *Chirp Radon Transform*
- *Synthetic seismic*

`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:

$$t_i(x, y) = \sqrt{t_{0,i}^2 + \frac{x^2}{v_{\text{rms}_x,i}^2} + \frac{y^2}{v_{\text{rms}_y,i}^2}}$$

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`

- *1D, 2D and 3D Sliding*
- *PhaseShift operator*
- *Synthetic seismic*

<code>marchenko.directwave(wav, trav, nt, dt[, ...])</code>	Analytical direct wave in acoustic media
---	--

pylops.waveeqprocessing.marchenko.directwave

`pylops.waveeqprocessing.marchenko.directwave(wav, trav, nt, dt, nfft=None, dist=None, kind='2d', derivative=True)`

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 created using `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 $[n_r \times 1]$
- 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 $[n_r \times 1]$
- 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 $[n_t \times n_r]$

Notes

The analytical Green's function in 2D [1] is :

$$G^{2D}(\mathbf{r}) = -\frac{i}{4}H_0^{(1)}(k|\mathbf{r}|)$$

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 `derivative=True`.

3.9.6 Signal-processing

<code>signalprocessing.convmtx(h, n)</code>	Convolution matrix
---	--------------------

Continued on next page

Table 56 – continued from previous page

<code>signalprocessing.</code>	Convolution matrix from a bank of filters
<code>nonstationary_convmtx(H, n)</code>	
<code>signalprocessing.slope_estimate(d[, dz, dx, ...])</code>	Local slope estimation

pylops.utils.signalprocessing.convmtx

`pylops.utils.signalprocessing.convmtx(h, n)`

Convolution matrix

Equivalent of [MATLAB's convmtx function](#). Makes a dense convolution matrix **C** such that the dot product `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 $\text{len}(h) \geq n$)

pylops.utils.signalprocessing.nonstationary_convmtx

`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 `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 $[n_{\text{filters}} \times n_h]$)

n [`int`] Number of columns of convolution matrix

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=1.0, dx=1.0, smooth=5, eps=0)`

Local slope estimation

Local slopes are estimated using the *Structure Tensor* algorithm [1]. Slopes are returned as $\tan \theta$, defined in a RHS coordinate system with z -axis pointing upward.

Note: For stability purposes, it is important to ensure that the orders of magnitude of the samplings are similar.

Parameters

d [np.ndarray] Input dataset of size $n_z \times n_x$

dz [float] Sampling in z -axis, Δz

Warning: Since version 1.17.0, defaults to 1.0.

dx [float] Sampling in x -axis, Δx

Warning: Since version 1.17.0, defaults to 1.0.

smooth [float, optional] Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.

Warning: Default changed in version 1.17.0 to 5 from previous value of 20.

eps [float, optional] New in version 1.17.0.

Regularization term. All slopes where $|g_{zx}| < \epsilon \max |g_{zx}|$ are set to zero. All anisotropies where $\lambda_{\max} < \epsilon$ are also set to zero. See Notes. When using with small values of `smooth`, start from a very small number (e.g. 1e-10) and start increasing by a power of 10 until results are satisfactory.

Returns

slopes [np.ndarray] Estimated local slopes. Unit is that of $\Delta z / \Delta x$.

Warning: Prior to version 1.17.0, erroneously returned angles in radians instead of slopes.

anisotropies [np.ndarray] Estimated local anisotropies: $1 - \lambda_{\min} / \lambda_{\max}$

Note: Since 1.17.0, changed name from `linearity` to `anisotropies`. Definition remains the same.

Notes

For each pixel of the input dataset **d** the local gradients $g_z = \frac{\partial \mathbf{d}}{\partial z}$ and $g_x = \frac{\partial \mathbf{d}}{\partial x}$ are computed and used to define the following three quantities:

$$\begin{aligned}
 g_{zz} &= \left(\frac{\partial \mathbf{d}}{\partial z} \right)^2 \\
 g_{xx} &= \left(\frac{\partial \mathbf{d}}{\partial x} \right)^2 \\
 g_{zx} &= \frac{\partial \mathbf{d}}{\partial z} \frac{\partial \mathbf{d}}{\partial x}
 \end{aligned} \tag{3.4}$$

They are then spatially smoothed and at each pixel their smoothed versions are arranged in a 2×2 matrix called the *smoothed gradient-square tensor*:

$$\mathbf{G} = \begin{bmatrix} g_{zz} & g_{zx} \\ g_{zx} & g_{xx} \end{bmatrix}$$

Local slopes can be expressed as $p = \frac{\lambda_{\max} - g_{zz}}{g_{zx}}$, where λ_{\max} is the largest eigenvalue of \mathbf{G} .

Moreover, we can obtain a measure of local anisotropy, defined as

$$a = 1 - \lambda_{\min}/\lambda_{\max}$$

where λ_{\min} is the smallest eigenvalue of \mathbf{G} . A value of $a = 0$ indicates perfect isotropy whereas $a = 1$ indicates perfect anisotropy.

Examples using `pylops.utils.signalprocessing.slope_estimate`

- *Seislet transform*
- *Slope estimation via Structure Tensor algorithm*

3.9.7 Tapers

<code>tapers.taper2d(nt, nmask, ntap[, tapertype])</code>	2D taper
<code>tapers.taper3d(nt, nmask, ntap[, tapertype])</code>	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 `ntap` along the first (and possibly second) dimensions

Parameters

- nt** [`int`] Number of samples along second dimension
- nmask** [`int`] Number of samples along first dimension
- ntap** [`int` or `list`] Number of samples of tapering at edges of first dimension (or both dimensions).
- 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`

- *PhaseShift operator*
- *Tapers*

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 `ntap` 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`

- *Multi-Dimensional Convolution*
- *PhaseShift operator*
- *Tapers*
- *09. Multi-Dimensional Deconvolution*

3.9.8 Wavelets

<code>wavelets.ricker(t[, f0])</code>	Ricker wavelet
<code>wavelets.gaussian(t[, std])</code>	Gaussian wavelet

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`

- *1D, 2D and 3D Sliding*
- *2D Patching*
- *AVO modelling*
- *Chirp Radon Transform*
- *Convolution*
- *MP, OMP, ISTA and FISTA*
- *Multi-Dimensional Convolution*
- *Normal Moveout (NMO) Correction*
- *PhaseShift operator*
- *Pre-stack modelling*
- *Shift*
- *Slope estimation via Structure Tensor algorithm*
- *Spread How-to*
- *Synthetic seismic*
- *Wavelet estimation*
- *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

3.9.9 Geophysical Reservoir characterization

<code>avo.zoeppritz_scattering(vp1, vs1, rho1, ...)</code>	Zoeppritz solution.
<code>avo.zoeppritz_element(vp1, vs1, rho1, vp0, ...)</code>	Single element of Zoeppritz solution.
<code>avo.zoeppritz_pp(vp1, vs1, rho1, vp0, vs0, ...)</code>	PP reflection coefficient from the Zoeppritz scattering matrix.
<code>avo.approx_zoeppritz_pp(vp1, vs1, rho1, vp0, ...)</code>	PP reflection coefficient from the approximate Zoeppritz equation.
<code>avo.akirichards(theta, vsvp[, n])</code>	Three terms Aki-Richards approximation.
<code>avo.fatti(theta, vsvp[, n])</code>	Three terms Fatti approximation.
<code>avo.ps(theta, vsvp[, n])</code>	PS reflection coefficient

pylops.avo.avo.zoeppritz_scattering

`pylops.avo.avo.zoeppritz_scattering(vp1, vs1, rho1, vp0, vs0, rho0, theta1)`
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

zoep [`np.ndarray`] 4×4 matrix representing the scattering matrix for the incident angle `theta1`

See also:

zoeppritz_element Single reflectivity element of Zoeppritz solution
zoeppritz_pp PP reflectivity element of Zoeppritz solution

pylops.avo.avo.zoeppritz_element

`pylops.avo.avo.zoeppritz_element(vp1, vs1, rho1, vp0, vs0, rho0, theta1, element='PdPu')`
Single element of Zoeppritz solution.

Simple wrapper to `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

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
element [`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`)

Returns

refl [`np.ndarray`] reflectivity values for all input angles for specific combination of incident and reflected wave.

See also:

[`zoeppritz_scattering`](#) Zoeppritz solution

[`zoeppritz_pp`](#) PP reflectivity element of Zoeppritz solution

pylops.avo.avo.zoeppritz_pp

`pylops.avo.avo.zoeppritz_pp` (`vp1`, `vs1`, `rho1`, `vp0`, `vs0`, `rho0`, `theta1`)
PP reflection coefficient from the Zoeppritz scattering matrix.

Simple wrapper to `pylops.avo.avo.scattering_matrix`, returning the PP reflection coefficient from the Zoeppritz scattering matrix 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

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`] V_S/V_P ratio

n [`int`, optional] Number of samples (if *vsvp* is a scalar)

Returns

G1 [`np.ndarray`] First coefficient of three terms Aki-Richards approximation [$n_\theta \times n_{vsvp}$]

G2 [`np.ndarray`] Second coefficient of three terms Aki-Richards approximation [$n_\theta \times n_{vsvp}$]

G3 [`np.ndarray`] Third coefficient of three terms Aki-Richards approximation [$n_\theta \times n_{vsvp}$]

Notes

The three terms Aki-Richards approximation [1], [2], is used to compute the reflection coefficient as linear combination of contrasts in V_P , V_S , and ρ . More specifically:

$$R(\theta) = G_1(\theta) \frac{\Delta V_P}{\bar{V}_P} + G_2(\theta) \frac{\Delta V_S}{\bar{V}_S} + G_3(\theta) \frac{\Delta \rho}{\bar{\rho}}$$

where

$$G_1(\theta) = \frac{1}{2 \cos^2 \theta}, \quad (3.7)$$

$$G_2(\theta) = -4(V_S/V_P)^2 \sin^2 \theta, \quad (3.8)$$

$$G_3(\theta) = 0.5 - 2(V_S/V_P)^2 \sin^2 \theta, \quad (3.9)$$

$$\frac{\Delta V_P}{\bar{V}_P} = 2 \frac{V_{P,2} - V_{P,1}}{V_{P,2} + V_{P,1}}, \quad (3.10)$$

$$\frac{\Delta V_S}{\bar{V}_S} = 2 \frac{V_{S,2} - V_{S,1}}{V_{S,2} + V_{S,1}}, \quad (3.11)$$

$$\frac{\Delta \rho}{\bar{\rho}} = 2 \frac{\rho_2 - \rho_1}{\rho_2 + \rho_1}, \quad (3.12)$$

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] V_S/V_P ratio

n [int, optional] Number of samples (if vsvp is a scalar)

Returns

G1 [np.ndarray] First coefficient of three terms Smith-Gidlow approximation [$n_\theta \times n_{\text{vsvp}}$]

G2 [np.ndarray] Second coefficient of three terms Smith-Gidlow approximation [$n_\theta \times n_{\text{vsvp}}$]

G3 [np.ndarray] Third coefficient of three terms Smith-Gidlow approximation [$n_\theta \times n_{\text{vsvp}}$]

Notes

The three terms Fatti approximation [1], [2], is used to compute the reflection coefficient as linear combination of contrasts in AI, SI, and ρ . More specifically:

$$R(\theta) = G_1(\theta) \frac{\Delta \text{AI}}{\bar{\text{AI}}} + G_2(\theta) \frac{\Delta \text{SI}}{\bar{\text{SI}}} + G_3(\theta) \frac{\Delta \rho}{\bar{\rho}}$$

where

$$\begin{aligned}
 G_1(\theta) &= 0.5(1 + \tan^2 \theta), \\
 G_2(\theta) &= -4(V_S/V_P)^2 \sin^2 \theta, \\
 G_3(\theta) &= 0.5(4(V_S/V_P)^2 \sin^2 \theta - \tan^2 \theta), \\
 \frac{\Delta AI}{AI} &= 2 \frac{AI_2 - AI_1}{AI_2 + AI_1}, \\
 \frac{\Delta SI}{SI} &= 2 \frac{SI_2 - SI_1}{SI_2 + SI_1}, \\
 \frac{\Delta \rho}{\bar{\rho}} &= 2 \frac{\rho_2 - \rho_1}{\rho_2 + \rho_1}
 \end{aligned} \tag{3.13}$$

pylops.avo.avo.ps

`pylops.avo.avo.ps(theta, vsvp, n=1)`

PS reflection coefficient

Computes the coefficients for the PS 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] V_S/V_P ratio

n [int, optional] Number of samples (if vsvp is a scalar)

Returns

G1 [np.ndarray] First coefficient for VP [$n_\theta \times n_{vsvp}$]. Since the PS reflection at zero angle is zero, this value is not used and is only available to ensure function signature compatibility with other linearization routines.

G2 [np.ndarray] Second coefficient for VS [$n_\theta \times n_{vsvp}$]

G3 [np.ndarray] Third coefficient for density [$n_\theta \times n_{vsvp}$]

Notes

The approximation in [1] is used to compute the PS reflection coefficient as linear combination of contrasts in V_P , V_S , and ρ . More specifically:

$$R(\theta) = G_2(\theta) \frac{\Delta V_S}{V_S} + G_3(\theta) \frac{\Delta \rho}{\bar{\rho}}$$

where

$$\begin{aligned}
 G_2(\theta) &= \tan \frac{\theta}{2} \{4(V_S/V_P)^2 \sin^2 \theta - 4(V_S/V_P) \cos \theta \cos \phi\}, \\
 G_3(\theta) &= -\tan \frac{\theta}{2} \{1 - 2(V_S/V_P)^2 \sin^2 \theta + 2(V_S/V_P) \cos \theta \cos \phi\}, \\
 \frac{\Delta V_S}{V_S} &= 2 \frac{V_{S,2} - V_{S,1}}{V_{S,2} + V_{S,1}}, \\
 \frac{\Delta \rho}{\bar{\rho}} &= 2 \frac{\rho_2 - \rho_1}{\rho_2 + \rho_1}
 \end{aligned} \tag{3.19}$$

Note that θ is the P-incidence angle whilst ϕ is the S-reflected angle which is computed using Snell's law and the average V_S/V_P ratio.

3.10 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.

3.10.1 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:

```
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 `\` between the operator and the data (e.g., `A\y`).

After that we define our new object:

```
class Diagonal(LinearOperator):
```

followed by a `numpydoc docstring` (starting with `r"""` 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.

```
def __init__(self, d, dtype=None):
    self.d = d.ravel()
    self.shape = (len(self.d), len(self.d))
    self.dtype = np.dtype(dtype)
    self.explicit = False
```

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:


```
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 $\mathbf{x} = \mathbf{A}^H \mathbf{y}$. 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):

```
def _rmatvec(self, x):
    return self.d*x
```

And that's it, we have implemented our first linear operator!

3.10.2 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 \mathbf{u} and \mathbf{v} and verify the the following *equality* within a numerical tolerance:

$$(\mathbf{A} * \mathbf{u})^H * \mathbf{v} = \mathbf{u}^H * (\mathbf{A}^H * \mathbf{v})$$

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

```
@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.

```
"""Dot-test and inversion for diagonal operator
"""
d = np.arange(par['nx']) + 1.

Dop = Diagonal(d)
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)
```

3.10.3 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.

3.10.4 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).

3.11 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.

3.11.1 Types of Contributions

Report Bugs

Report bugs at <https://github.com/PyLops/pylops/issues>.

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/PyLops/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.

3.11.2 Getting Started to contribute

Ready to contribute?

1. Fork the *PyLops* repo.
2. Clone your fork locally:

```
>> 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):

```
>> git remote add upstream https://github.com/equinor/pylops
```

5. Create a branch for local development:

```
>> 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 `-u` when pushing the branch for the first time.

8. Submit a pull request through the GitHub website.

3.11.3 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.

3.12 Changelog

3.12.1 Version 1.18.1

Released on: 29/04/2022

- Refracted `pylops.utils.dottest`, and added two new optional input parameters (*atol* and *rtol*)
- Added optional parameter *densesolver* to `pylops.LinearOperator.div`

3.12.2 Version 1.18.0

Released on: 19/02/2022

- Added *NMO* example to gallery
- Extended `pylops.Laplacian` to N-dimensional arrays
- Added *forward* kind to `pylops.SecondDerivative` and `pylops.Laplacian`
- Added *chirp-sliding* kind to `pylops.waveeqprocessing.seismicinterpolation.SeismicInterpolation`
- Fixed bug due to the new internal structure of *LinearOperator* submodule introduced in *scipy1.8.0*

3.12.3 Version 1.17.0

Released on: 29/01/2022

- Added `pylops.utils.describe.describe` method
- Added fftengine to `pylops.waveeqprocessing.Marchenko`
- Added *ifftshift_before* and *fftshift_after* optional input parameters in `pylops.signalprocessing.FFT`
- Added *norm* optional input parameter to `pylops.signalprocessing.FFT2D` and `pylops.signalprocessing.FFTND`
- Added *scipy* backend to `pylops.signalprocessing.FFT` and `pylops.signalprocessing.FFT2D` and `pylops.signalprocessing.FFTND`
- Added *eps* optional input parameter in `pylops.utils.signalprocessing.slope_estimate`
- Added pre-commit hooks
- Improved pre-commit hooks
- Vectorized `pylops.utils.signalprocessing.slope_estimate`

- Handlexd `nfft<nt` case in `pylops.signalprocessing.FFT` and `pylops.signalprocessing.FFT2D` and `pylops.signalprocessing.FFTND`
- Introduced automatic casting of dtype in `pylops.MatrixMult`
- Improved documentation and definition of optimal parameters of `pylops.Spread`
- Major clean up of documentation and mathematical formulas
- Major refractoring of the inner structure of `pylops.signalprocessing.FFT` and `pylops.signalprocessing.FFT2D` and `pylops.signalprocessing.FFTND`
- Reduced warnings in test suite
- Reduced computational time of `test_wavedecomposition` in the test suite
- Fixed bug in `pylops.signalprocessing.Sliding1D`, `pylops.signalprocessing.Sliding2D` and `pylops.signalprocessing.Sliding3D` where the dtype of the Restriction operator is inferred from Op
- Fixed bug in `pylops.signalprocessing.Radon2D` and `pylops.signalprocessing.Radon3D` when using centered spatial axes
- Fixed scaling in `pylops.signalprocessing.FFT` with `real=True` to pass the dot-test

3.12.4 Version 1.16.0

Released on: 11/12/2021

- Added `pylops.utils.estimators` submodule for trace estimation
- Added `x0` in `pylops.optimization.sparsity.ISTA` and `pylops.optimization.sparsity.FISTA` to handle non-zero initial guess
- Modified `pylops.optimization.sparsity.ISTA` and `pylops.optimization.sparsity.FISTA` to handle multiple right hand sides
- Modified creation of `haxis` in `pylops.signalprocessing.Radon2D` and `pylops.signalprocessing.Radon3D` to allow for uncentered spatial axes
- Fixed `_rmatvec` for explicit in `pylops.LinearOperator._ColumnLinearOperator`

3.12.5 Version 1.15.0

Released on: 23/10/2021

- Added `pylops.signalprocessing.Shift` operator.
- Added option to choose derivative kind in `pylops.avo.poststack.PoststackInversion` and `pylops.avo.prestack.PrestackInversion`.
- Improved efficiency of adjoint of `pylops.signalprocessing.Fredholm1` by applying complex conjugation to the vectors.
- Added `vsvp` to `pylops.avo.prestack.PrestackInversion` allowing to use user defined VS/VP ratio.
- Added `kind` to `pylops.basicoperators.CausalIntegration` allowing full, half, or trapezoidal integration.
- Fixed `_hardthreshold_percentile` in `pylops.optimization.sparsity` - Issue #249.
- Fixed `r2norm` in `pylops.optimization.solver.cgls`.

3.12.6 Version 1.14.0

Released on: 09/07/2021

- Added `pylops.optimization.solver.lsqr` solver
- Added utility routine `pylops.utils.scalability_test` for scalability tests when using multiprocessing
- Added `pylops.avo.avo.ps` AVO modelling option and restructured `pylops.avo.prestack.PrestackLinearModelling` to allow passing any function handle that can perform AVO modelling apart from those directly available
- Added R-linear operators (when setting the property `clinear=False` of a linear operator). `pylops.basicoperators.Real`, `pylops.basicoperators.Imag`, and `pylops.basicoperators.Conj`
- Added possibility to run operators `pylops.basicoperators.HStack`, `pylops.basicoperators.VStack`, `pylops.basicoperators.Block`, `pylops.basicoperators.BlockDiag`, and `pylops.signalprocessing.Sliding3D` using multiprocessing
- Added dtype to vector `X` when using `scipy.sparse.linalg.lobpcg` in `eigs` method of `pylops.LinearOperator`
- Use `kind=forward` for FirstDerivative in `pylops.avo.poststack.PoststackInversion` inversion when dealing with L1 regularized inversion as it makes the inverse problem more stable (no ringing in solution)
- Changed `cost` in `pylops.optimization.solver.cg` and `pylops.optimization.solver.cgl` to be L2 norms of residuals
- Fixed `pylops.utils.dottest.dottest` for imaginary vectors and to ensure u and v vectors are of same dtype of the operator

3.12.7 Version 1.13.0

Released on: 26/03/2021

- Added `pylops.signalprocessing.Sliding1D` and `pylops.signalprocessing.Patch2D` operators
- Added `pylops.basicoperators.MemoizeOperator` operator
- Added decay and analysis option in `pylops.optimization.sparsity.ISTA` and `pylops.optimization.sparsity.FISTA` solvers
- Added `toreal` and `toimag` methods to `pylops.LinearOperator`
- Make `nr` and `nc` optional in `pylops.utils.dottest.dottest`
- Fixed complex check in `pylops.basicoperators.MatrixMult` when working with complex-valued cupy arrays
- Fixed bug in data reshaping in check in `pylops.avo.prestack.PrestackInversion`
- Fixed loading error when using old cupy and/or cusignal (see [Issue #201](#))

3.12.8 Version 1.12.0

Released on: 22/11/2020

- Modified all operators and solvers to work with cupy arrays

- Added eigs and solver submodules to `pylops.optimization`
- Added deps and backend submodules to `pylops.utils`
- Fixed bug in `pylops.signalprocessing.Convolve2D` and `pylops.signalprocessing.ConvolveND` when dealing with filters that have less dimensions than the input vector.

3.12.9 Version 1.11.1

Released on: 24/10/2020

- Fixed import of `pyfttw` when not available in `py:class:pylops.signalprocessing.ChirpRadon3D`

3.12.10 Version 1.11.0

Released on: 24/10/2020

- Added `pylops.signalprocessing.ChirpRadon2D` and `pylops.signalprocessing.ChirpRadon3D` operators.
- Fixed bug in the inferred dimensions for regularization data creation in `pylops.optimization.leastsquares.NormalEquationsInversion`, `pylops.optimization.leastsquares.RegularizedInversion`, and `pylops.optimization.sparsity.SplitBregman`.
- Changed dtype of `pylops.HStack` to allow automatic inference from dtypes of input operator.
- Modified dtype of `pylops.waveeqprocessing.Marchenko` operator to ensure that outputs of forward and adjoint are real arrays.
- Reverted to previous complex-friendly implementation of `pylops.optimization.sparsity._softthreshold` to avoid division by 0.

3.12.11 Version 1.10.0

Released on: 13/08/2020

- Added `tosparse` method to `pylops.LinearOperator`.
- Added `kind=linear` in `pylops.signalprocessing.Seislet` operator.
- Added `kind` to `pylops.FirstDerivative` operator to perform forward and backward (as well as centered) derivatives.
- Added `kind` to `pylops.optimization.sparsity.IRLS` solver to choose between data or model sparsity.
- Added possibility to use `scipy.sparse.linalg.lobpcg` in `pylops.LinearOperator.eigs` and `pylops.LinearOperator.cond`
- Added possibility to use `scipy.signal.oaconvolve` in `pylops.signalprocessing.Convolve1D`.
- Added `NRegs` to `pylops.optimization.leastsquares.NormalEquationsInversion` to allow providing regularization terms directly in the form of $H^T H$.

3.12.12 Version 1.9.1

Released on: 25/05/2020

- Changed internal behaviour of `pylops.sparsity.OMP` when `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.

3.12.13 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.sparsity.ISTA` 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 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 `onthe-fly=True` returning the same result as when `onthe-fly=False`

3.12.14 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.waveeqprocessing.UpDownComposition3Doperator`, and `pylops.waveeqprocessing.PhaseShift` operators
- Fix bug in `pylops.basicoperators.Kronecker` (see [Issue #125](#))

3.12.15 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

3.12.16 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

3.12.17 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

3.12.18 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.sparsity.ISTA` 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

3.12.19 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

3.12.20 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.Flip` 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

3.12.21 Version 1.1.0

Released on: 13/12/2018

- Added `pylops.CausalIntegration` operator

3.12.22 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*.

3.12.23 Version 1.0.0

Released on: 04/12/2018

- First official release.

3.13 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.

3.13.1 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](#).

3.13.2 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](#).

3.13.3 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](#).

3.13.4 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 - [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 MDC operator - [Issue #32](#).
- NMO operator performing NMO modelling - [Issue #29](#).
- WavefieldDecomposition operator performing acoustic wavefield separation by inversion - [Issue #30](#).

3.14 Papers

This section lists various conference abstracts and papers using the PyLops framework:

- Vakalis, S.Chen, D., Yan, M., and Nanzer, J. A., *Image enhancement in active incoherent millimeter-wave imaging*. Passive and Active Millimeter-Wave Imaging XXIV 11745 [link](#) (2021).
- Li, X., Becker, T., Ravasi, M., Robertsson, J., and van Manen D.J., *Closed-aperture unbounded acoustics experimentation using multidimensional deconvolution*. The Journal of the Acoustical Society of America 149 (3), 1813-1828 [link](#) (2021).
- Kuijpers, D., Vasconcelos, I., and Putzky, P., *Reconstructing missing seismic data using Deep Learning*. arXiv, [link](#) (2021).

- Ravasi, M., and Birnie, C., *A Joint Inversion-Segmentation approach to Assisted Seismic Interpretation*. arXiv, [link](#) - [code](#) (2021).
- Haindl, C., Ravasi, M., and Broggin, F., *Handling gaps in acquisition geometries — Improving Marchenko-based imaging using sparsity-promoting inversion and joint inversion of time-lapse data*. Geophysics, 86 (2), S143-S154 [link](#) - [code](#) (2021).
- Ulrich, I. E., Zunino, A., Boehm, C., and Fichtner, A., *Sparsifying regularizations for stochastic sample average minimization in ultrasound computed tomography*. Medical Imaging 2021: Ultrasonic Imaging and Tomography. International Society for Optics and Photonics, [link](#) (2021).
- R Feng, R., Mejer Hansen, T., Grana, D., and Balling, N., *An unsupervised deep-learning method for porosity estimation based on poststack seismic data*. Geophysics, 85 (6), M97-M105. [link](#) (2020).
- Nightingale J. W., Hayes, R.G, et al., *PyAutoLens: Open-Source Strong Gravitational Lensing*, JOSS, [link](#) - [code](#) (2020).
- Zhang, M., *Marchenko Green's functions from compressive sensing acquisition* SEG Technical Program Expanded Abstracts, [link](#) (2020).
- Vargas, D., and Vasconcelos I., *Rayleigh-Marchenko Redatuming Using Scattered Fields in Highly Complex Media*, EAGE Technical Program Expanded Abstracts, [link](#) (2020).
- Ravasi, M., and Vasconcelos I., *Implementation of Large-Scale Integral Operators with Modern HPC Solutions*, EAGE Technical Program Expanded Abstracts, [link](#) - [code](#) (2020).
- Ruan, J., and Vasconcelos I., *Data-and prior-driven sampling and wavefield reconstruction for sparse, irregularly-sampled, higher-order gradient data*, SEG Technical Program Expanded Abstracts, [link](#) - [code](#) (2019).

3.15 Citing

When using `pylops` in scientific publications, please cite the following paper:

- Ravasi, M., and Vasconcelos I., *PyLops—A linear-operator Python library for scalable algebra and optimization*, Software X, (2020). [link](#).

3.16 Contributors

- Matteo Ravasi, [mrava87](#)
- Carlos da Costa, [cako](#)
- Dieter Werthmüller, [prisae](#)
- Tristan van Leeuwen, [TristanvanLeeuwen](#)
- Leonardo Uieda, [leouieda](#)
- Filippo Broggin, [filippo82](#)
- Tyler Hughes, [twhughes](#)
- Lyubov Skopintseva, [lskopintseva](#)
- Francesco Picetti, [fpicetti](#)
- Alan Richardson, [ar4](#)
- BurningKarl, [BurningKarl](#)

- [Nick Luiken](#), NickLuiken

Bibliography

- [1] <http://www.caam.rice.edu/software/ARPACK/>
- [1] Fomel, S., Liu, Y., “Seislet transform and seislet frame”, *Geophysics*, 75, no. 3, V25-V38. 2010.
- [1] Andersson, F and Robertsson J. “Fast $\tau - p$ transforms by chirp modulation”, *Geophysics*, vol 84, NO.1, pp. A13-A17, 2019.
- [1] Andersson, F and Robertsson J. “Fast $\tau - p$ transforms by chirp modulation”, *Geophysics*, vol 84, NO.1, pp. A13-A17, 2019.
- [1] Wapenaar, K. “Reciprocity properties of one-way propagators”, *Geophysics*, vol. 63, pp. 1795-1798. 1998.
- [1] Wapenaar, K. “Reciprocity properties of one-way propagators”, *Geophysics*, vol. 63, pp. 1795-1798. 1998.
- [1] Wapenaar, K., van der Neut, J., Ruigrok, E., Draganov, D., Hunziker, J., Slob, E., Thorbecke, J., and Snieder, R., “Seismic interferometry by crosscorrelation and by multi-dimensional deconvolution: a systematic comparison”, *Geophysical Journal International*, vol. 185, pp. 1335-1364. 2011.
- [1] Paige, C. C., and Saunders, M. A. “LSQR: An algorithm for sparse linear equations and sparse least squares”, *ACM TOMS*, vol. 8, pp. 43-71, 1982.
- [1] https://en.wikipedia.org/wiki/Iteratively_reweighted_least_squares
- [2] Chartrand, R., and Yin, W. “Iteratively reweighted algorithms for compressive sensing”, *IEEE*. 2008.
- [1] Daubechies, I., Defrise, M., and De Mol, C., “An iterative thresholding algorithm for linear inverse problems with a sparsity constraint”, *Communications on pure and applied mathematics*, vol. 57, pp. 1413-1457. 2004.
- [1] Beck, A., and Teboulle, M., “A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems”, *SIAM Journal on Imaging Sciences*, vol. 2, pp. 183-202. 2009.
- [1] van den Berg E., Friedlander M.P., “Probing the Pareto frontier for basis pursuit solutions”, *SIAM J. on Scientific Computing*, vol. 31(2), pp. 890-912. 2008.
- [1] Goldstein T. and Osher S., “The Split Bregman Method for L1-Regularized Problems”, *SIAM J. on Scientific Computing*, vol. 2(2), pp. 323-343. 2008.
- [1] Amundsen, L., 1993, Wavenumber-based filtering of marine point-source data: *GEOPHYSICS*, 58, 1335–1348.
- [1] Wapenaar, K. “Reciprocity properties of one-way propagators”, *Geophysics*, vol. 63, pp. 1795-1798. 1998.

- [1] Wapenaar, K., van der Neut, J., Ruigrok, E., Draganov, D., Hunziker, J., Slob, E., Thorbecke, J., and Snieder, R., “Seismic interferometry by crosscorrelation and by multi-dimensional deconvolution: a systematic comparison”, *Geophysical Journal International*, vol. 185, pp. 1335-1364. 2011.
- [1] Wapenaar, K., Thorbecke, J., Van der Neut, J., Broggini, F., Slob, E., and Snieder, R., “Marchenko imaging”, *Geophysics*, vol. 79, pp. WA39-WA57. 2014.
- [2] van der Neut, J., Vasconcelos, I., and Wapenaar, K. “On Green’s function retrieval by iterative substitution of the coupled Marchenko equations”, *Geophysical Journal International*, vol. 203, pp. 792-813. 2015.
- [1] Hutchinson, M. F. (1990). *A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines*. *Communications in Statistics - Simulation and Computation*, 19(2), 433–450.
- [2] Avron, H., and Toledo, S. (2011). *Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix*. *Journal of the ACM*, 58(2), 1–34.
- [1] Meyer, R. A., Musco, C., Musco, C., & Woodruff, D. P. (2021). *Hutch++: Optimal Stochastic Trace Estimation*. In *Symposium on Simplicity in Algorithms (SOSA)* (pp. 142–155). Philadelphia, PA: Society for Industrial and Applied Mathematics. [link](#)
- [1] Meyer, R. A., Musco, C., Musco, C., & Woodruff, D. P. (2021). *Hutch++: Optimal Stochastic Trace Estimation*. In *Symposium on Simplicity in Algorithms (SOSA)* (pp. 142–155). Philadelphia, PA: Society for Industrial and Applied Mathematics. [link](#)
- [1] Snieder, R. “A Guided Tour of Mathematical Methods for the Physical Sciences”, Cambridge University Press, pp. 302, 2004.
- [1] Van Vliet, L. J., Verbeek, P. W., “Estimators for orientation and anisotropy in digitized images”, *Journal ASCI Imaging Workshop*. 1995.
- [1] Dvorkin et al. *Seismic Reflections of Rock Properties*. Cambridge. 2014.
- [1] https://wiki.seg.org/wiki/AVO_equations
- [2] Aki, K., and Richards, P. G. (2002). *Quantitative Seismology* (2nd ed.). University Science Books.
- [1] https://www.subsurfwiki.org/wiki/Fatti_equation
- [2] Jan L. Fatti, George C. Smith, Peter J. Vail, Peter J. Strauss, and Philip R. Levitt, (1994), “Detection of gas in sandstone reservoirs using AVO analysis: A 3-D seismic case history using the Geostack technique,” *Geophysics* 59: 1362-1376.
- [1] Xu, Y., and Bancroft, J.C., “Joint AVO analysis of PP and PS seismic data”, *CREWES Report*, vol. 9. 1997.

A

akirichards() (in module *pylops.avo.avo*), 445
 apply() (*pylops.Regression* method), 327
 apply_columns() (*pylops.LinearOperator* method), 306
 apply_multiplepoints() (*pylops.waveeqprocessing.Marchenko* method), 421
 apply_onepoint() (*pylops.waveeqprocessing.Marchenko* method), 421
 approx_zoeppritz_pp() (in module *pylops.avo.avo*), 445
 AVOLinearModelling (class in *pylops.avo.avo*), 389

B

Bilinear (class in *pylops.signalprocessing*), 356
 Block() (in module *pylops*), 335
 BlockDiag (class in *pylops*), 336

C

CausalIntegration (class in *pylops*), 328
 cg() (in module *pylops.optimization.solver*), 394
 cgls() (in module *pylops.optimization.solver*), 395
 ChirpRadon2D (class in *pylops.signalprocessing*), 374
 ChirpRadon3D (class in *pylops.signalprocessing*), 375
 cond() (*pylops.LinearOperator* method), 305
 Conj (class in *pylops*), 340
 conj() (*pylops.LinearOperator* method), 306
 convmtx() (in module *pylops.utils.signalprocessing*), 438
 Convolve1D (class in *pylops.signalprocessing*), 350
 Convolve2D() (in module *pylops.signalprocessing*), 352
 ConvolveND (class in *pylops.signalprocessing*), 353

D

Deghosting() (in module *pylops.waveeqprocessing*), 415

Demigration() (in module *pylops.waveeqprocessing*), 388
 describe() (in module *pylops.utils.describe*), 428
 Diagonal (class in *pylops*), 315
 directwave() (in module *pylops.waveeqprocessing.marchenko*), 437
 div() (*pylops.LinearOperator* method), 303
 dot() (*pylops.LinearOperator* method), 303
 dottest() (in module *pylops.utils*), 427
 DWT (class in *pylops.signalprocessing*), 366
 DWT2D (class in *pylops.signalprocessing*), 367

E

eigs() (*pylops.LinearOperator* method), 304

F

fatti() (in module *pylops.avo.avo*), 446
 FFT() (in module *pylops.signalprocessing*), 357
 FFT2D() (in module *pylops.signalprocessing*), 360
 FFTND() (in module *pylops.signalprocessing*), 363
 FirstDerivative (class in *pylops*), 343
 FirstDirectionalDerivative() (in module *pylops*), 348
 FISTA() (in module *pylops.optimization.sparsity*), 407
 Flip (class in *pylops*), 318
 Fredholm1 (class in *pylops.signalprocessing*), 380
 FunctionOperator (class in *pylops*), 309

G

gaussian() (in module *pylops.utils.wavelets*), 442
 Gradient() (in module *pylops*), 347

H

HStack (class in *pylops*), 334
 hyperbolic2d() (in module *pylops.utils.seismic_events*), 434
 hyperbolic3d() (in module *pylops.utils.seismic_events*), 436

I

Identity (class in *pylops*), 313
 Imag (class in *pylops*), 339
 Interp () (in module *pylops.signalprocessing*), 355
 inv () (*pylops.MatrixMult* method), 312
 IRLS () (in module *pylops.optimization.sparsity*), 402
 ISTA () (in module *pylops.optimization.sparsity*), 405

K

Kronecker (class in *pylops*), 337

L

Laplacian () (in module *pylops*), 346
 linear2d () (in module *pylops.utils.seismicevents*), 433
 linear3d () (in module *pylops.utils.seismicevents*), 435
 LinearOperator (class in *pylops*), 302
 LinearRegression () (in module *pylops*), 327
 LSM (class in *pylops.waveeqprocessing*), 422
 lsqr () (in module *pylops.optimization.solver*), 396

M

makeaxis () (in module *pylops.utils.seismicevents*), 432
 Marchenko (class in *pylops.waveeqprocessing*), 419
 mask () (*pylops.Restriction* method), 325
 matmat () (*pylops.LinearOperator* method), 303
 matrix () (*pylops.Diagonal* method), 316
 MatrixMult (class in *pylops*), 311
 matvec () (*pylops.LinearOperator* method), 303
 MDC () (in module *pylops.waveeqprocessing*), 386
 MDD () (in module *pylops.waveeqprocessing*), 417
 MemoizeOperator (class in *pylops*), 310

N

nonstationary_convmtx () (in module *pylops.utils.signalprocessing*), 438
 NormalEquationsInversion () (in module *pylops.optimization.leastsquares*), 398

O

OMP () (in module *pylops.optimization.sparsity*), 404

P

Pad (class in *pylops*), 320
 parabolic2d () (in module *pylops.utils.seismicevents*), 433
 Patch2D () (in module *pylops.signalprocessing*), 379
 PhaseShift () (in module *pylops.waveeqprocessing*), 387
 PoststackInversion () (in module *pylops.avo.poststack*), 424

PoststackLinearModelling () (in module *pylops.avo.poststack*), 391
 PreconditionedInversion () (in module *pylops.optimization.leastsquares*), 401
 PressureToVelocity () (in module *pylops.waveeqprocessing*), 382
 PrestackInversion () (in module *pylops.avo.prestack*), 425
 PrestackLinearModelling () (in module *pylops.avo.prestack*), 392
 PrestackWaveletModelling () (in module *pylops.avo.prestack*), 393
 ps () (in module *pylops.avo.avo*), 447

R

Radon2D () (in module *pylops.signalprocessing*), 371
 Radon3D () (in module *pylops.signalprocessing*), 372
 Real (class in *pylops*), 338
 Regression (class in *pylops*), 325
 RegularizedInversion () (in module *pylops.optimization.leastsquares*), 399
 Restriction (class in *pylops*), 323
 ricker () (in module *pylops.utils.wavelets*), 441
 rmatmat () (*pylops.LinearOperator* method), 303
 rmatvec () (*pylops.LinearOperator* method), 303
 Roll (class in *pylops*), 319

S

scalability_test () (in module *pylops.utils*), 431
 SecondDerivative (class in *pylops*), 345
 SecondDirectionalDerivative () (in module *pylops*), 349
 Seislet (class in *pylops.signalprocessing*), 369
 SeismicInterpolation () (in module *pylops.waveeqprocessing*), 412
 Shift () (in module *pylops.signalprocessing*), 365
 Sliding1D () (in module *pylops.signalprocessing*), 376
 Sliding2D () (in module *pylops.signalprocessing*), 377
 Sliding3D () (in module *pylops.signalprocessing*), 378
 slope_estimate () (in module *pylops.utils.signalprocessing*), 438
 Smoothing1D () (in module *pylops*), 342
 Smoothing2D () (in module *pylops*), 343
 solve () (*pylops.waveeqprocessing.LSM* method), 423
 SPGL1 () (in module *pylops.optimization.sparsity*), 408
 SplitBregman () (in module *pylops.optimization.sparsity*), 410
 Spread (class in *pylops*), 330
 Sum (class in *pylops*), 321
 Symmetrize (class in *pylops*), 322

T

`taper2d()` (in module `pylops.utils.tapers`), 440
`taper3d()` (in module `pylops.utils.tapers`), 441
`todense()` (`pylops.Diagonal` method), 316
`todense()` (`pylops.LinearOperator` method), 304
`toimag()` (`pylops.LinearOperator` method), 306
`toreal()` (`pylops.LinearOperator` method), 306
`tosparse()` (`pylops.LinearOperator` method), 304
`trace()` (`pylops.LinearOperator` method), 306
`trace_hutchinson()` (in module `pylops.utils`), 428
`trace_hutchpp()` (in module `pylops.utils`), 429
`trace_nahutchpp()` (in module `pylops.utils`), 430
`Transpose` (class in `pylops`), 317

U

`UpDownComposition2D()` (in module `pylops.waveeqprocessing`), 383
`UpDownComposition3D()` (in module `pylops.waveeqprocessing`), 385

V

`VStack` (class in `pylops`), 332

W

`WavefieldDecomposition()` (in module `pylops.waveeqprocessing`), 416

Z

`Zero` (class in `pylops`), 314
`zoeppritz_element()` (in module `pylops.avo.avo`), 443
`zoeppritz_pp()` (in module `pylops.avo.avo`), 444
`zoeppritz_scattering()` (in module `pylops.avo.avo`), 443