PyLops-distributed

Nov 15, 2020

Getting started:

Note: This library is under early development.

Expect things to constantly change until version v1.0.0.

This library is an extension of PyLops for distributed operators.

As much as numpy and scipy lie at the core of the parent project PyLops, PyLops-distributed heavily builds on top of Dask, and more specifically Dask arrays.

Doing so, linear operators can be parallelized across several processes on a single node or across multiple nodes. Their forward and adjoint are first lazily built as directed acyclic graphs and evaluated only when requested by the user (or automatically within one of our solvers).

Most of the operators and solvers in PyLops-distributed mirror their equivalents in PyLops and users can seamlessly switch between PyLops and PyLops-distributed or even combine operators acting locally with distributed operators.

Here is a simple example showing how a diagonal operator can be created, applied and inverted using PyLops:

```
import numpy as np
from pylops import Diagonal

n = 10
x = np.ones(n)
d = np.arange(n) + 1

Dop = Diagonal(d)

# y = Dx
y = Dop*x
# x = D'y
xadj = Dop.H*y
# xinv = D^-1 y
xinv = Dop / y
```

and similarly using PyLops-distributed:

```
import numpy as np
import dask.array as da
import pylops_distributed
from pylops_distributed import Diagonal

# set-up client
client = pylops_distributed.utils.backend.dask()

n = 10
x = da.ones(n, chunks=(n//2,))
d = da.from_array(np.arange(n) + 1, chunks=(n//2, n//2))

Dop = Diagonal(d)

# y = Dx
y = Dop*x
# x = D'y
xadj = Dop.H*y
# xinv = D^-1 y
xinv = Dop / y
```

(continues on next page)

(continued from previous page)

```
da.compute((y, xadj, xinv))
client.close()
```

It is worth noticing two things at this point:

- In this specific case we did not even need to reimplement the `Diagonal` operator. Calling numpy operations as methods (e.g., `x.sum()`) instead of functions (e.g., `np.sum(x)`) makes it automatic for our operator to act as a distributed operator when a dask array is provided instead. Unfortunately not all numpy functions are also implemented as methods: in those cases we reimplement the o perator directly within PyLops-distributed.
- Using `*` and `.H*` is still possible also within PyLops-distributed, however when initializing an operator we will need to decide whether we want to simply create dask graph or also evaluation. This gives flexibility as we can decide if and when apply evaluation using the `compute` method on a dask array of choice.

CHAPTER 1

History

PyLops-Distributed was initially written and it is currently maintained by Equinor It is an extension of PyLops for large-scale optimization with *distributed* linear operators that can be tailored to our needs, and as contribution to the free software community.

1.1 Installation

You will need **Python 3.5 or greater** to get started.

1.1.1 Dependencies

Our mandatory dependencies are limited to:

- numpy
- scipy
- numba
- dask
- pylops

We advise using the Anaconda Python distribution to ensure that these dependencies are installed via the `Conda` package manager.

1.1.2 Step-by-step installation for users

Python environment

Stable releases on PyPI and Conda coming soon...

To install the latest source from github:

```
>> pip install https://git@github.com/equinor/pylops-distributed.git@master
```

or just clone the repository

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

1.1.3 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-distributed.git
```

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

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

If you prefer to build a new Conda enviroment 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 successfull.

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

```
>> source activate pylops-distributed
```

1.2 Tutorials

1.2.1 09. Multi-Dimensional Deconvolution

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

Data are first created as numpy arrays and then converted into Dask arrays. Data is chunked over frequencies to allow distributed computations of the Fredholm integral involved in the forward model.

NOTE: do not expect this code to run any fast than its pylops equivalent. for small datasets. The pylops-distributed framework should only be used when dealing with large datasets that do not fit in memory and benefit from distributed computing.

```
import warnings
import numpy as np
import dask.array as da
import matplotlib.pyplot as plt

from pylops.utils.tapers import taper3d
from pylops.utils.wavelets import ricker
from pylops.utils.seismicevents import makeaxis, hyperbolic2d
from pylops_distributed.waveeqprocessing import MDC, MDD

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

# sphinx_gallery_thumbnail_number = 3
```

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.]
amp_m = [1.]

t0_G = [0.2, 0.5, 0.7]
vrms_G = [800., 1200., 1500.]
amp_G = [1., 0.6, 0.5]

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

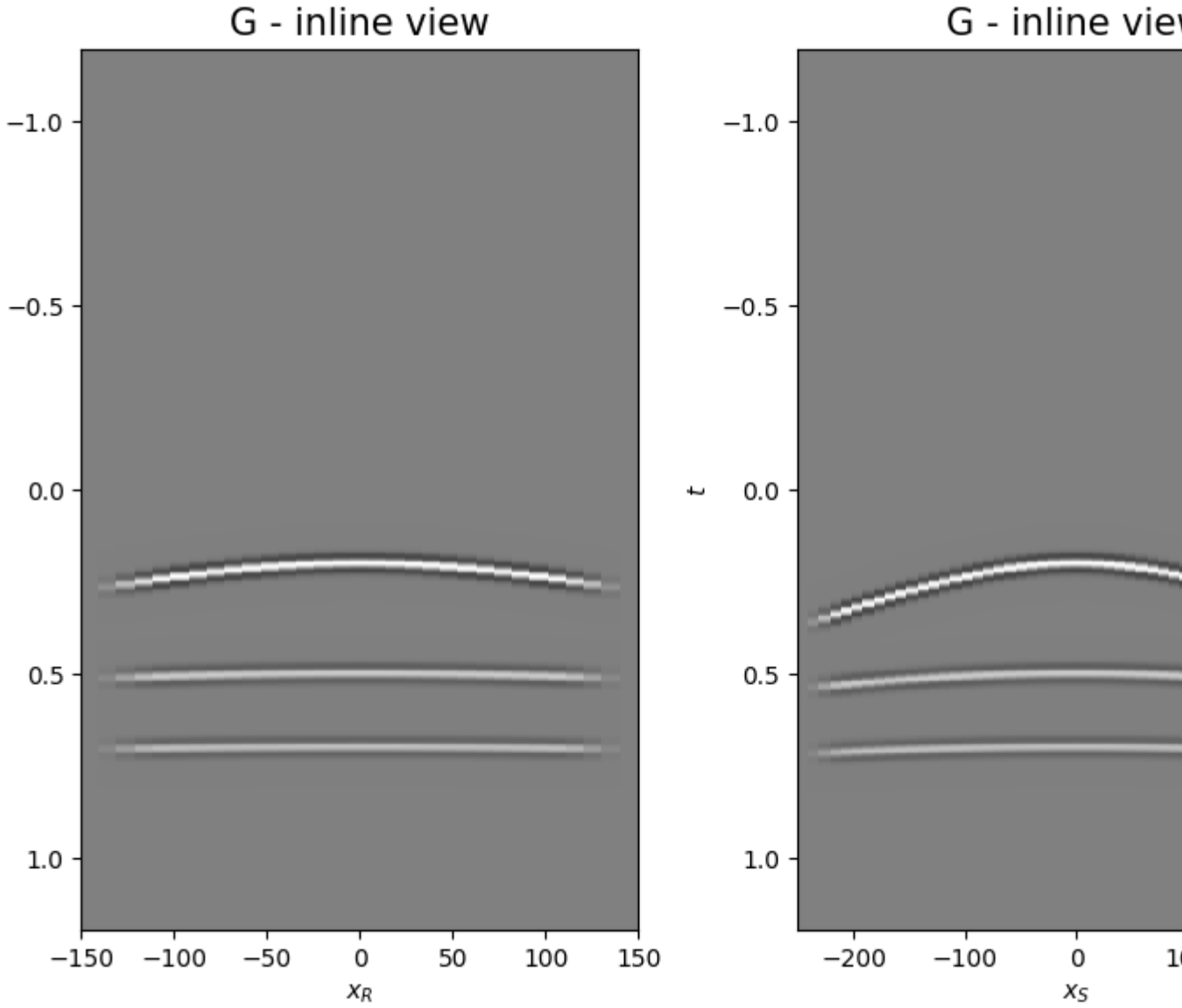
```
# Define MDC linear operator
Gwav_fft = np.fft.rfft(Gwav2, 2*par['nt']-1, axis=-1)
Gwav_fft = Gwav_fft[..., :par['nfmax']]
Gwav_fft = np.transpose(Gwav_fft, (2, 0, 1))

# Convert inputs to Dask and chunk frequency axis in 4 equal parts
m = da.from_array(m.T, chunks=(2*par['nt']-1, par['nx'])).ravel()
Gwav_fft = da.from_array(Gwav_fft, chunks=(par['nfmax'] // 4,
                                           par['ny'], par['nx']))
print(Gwav_fft)

# Define MDC linear operator
MDCop = MDC(Gwav_fft, nt=2 * par['nt']-1, nv=1)

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

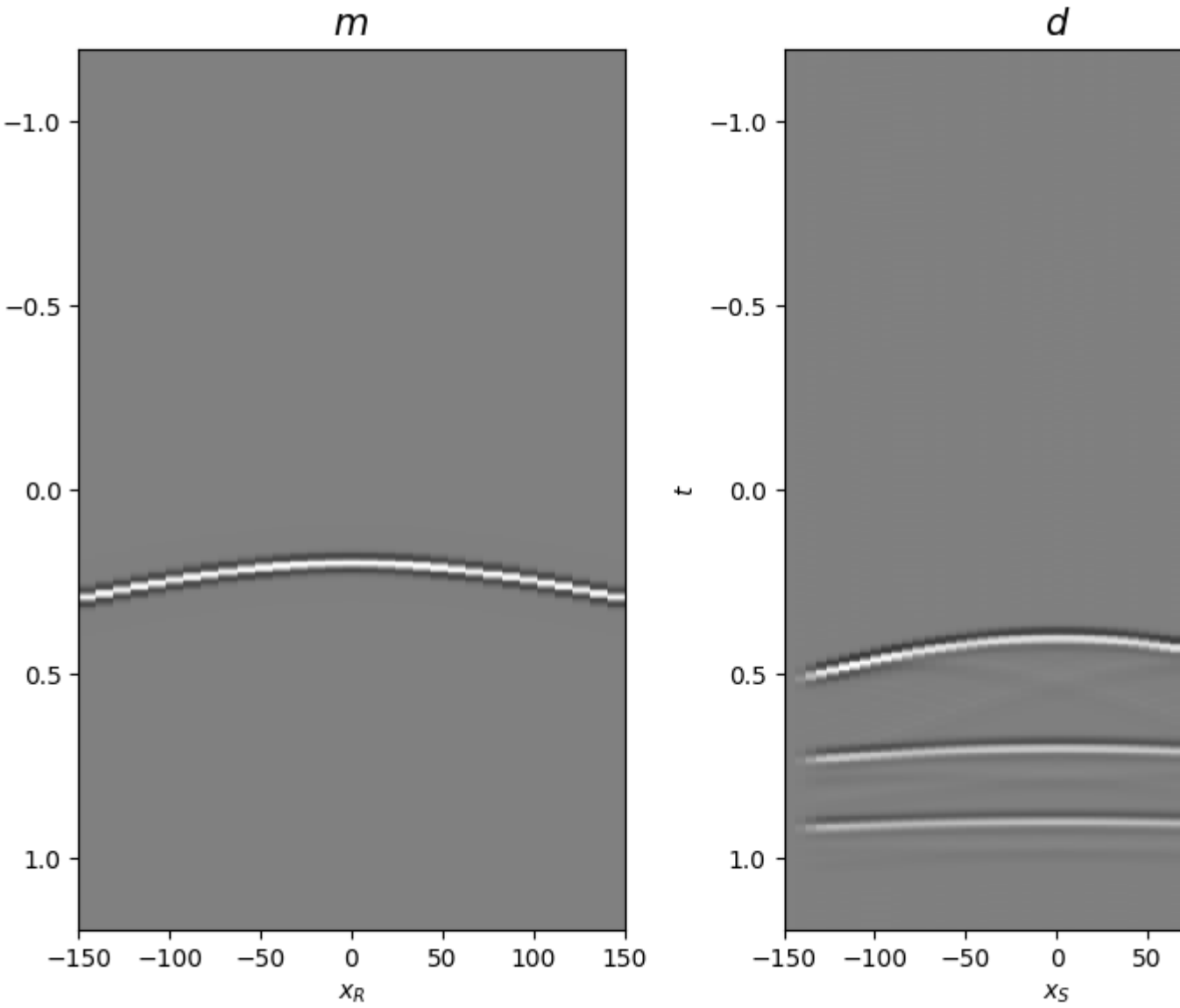
Out:

```
dask.array<array, shape=(200, 51, 31), dtype=complex128, chunksize=(50, 51, 31),␣
↪chunktype=numpy.ndarray>
```

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()), 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, 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()
```


G - inline view
G - inline vie
−1.0
−0.5
0.0
0.5
1.0
t
−150
−100
−50
0
50
100
150
x_R
−200
−100
0
x_S



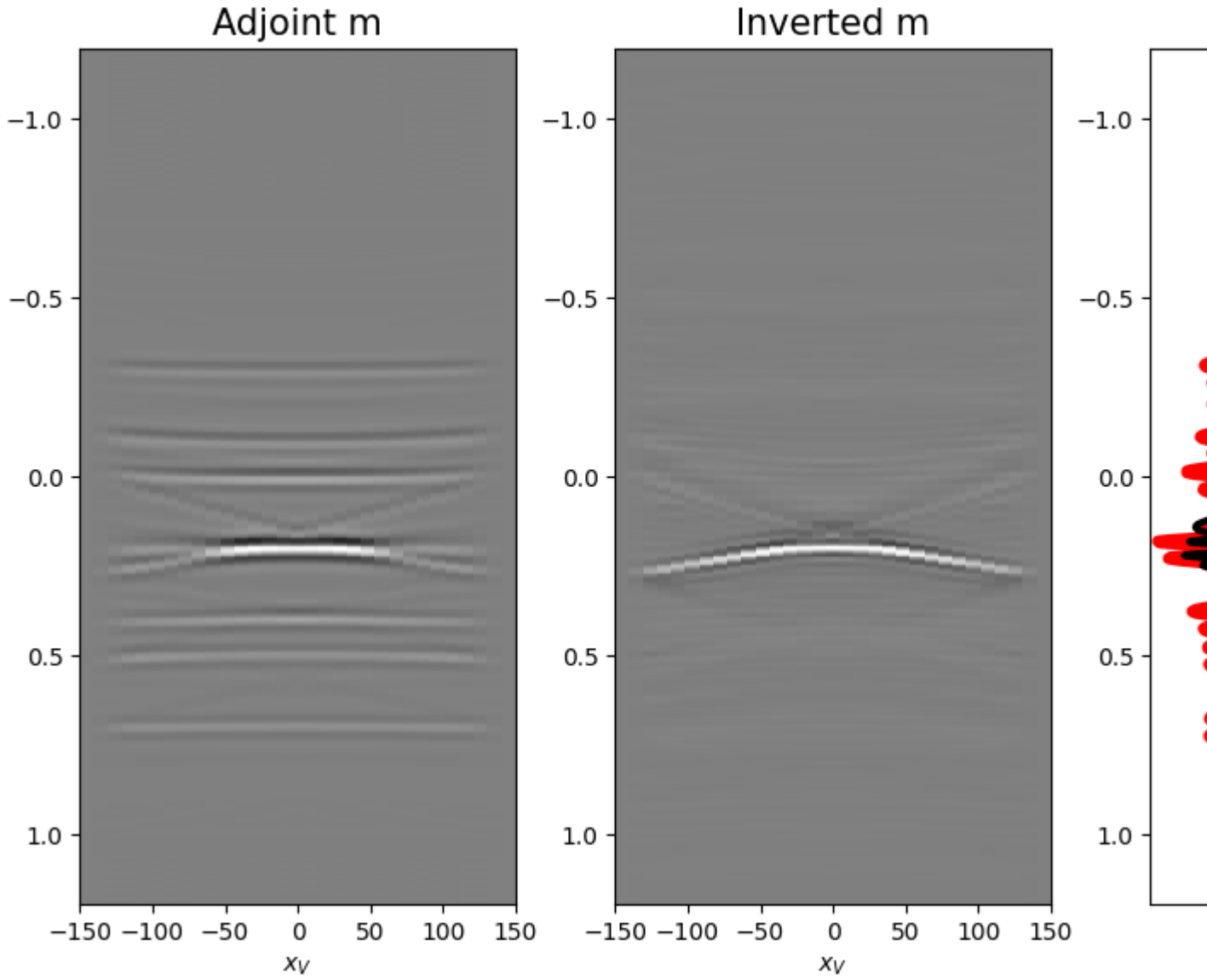
We are now ready to feed our operator to `pylops.waveeqprocessing.MDD` and invert back for our input model

```
minv, madj = MDD(Gwav_fft, d[par['nt'] - 1:],
                 dt=par['dt'], dr=par['dx'],
                 nfmax=par['nfmax'], wav=wav,
                 twosided=True, add_negative=True,
                 adjoint=True, dottest=False,
                 **dict(niter=10, compute=False))

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, 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()))
```

(continues on next page)

(continued from previous page)

```
ax1.set_title('Adjoint m', fontsize=15)
ax1.set_xlabel(r'$x_V$')
axs[1].set_ylabel(r'$t$')
ax2.imshow(minv, 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()
```



Total running time of the script: (0 minutes 2.963 seconds)

1.2.2 Marchenko redatuming by inversion

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

Data are first converted to frequency domain and stored in the high-performance format Zarr. This allows lazy loading using a Dask array and distributing over frequencies the computation of the various Fredholm integrals involved in the forward model.

NOTE: do not expect this code to run any fast than its pylops equivalent. for small datasets. The pylops-distributed framework should only be used when dealing with large datasets that do not fit in memory and benefit from distributed computing.

```
import warnings
import numpy as np
import dask.array as da
import matplotlib.pyplot as plt

from scipy.signal import convolve
from pylops_distributed.waveeqprocessing import Marchenko

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

# sphinx_gallery_thumbnail_number = 4
```

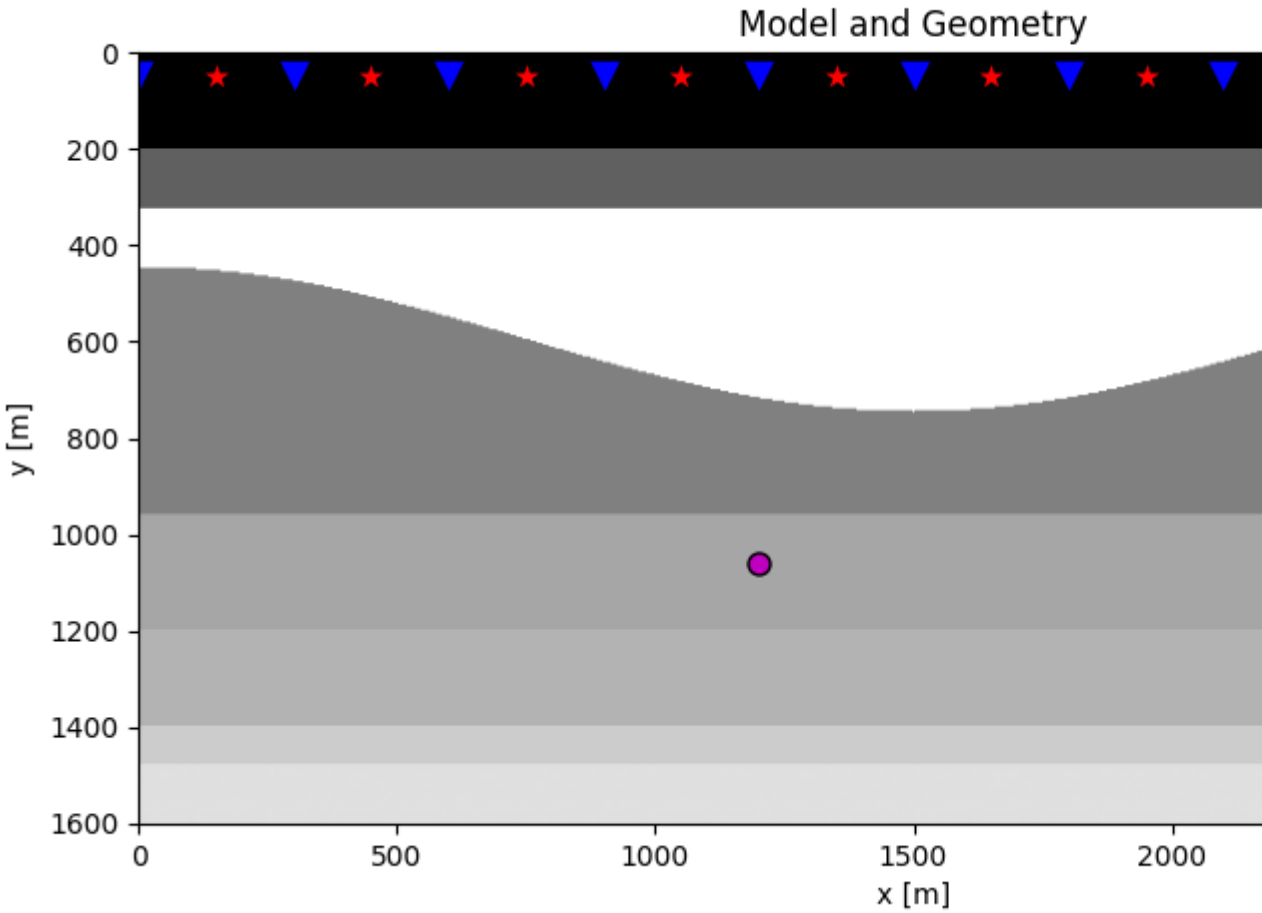
Let's start by defining some input parameters and loading the test data

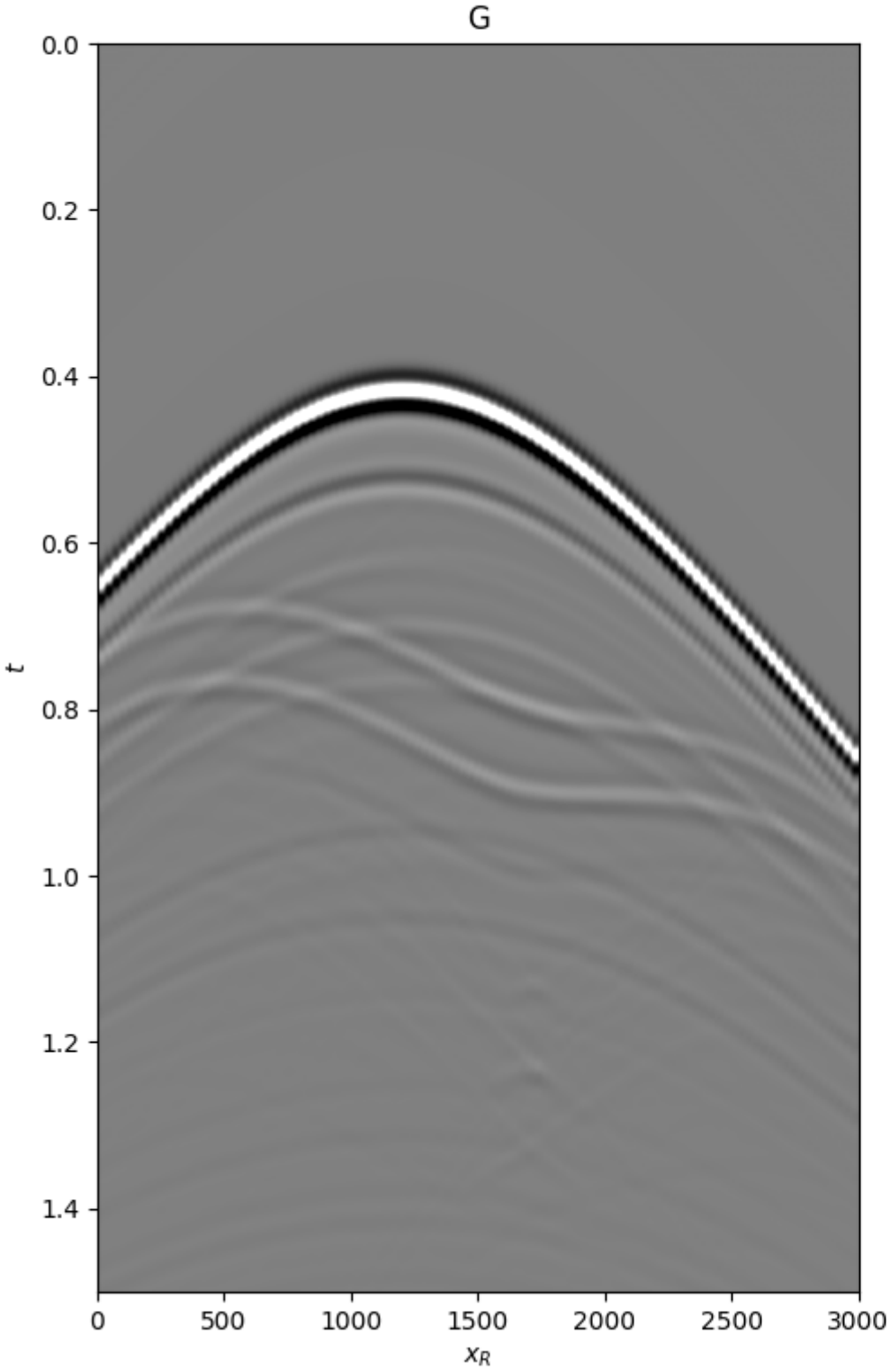
```
# Input parameters
inputfile = '../testdata/marchenko/input.npz'
inputzarr = '../testdata/marchenko/input.zarr'

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']
```

(continues on next page)

(continued from previous page)

```
z, x = inputdata['z'], inputdata['x']

# Reflection response in frequency domain (R[f, s, r])
R_fft = da.from_zarr(inputzarr)
print('R_fft:', R_fft)

# Subsurface fields
Gsub = inputdata['Gsub']
G0sub = inputdata['G0sub']
wav = inputdata['wav']
wav_c = np.argmax(wav)

t = inputdata['t']
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, 2, sharey=True, figsize=(12, 9))
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)
```


Model and Geometry
y [m]
x [m]
0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000


G
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
t
0
500
1000
1500
2000
2500
3000
x_R

Out:

```
R_fft: dask.array<from-zarr, shape=(500, 101, 101), dtype=complex64, chunksize=(125,␣
↪101, 101), chunktype=numpy.ndarray>

(1.5, 0.0)
```

Let's now create an object of the *pylops_distributed.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_fft, nt=nt, dt=dt, dr=dr, wav=wav,
                        toff=toff, nsmooth=nsmooth)

f1_inv_minus, f1_inv_plus, p0_minus, g_inv_minus, g_inv_plus = \
    MarchenkoWM.apply_onepoint(trav, G0=G0sub.T, rtm=True, greens=True,
                               dottest=False, **dict(niter=niter, compute=True))
g_inv_tot = g_inv_minus + g_inv_plus
```

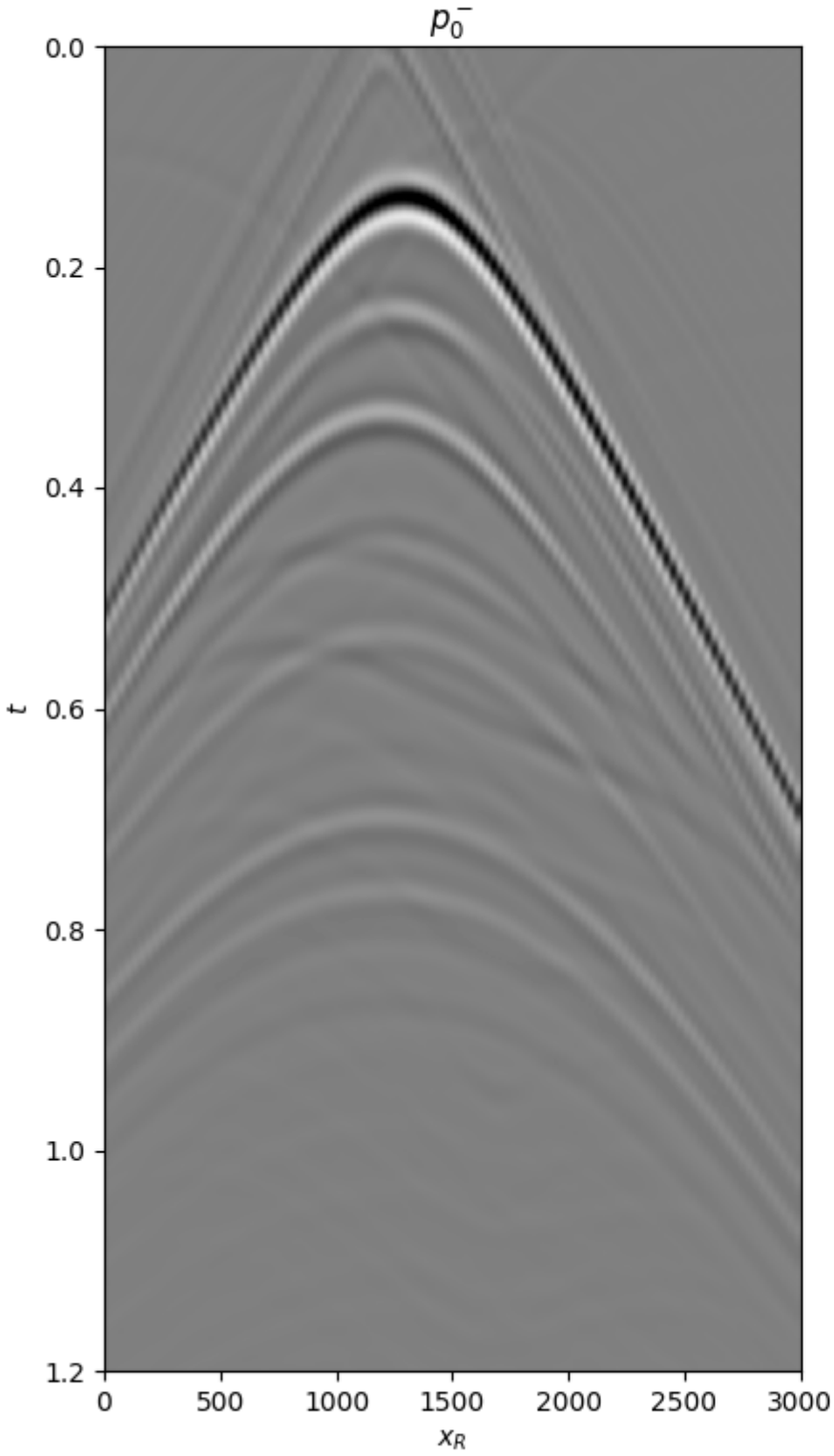
We can now compare the result of Marchenko redatuming via LSQR with standard redatuming

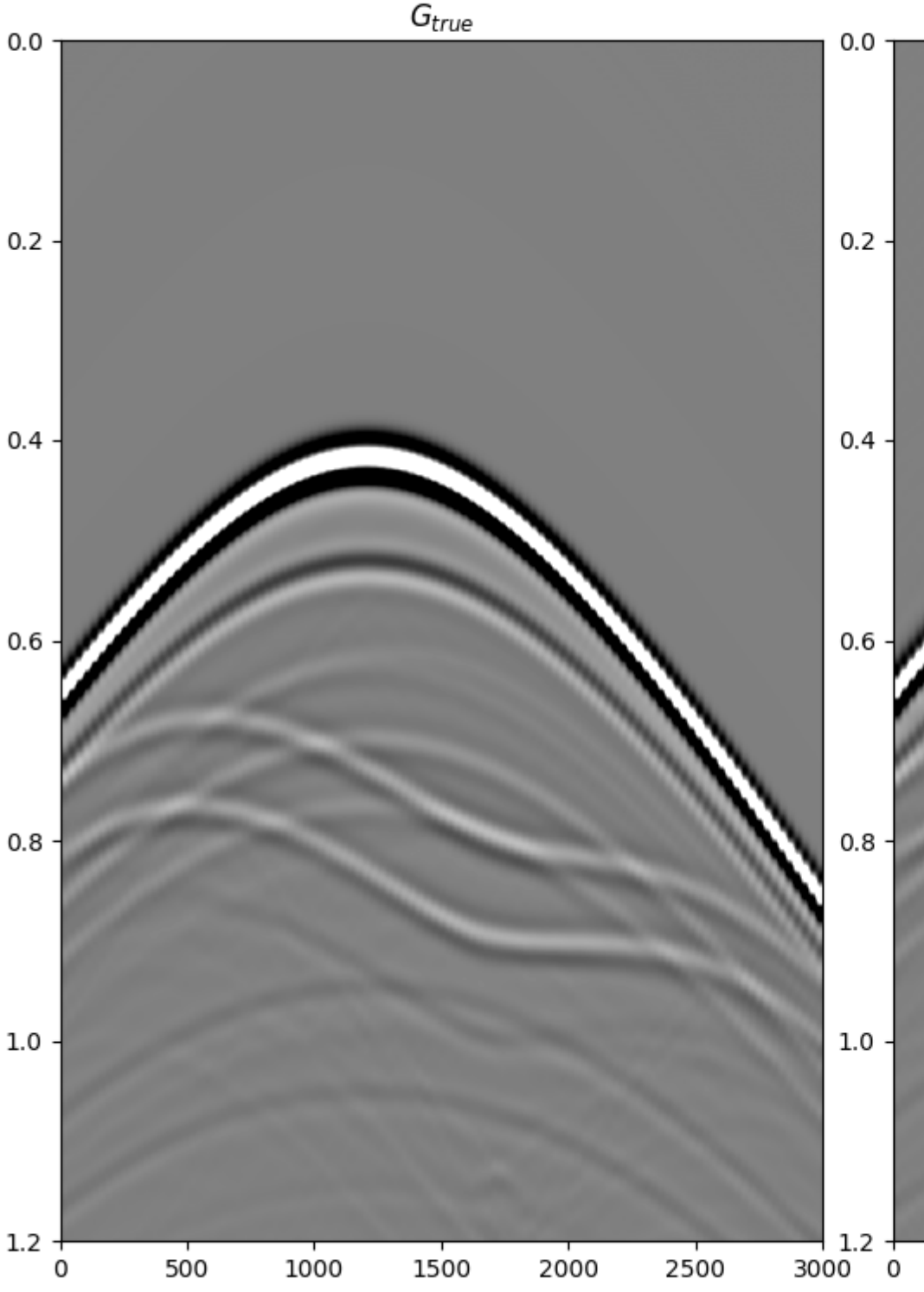
```
fig, axs = plt.subplots(1, 3, sharey=True, figsize=(16, 9))
axs[0].imshow(p0_minus.T, cmap='gray', vmin=-5e5, vmax=5e5,
              extent=(r[0, 0], r[0, -1], t[-1], -t[-1]))
axs[0].set_title(r'$p_0^-$')
axs[0].set_xlabel(r'$x_R$')
axs[0].set_ylabel(r'$t$')
axs[0].axis('tight')
axs[0].set_ylim(1.2, 0)
axs[1].imshow(g_inv_minus.T, cmap='gray', vmin=-5e5, vmax=5e5,
              extent=(r[0, 0], r[0, -1], t[-1], -t[-1]))
axs[1].set_title(r'$g^-$')
axs[1].set_xlabel(r'$x_R$')
axs[1].set_ylabel(r'$t$')
axs[1].axis('tight')
axs[1].set_ylim(1.2, 0)
axs[2].imshow(g_inv_plus.T, cmap='gray', vmin=-5e5, vmax=5e5,
              extent=(r[0, 0], r[0, -1], t[-1], -t[-1]))
axs[2].set_title(r'$g^+$')
axs[2].set_xlabel(r'$x_R$')
axs[2].set_ylabel(r'$t$')
axs[2].axis('tight')
axs[2].set_ylim(1.2, 0)

fig = plt.figure(figsize=(15, 9))
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,
```

(continues on next page)

(continued from previous page)

```
           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)
```


p_0^-
0.0
0.2
0.4
0.6
0.8
1.0
1.2
t
0
500
1000
1500
2000
2500
3000
x_R


G_{true}
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
500
1000
1500
2000
2500
3000

Out:

```
(1.2, 0.0)
```

Total running time of the script: (0 minutes 5.008 seconds)

1.3 PyLops-distributed API

1.3.1 Linear operators

Basic operators

MatrixMult(A[, dims, compute, todask, dtype])	Matrix multiplication.
Identity(N[, M, inplace, compute, todask, dtype])	Identity operator.
Diagonal(diag[, dims, dir, compute, todask, ...])	Diagonal operator.
Transpose(dims, axes[, compute, todask, dtype])	Transpose operator.
Roll(N[, dims, dir, shift, compute, todask, ...])	Roll along an axis.
Restriction(M, iava[, dims, dir, inplace, ...])	Restriction (or sampling) operator.
Spread(dims, dimsd[, table, dtable, ...])	Spread operator.
VStack(ops[, chunks, compute, todask, ...])	Vertical stacking.
HStack(ops[, chunks, compute, todask, dtype])	Horizontal stacking.
BlockDiag(ops[, chunks, compute, todask, dtype])	Block-diagonal operator.

pylops_distributed.MatrixMult

class pylops_distributed.**MatrixMult**(*A*, *dims=None*, *compute=(False, False)*, *todask=(False, False)*, *dtype='float64'*)

Matrix multiplication.

Simple wrapper to `dask.array.dot` for an input matrix **A**.

Parameters

A [`dask.array.ndarray`] 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).

compute [`tuple`, optional] Compute the outcome of forward and adjoint or simply define the graph and return a `dask.array`

todask [`tuple`, optional] Apply `dask.array.from_array` to model and data before applying forward and adjoint respectively

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

Notes

Refer to `pylops.basicoperators.MatrixMult` for implementation details.

Attributes

shape [`tuple`] Operator shape

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

Methods

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

inv()

Return the inverse of $\mathbf{A}$.

Returns

Ainv [numpy.ndarray] Inverse matrix.

pylops_distributed.Identity

class pylops_distributed.**Identity**(*N*, *M=None*, *inplace=True*, *compute=(False*, *False)*, *todask=(False*, *False)*, *dtype='float64'*)

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.

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).

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Raises

ValueError If M is different from N

Notes

Refer to `pylops.basicoperators.Identity` for implementation details.

Attributes

shape [`tuple`] Operator shape

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

Methods

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

pylops_distributed.Diagonal

class pylops_distributed.**Diagonal**(*diag*, *dims=None*, *dir=0*, *compute=(False, False)*, *todask=(False, False)*, *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 perfomed on the direction `dir`. Note that the vector `diag` will need to have size equal to `dims[dir]`.

Parameters

diag [`dask.array.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.

compute [`tuple`, optional] Compute the outcome of forward and adjoint or simply define the graph and return a `dask.array.array`

todask [`tuple`, optional] Apply `dask.array.from_array` to model and data before applying forward and adjoint respectively

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

Notes

Refer to `pylops.basicoperators.Diagonal` for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

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

Examples using `pylops_distributed.Diagonal`

- sphx_glr_gallery_plot_diagonal.py

pylops_distributed.Transpose

class pylops_distributed.**Transpose**(*dims*, *axes*, *compute=(False, False)*, *todask=(False, False)*, *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 (`None` if only one dimension is available)

axes [tuple, optional] Direction along which transposition is applied

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a `dask.array.array`

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Raises

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

Notes

Refer to pylops.basicoperators.Transpose for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

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

pylops_distributed.Roll

class pylops_distributed.**Roll**(*N*, *dims=None*, *dir=0*, *shift=1*, *compute=(False, False)*, *todask=(False, False)*, *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

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Raises

ValueError If M is different from N and chunks is not provided

Notes

Refer to pylops.basicoperators.Roll for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

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

pylops_distributed.Restriction

class pylops_distributed.**Restriction**(*M*, *iava*, *dims=None*, *dir=0*, *inplace=True*, *compute=(False, False)*, *todask=(False, False)*, *dtype='float64'*)

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.

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).

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Notes

Refer to pylops.basicoperators.Restriction for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

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

Examples using pylops_distributed.Restriction

- sphx_glr_gallery_plot_restriction.py

pylops_distributed.Spread

class pylops_distributed.**Spread**(*dims*, *dimsd*, *table=None*, *dtable=None*, *compute=(False, False)*, *todask=(False, False)*, *dtype='float64'*)

Spread operator.

Spread values from the input model vector arranged as a 2-dimensional array of size $[n_{x0} \times n_{t0}]$ into the data vector of size $[n_x \times n_t]$. Spreading is performed along parametric curves provided as look-up table of pre-computed indices (table) or computed on-the-fly using a function handle (fh).

In adjont 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 $[n_{x0} \times n_{t0}]$, where the first dimension is the spreading/stacking direction)

dimsd [tuple] Dimensions of model vector (vector will be reshaped internal into a two-dimensional array of size $[n_x \times n_t]$)

table [np.ndarray or dask.array.core.Array, optional] Look-up table of indeces of size $[n_x \times n_{x0} \times n_{t0}]$

dtable [np.ndarray or dask.array.core.Array, optional] Look-up table of decimals remainders for linear interpolation of same size as dtable

fh [np.ndarray, optional] Function handle that returns an index (and a fractional value in case of interp=True) to be used for spreading/stacking given indices in $x0$ and t axes (if None use look-up table table)

interp [bool, optional] Apply linear interpolation (True) or nearest interpolation (False) during stacking/spreading along parametric curve. To be used only if engine='numba', inferred directly from the number of outputs of fh for engine='numpy'

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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 $[n_{x0} \times n_t 0 \times n_x]$

Notes

Refer to pylops.basicoperators.Spread for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

__init__(dims, dimsd[, table, dtable, ...])	Initialize this LinearOperator.
adjoint()	Hermitian adjoint.
apply_columns(cols)	Apply subset of columns of operator
cond([uselobpcg])	Condition number of linear operator.
conj()	Complex conjugate operator
div(y[, niter])	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.

Continued on next page

Table 8 – continued from previous page

dot(x)	Matrix-vector multiplication.
eigs([neigs, symmetric, niter, uselobpcg])	Most significant eigenvalues of linear operator.
matmat(X)	Matrix-matrix multiplication.
matvec(x)	Matrix-vector multiplication.
rmatmat(X)	Adjoint matrix-matrix multiplication.
rmatvec(x)	Adjoint Matrix-vector multiplication.
todense()	Return dense matrix.
tosparse()	Return sparse matrix.
transpose()	Transpose this linear operator.

pylops_distributed.VStack

class pylops_distributed.**VStack**(*ops*, *chunks=None*, *compute=(False, False)*, *todask=(False, False)*, *usedelayed=False*, *dtype=None*)

Vertical stacking.

Stack a set of N linear operators vertically.

Parameters

ops [list] Linear operators to be stacked. Operators must be of pylops_distributed.LinearOperator type for usedelayed=False and pylops.LinearOperator for usedelayed=True

chunks [tuple, optional] Chunks for model and data (an array with a single chunk is created if chunks is not provided)

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

usedelayed [bool, optional] Use dask.delayed to parallelize over the N operators. Note that when this is enabled the input model and data should be passed as numpy.ndarray

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

Notes

Refer to pylops.basicoperators.VStack for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

__init__(ops[, chunks, compute, todask, ...])	Initialize this LinearOperator.
adjoint()	Hermitian adjoint.
apply_columns(cols)	Apply subset of columns of operator
cond([uselobpcg])	Condition number of linear operator.

Continued on next page

Table 9 – continued from previous page

conj()	Complex conjugate operator
div(y[, niter])	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
dot(x)	Matrix-vector multiplication.
eigs([neigs, symmetric, niter, uselobpcg])	Most significant eigenvalues of linear operator.
matmat(X)	Matrix-matrix multiplication.
matvec(x)	Matrix-vector multiplication.
rmatmat(X)	Adjoint matrix-matrix multiplication.
rmatvec(x)	Adjoint Matrix-vector multiplication.
todense()	Return dense matrix.
tosparse()	Return sparse matrix.
transpose()	Transpose this linear operator.

pylops_distributed.HStack

class pylops_distributed.**HStack**(*ops*, *chunks=None*, *compute=(False*, *False)*, *todask=(False*, *False)*, *dtype=None*)

Horizontal stacking.

Stack a set of N linear operators horizontally.

Parameters

ops [list] Linear operators to be stacked

chunks [tuple, optional] Chunks for model and data (an array with a single chunk is created if chunks is not provided)

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Notes

Refer to pylops.basicoperators.HStack for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

__init__(ops[, chunks, compute, todask, dtype])	Initialize this LinearOperator.
adjoint()	Hermitian adjoint.
apply_columns(cols)	Apply subset of columns of operator
cond([uselobpcg])	Condition number of linear operator.
conj()	Complex conjugate operator
div(y[, niter])	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
dot(x)	Matrix-vector multiplication.

Continued on next page

Table 10 – continued from previous page

eigs([neigs, symmetric, niter, uselobpcg])	Most significant eigenvalues of linear operator.
matmat(X)	Matrix-matrix multiplication.
matvec(x)	Matrix-vector multiplication.
rmatmat(X)	Adjoint matrix-matrix multiplication.
rmatvec(x)	Adjoint Matrix-vector multiplication.
todense()	Return dense matrix.
tosparse()	Return sparse matrix.
transpose()	Transpose this linear operator.

pylops_distributed.BlockDiag

class pylops_distributed.**BlockDiag**(*ops*, *chunks=None*, *compute=(False, False)*, *todask=(False, False)*, *dtype=None*)

Block-diagonal operator.

Create a block-diagonal operator from N linear operators.

Parameters

ops [list] Linear operators to be stacked

chunks [tuple, optional] Chunks for model and data (an array with a single chunk is created if chunks is not provided)

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Notes

Refer to pylops.basicoperators.VStack for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

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

Continued on next page

Table 11 – continued from previous page

rmatmat(X)	Adjoint matrix-matrix multiplication.
rmatvec(x)	Adjoint Matrix-vector multiplication.
todense()	Return dense matrix.
tosparse()	Return sparse matrix.
transpose()	Transpose this linear operator.

Smoothing and derivatives

Smoothing1D(nsmooth, dims[, dir, compute, ...])	1D Smoothing.
FirstDerivative(N[, dims, dir, sampling, ...])	First derivative.
SecondDerivative(N[, dims, dir, sampling, ...])	Second derivative.
Laplacian(dims[, dirs, weights, sampling, ...])	Laplacian.

pylops_distributed.Smoothing1D

pylops_distributed.**Smoothing1D**(*nsmooth*, *dims*, *dir=0*, *compute=(False*, *False)*, *chunks=(None*, *None)*, *todask=(False*, *False)*, *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] Lenght 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

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array.array

chunks [tuple, optional] Chunk size for model and data. If provided it will rechunk the model before applying the forward pass and the data before applying the adjoint pass

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Notes

Refer to pylops.basicoperators.Smoothing1D for implementation details.

Attributes

shape [tuple] Operator shape

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

Examples using pylops_distributed.Smoothing1D

- sphx_glr_gallery_plot_smoothing1d.py

pylops_distributed.FirstDerivative

class pylops_distributed.**FirstDerivative**(*N*, *dims=None*, *dir=0*, *sampling=1.0*, *compute=(False, False)*, *chunks=(None, None)*, *todask=(False, False)*, *dtype='float64'*)

First derivative.

Apply second-order centered first derivative.

Parameters

N [int] Number of samples in model.

dims [tuple, optional] Number of samples for each dimension (None if only one dimension is available)

dir [int, optional] Direction along which smoothing is applied.

sampling [float, optional] Sampling step dx.

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array.array

chunks [tuple, optional] Chunk size for model and data. If provided it will rechunk the model before applying the forward pass and the data before applying the adjoint pass

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Notes

Refer to pylops.basicoperators.FirstDerivative for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

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

Continued on next page

Table 13 – continued from previous page

transpose()	Transpose this linear operator.

Examples using pylops_distributed.FirstDerivative

- sphx_glr_gallery_plot_derivative.py

pylops_distributed.SecondDerivative

class pylops_distributed.**SecondDerivative**(*N*, *dims=None*, *dir=0*, *sampling=1.0*, *compute=(False*, *False)*, *chunks=(None*, *None)*, *todask=(False*, *False)*, *dtype='float64'*)

Second derivative.

Apply second-order centered second derivative.

Parameters

N [int] Number of samples in model.

dims [tuple, optional] Number of samples for each dimension (None if only one dimension is available)

dir [int, optional] Direction along which smoothing is applied.

sampling [float, optional] Sampling step dx.

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array.array

chunks [tuple, optional] Chunk size for model and data. If provided it will rechunk the model before applying the forward pass and the data before applying the adjoint pass

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Notes

Refer to pylops.basicoperators.SecondDerivative for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

__init__(N[, dims, dir, sampling, compute, ...])	Initialize this LinearOperator.
adjoint()	Hermitian adjoint.
apply_columns(cols)	Apply subset of columns of operator
cond([uselobpcg])	Condition number of linear operator.
conj()	Complex conjugate operator
div(y[, niter])	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.

Continued on next page

Table 14 – continued from previous page

dot(x)	Matrix-vector multiplication.
eigs([neigs, symmetric, niter, uselobpcg])	Most significant eigenvalues of linear operator.
matmat(X)	Matrix-matrix multiplication.
matvec(x)	Matrix-vector multiplication.
rmatmat(X)	Adjoint matrix-matrix multiplication.
rmatvec(x)	Adjoint Matrix-vector multiplication.
todense()	Return dense matrix.
tosparse()	Return sparse matrix.
transpose()	Transpose this linear operator.

Examples using `pylops_distributed.SecondDerivative`

- sphx_glr_gallery_plot_derivative.py

pylops_distributed.Laplacian

pylops_distributed.**Laplacian**(*dims*, *dirs=(0, 1)*, *weights=(1, 1)*, *sampling=(1, 1)*, *compute=(False, False)*, *chunks=(None, None)*, *todask=(False, False)*, *dtype='float64'*)

Laplacian.

Apply second-order centered Laplacian operator to a multi-dimensional array (at least 2 dimensions are required)

Parameters

dims [tuple] Number of samples for each dimension.

dirs [tuple, optional] Directions along which laplacian is applied.

weights [tuple, optional] Weight to apply to each direction (real laplacian operator if `weights=[1,1]`)

sampling [tuple, optional] Sampling steps for each direction

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a `dask.array.array`

chunks [tuple, optional] Chunk size for model and data. If provided it will rechunk the model before applying the forward pass and the data before applying the adjoint pass

todask [tuple, optional] Apply `dask.array.from_array` to model and data before applying forward and adjoint respectively

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

Returns

l2op [pylops.LinearOperator] Laplacian linear operator

Notes

Refer to `pylops.basicoperators.Laplacian` for implementation details.

Examples using `pylops_distributed.Laplacian`

- sphx_glr_gallery_plot_derivative.py

Signal processing

FFT(dims[, dir, nfft, sampling, real, ...])	One dimensional Fast-Fourier Transform.
Convolve1D(N, h[, offset, dims, dir, ...])	1D convolution operator.
Fredholm1(G[, nz, saveGt, compute, chunks, ...])	Fredholm integral of first kind.

pylops_distributed.signalprocessing.FFT

class pylops_distributed.signalprocessing.**FFT**(*dims*, *dir=0*, *nfft=None*, *sampling=1.0*, *real=False*, *fftshift=False*, *compute=(False, False)*, *chunks=(None, None)*, *todask=(None, None)*, *dtype='float64'*)

One dimensional Fast-Fourier Transform.

Apply Fast-Fourier Transform (FFT) along a specific direction `dir` of a multi-dimensional array of size `dim`.

Note that the FFT operator is an overload to the dask `dask.array.fft.fft` (or `dask.array.fft.rfft` for real models) in forward mode and to the dask `dask.array.fft.ifft` (or `dask.array.fft.irfft` for real models) in adjoint mode.

Scaling is properly taken into account to guarantee that the operator is passing the dot-test.

Note: For a real valued input signal, it is possible to store the values of the Fourier transform at positive frequencies only as values at negative frequencies are simply their complex conjugates. However as the operation of removing the negative part of the frequency axis in forward mode and adding the complex conjugates in adjoint mode is nonlinear, the Linear Operator FTT with `real=True` is not expected to pass the dot-test. It is thus *only* advised to use this flag when a forward and adjoint FFT is used in the same chained operator (e.g., `FFT.H*Op*FFT`) such as in `pylops_distributed.waveeqprocessing.mdd.MDC`.

Parameters

dims [`tuple`] Number of samples for each dimension

dir [`int`, optional] Direction along which FFT is applied.

nfft [`int`, optional] Number of samples in Fourier Transform (same as input if `nfft=None`)

sampling [`float`, optional] Sampling step `dt`.

real [`bool`, optional] Model to which fft is applied has real numbers (`True`) or not (`False`). Used to enforce that the output of adjoint of a real model is real.

fftshift [`bool`, optional] Apply fftshift/ifftshift (`True`) or not (`False`)

compute [`tuple`, optional] Compute the outcome of forward and adjoint or simply define the graph and return a `dask.array.array`

chunks [`tuple`, optional] Chunk size for model and data. If provided it will rechunk the model before applying the forward pass and the data before applying the adjoint pass

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Raises

ValueError If dims is not provided and if dir is bigger than len(dims)

Notes

Refer to pylops.signalprocessing.FFT for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

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

pylops_distributed.signalprocessing.Convolve1D

class pylops_distributed.signalprocessing.**Convolve1D**(*N*, *h*, *offset=0*, *dims=None*, *dir=0*, *compute=(False, False)*, *chunks=(None, None)*, *todask=(False, False)*, *dtype='float64'*)

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.

Note that if a multi-dimensional array is provided the array can also be chuncked along the direction dir where convolution is performed. In this case, dask handles the communication of borders between neighboring blocks.

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

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array.array

chunks [tuple, optional] Chunk size for model and data. If provided it will rechunk the model before applying the forward pass and the data before applying the adjoint pass

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

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

Raises

ValueError If offset is bigger than len(h) - 1

Notes

Refer to pylops.signalprocessing.Convolve1D for implementation details.

Attributes

shape [tuple] Operator shape

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

Methods

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

Examples using pylops_distributed.signalprocessing.Convolve1D

- sphx_glr_gallery_plot_smoothing1d.py
- sphx_glr_gallery_plot_convolve.py

pylops_distributed.signalprocessing.Fredholm1

class pylops_distributed.signalprocessing.**Fredholm1**(*G*, *nz=1*, *saveGt=True*, *compute=(False*, *False)*, *chunks=(None*, *None)*, *todask=(None*, *None)*, *dtype='float64'*)

Fredholm integral of first kind.

Implement a multi-dimensional Fredholm integral of first kind. Note that if the integral is two dimensional, this can be directly implemented using `pylops.basicoperators.MatrixMult`. A multi-dimensional Fredholm integral can be performed as a `pylops.basicoperators.BlockDiag` operator of a series of `pylops.basicoperators.MatrixMult`. However, here we take advantage of the structure of the kernel and perform it in a more efficient manner.

Parameters

G [`numpy.ndarray`] Multi-dimensional convolution kernel of size $[n_{slice} \times n_x \times n_y]$

nz [`numpy.ndarray`, optional] Additional dimension of model

saveGt [`bool`, optional] Save `G` and `G^H` to speed up the computation of adjoint (`True`) or create `G^H` on-the-fly (`False`) Note that `saveGt=True` will double the amount of required memory

compute [`tuple`, optional] Compute the outcome of forward and adjoint or simply define the graph and return a `dask.array`

chunks [`tuple`, optional] Chunk size for model and data. If provided it will rechunk the model before applying the forward pass and the data before applying the adjoint pass

todask [`tuple`, optional] Apply `dask.array.from_array` to model and data before applying forward and adjoint respectively

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

Notes

Refer to `pylops.signalprocessing.Identity` for implementation details.

Attributes

shape [`tuple`] Operator shape

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

Methods

`__init__`(G[, nz, saveGt, compute, chunks, ...])	Initialize this LinearOperator.
`adjoint`()	Hermitian adjoint.
`apply_columns`(cols)	Apply subset of columns of operator
`cond`([uselobpcg])	Condition number of linear operator.
`conj`()	Complex conjugate operator
`div`(y[, niter])	Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.
`dot`(x)	Matrix-vector multiplication.
`eigs`([neigs, symmetric, niter, uselobpcg])	Most significant eigenvalues of linear operator.

Continued on next page

Table 18 – continued from previous page

matmat(X)	Matrix-matrix multiplication.
matvec(x)	Matrix-vector multiplication.
rmatmat(X)	Adjoint matrix-matrix multiplication.
rmatvec(x)	Adjoint Matrix-vector multiplication.
todense()	Return dense matrix.
tosparse()	Return sparse matrix.
transpose()	Transpose this linear operator.

Wave-Equation processing

MDC(G, nt, nv[, dt, dr, twosided, saveGt, ...])	Multi-dimensional convolution.
MDD(G, d[, dt, dr, nfmax, wav, twosided, ...])	Multi-dimensional deconvolution.
Marchenko(R, nt[, dt, dr, wav, toff, ...])	Marchenko redatuming
Demigration(z, x, t, srcs, recs, vel, wav, ...)	Demigration operator.

pylops_distributed.waveeqprocessing.MDC

class pylops_distributed.waveeqprocessing.**MDC**(*G*, *nt*, *nv*, *dt=1.0*, *dr=1.0*, *twosided=True*, *saveGt=False*, *conj=False*, *prescaled=False*, *chunks=(None, None)*, *compute=(False, False)*, *todask=(False, False)*, *dtype=None*)

Multi-dimensional convolution.

Apply multi-dimensional convolution between two datasets. 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 $2 * n_t - 1$ for `twosided=True`), respectively.

Parameters

G [dask.array.ndarray] Multi-dimensional convolution kernel in frequency domain of size $[n_{fmax} \times n_s \times n_r]$

nt [int] Number of samples along time axis

nv [int] Number of samples along virtual source axis

dt [float, optional] Sampling of time integration axis

dr [float, optional] Sampling of receiver integration axis

twosided [bool, optional] MDC operator has both negative and positive time (True) or only positive (False)

saveGt [bool, optional] Save G and G^H to speed up the computation of adjoint of *pylops_distributed.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

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.

compute [tuple, optional] Compute the outcome of forward and adjoint or simply define the graph and return a dask.array

todask [tuple, optional] Apply dask.array.from_array to model and data before applying forward and adjoint respectively

dtype [str, optional] Type of elements in input array. If None, automatically inferred from G

Notes

Refer to pylops.waveeqprocessing.MDC for implementation details.

Methods

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

Examples using pylops_distributed.waveeqprocessing.MDC

- *09. Multi-Dimensional Deconvolution*

pylops_distributed.waveeqprocessing.MDD

pylops_distributed.waveeqprocessing.**MDD**(*G*, *d*, *dt=0.004*, *dr=1.0*, *nfmax=None*, *wav=None*, *twosided=True*, *adjoint=False*, *dottest=False*, *saveGt=False*, *add_negative=True*, ***kwargs_cgls*)

Multi-dimensional deconvolution.

Solve multi-dimensional deconvolution problem using scipy.sparse.linalg.lsqr iterative solver.

Parameters

G [dask.array.ndarray] Multi-dimensional convolution kernel in frequency domain of size $[n_{f,max} \times n_s \times n_r]$

d [dask.array.ndarray] Data in time domain $[n_t \times n_s (\times n_v s)]$ if twosided=False or twosided=True and add_negative=True (with only positive times) or size $[2 * n_t - 1 \times n_s (\times n_v s)]$ 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.

adjoint [bool, optional] Compute and return adjoint(s)

dottest [bool, optional] Apply dot-test

saveGt [bool, optional] Save G and G^H to speed up the computation of adjoint of *pylops_distributed.signalprocessing.Fredholm1* (True) or create G^H on-the-fly (False) Note that saveGt=True will be faster but double the amount of required memory

****kwargs_cgls** Arbitrary keyword arguments for *pylops_distributed.optimization.cg.cgls* solver

Returns

minv [dask.array.ndarray] Inverted model of size $[n_t \times n_r (\times n_{vs})]$ for twosided=False or $[2 * n_t - 1 \times n_r (\times n_v s)]$ for twosided=True

madj [dask.array.ndarray] Adjoint model of size $[n_t \times n_r (\times n_{vs})]$ for twosided=False or $[2 * n_t - 1 \times n_r (\times n_r)]$ for twosided=True

See also:

MDC Multi-dimensional convolution

Notes

Refer to pylops.waveeqprocessing.MDD for implementation details. Note that this implementation is currently missing the wav and causality_precond=False options.

Examples using `pylops_distributed.waveeqprocessing.MDD`

- *09. Multi-Dimensional Deconvolution*

pylops_distributed.waveeqprocessing.Marchenko

class pylops_distributed.waveeqprocessing.**Marchenko**(*R*, *nt*, *dt=0.004*, *dr=1.0*, *wav=None*, *toff=0.0*, *nsmooth=10*, *saveRt=False*, *prescaled=False*, *dtype='float32'*)

Marchenko redatuming

Solve multi-dimensional Marchenko redatuming problem using scipy.sparse.linalg.lsqr iterative solver.

Parameters

R [dask.array] Multi-dimensional reflection response in frequency domain of size $[n_{fmax} \times n_s \times n_r]$. Note that the reflection response should have already been multiplied by 2.

nt [float, optional] Number of samples in time

dt [float, optional] Sampling of time integration axis

dr [float, optional] Sampling of receiver integration axis

wav [numpy.ndarray, optional] Wavelet to apply to direct arrival when created using trav

toff [float, optional] Time-offset to apply to traveltime

nsmooth [int, optional] Number of samples of smoothing operator to apply to window

saveRt [bool, optional] Save R and R^H to speed up the computation of the adjoint of *pylops_distributed.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.

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

Raises

TypeError If t is not numpy.ndarray.

See also:

MDC Multi-dimensional convolution

Notes

Refer to pylops.waveeqprocessing.Marchenko for implementation details.

Attributes

ns [int] Number of samples along source axis

nr [int] Number of samples along receiver axis

shape [tuple] Operator shape

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

Methods

__init__(R, nt[, dt, dr, wav, toff, ...])	Initialize self.
apply_multiplepoints(trav[, dist, G0, nfft, ...])	Marchenko redatuming for multiple points
apply_onepoint(trav[, dist, G0, nfft, rtm, ...])	Marchenko redatuming for one point

apply_onepoint(*trav*, *dist=None*, *G0=None*, *nfft=None*, *rtm=False*, *greens=False*, *dottest=False*, ***kwargs_cgls*)

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 $[n_r \times 1]$

dist: :obj:'numpy.ndarray', optional Distance between subsurface point to surface receivers of size $[nr \times 1]$ (if provided the analytical direct arrival will be computed using a 3d formulation)

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

****kwargs_cgls** Arbitrary keyword arguments for *pylops_distributed.optimization.cg.cgls* solver

Returns

f1_inv_minus [dask.array] Inverted upgoing focusing function of size $[n_r \times n_t]$

f1_inv_plus [dask.array] Inverted downgoing focusing function of size $[n_r \times n_t]$

p0_minus [dask.array] Single-scattering standard redatuming upgoing Green's function of size $[n_r \times n_t]$

g_inv_minus [dask.array] Inverted upgoing Green's function of size $[n_r \times n_t]$

g_inv_plus [dask.array] Inverted downgoing Green's function of size $[n_r \times n_t]$

apply_multiplepoints(*trav*, *dist=None*, *G0=None*, *nfft=None*, *rtm=False*, *greens=False*, *dottest=False*, ***kwargs_cgls*)

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 $[n_r \times n_{vs}]$

dist: :obj:'numpy.ndarray', optional Distance between subsurface point to surface receivers of size $[n_r \times n_{vs}]$ (if provided the analytical direct arrival will be computed using a 3d formulation)

G0 [numpy.ndarray, optional] Direct arrival in time domain of size $[n_r \times n_{vs} \times n_t]$ (if None, create arrival using `trav`)

nfft [int, optional] Number of samples in fft when creating the analytical direct wave

rtm [bool, optional] Compute and return rtm redatuming

greens [bool, optional] Compute and return Green's functions

dottest [bool, optional] Apply dot-test

****kwargs_cgls** Arbitrary keyword arguments for *pylops_distributed.optimization.cg.cgls* solver

Returns

f1_inv_minus [numpy.ndarray] Inverted upgoing focusing function of size $[n_r \times n_{vs} \times n_t]$

f1_inv_plus [numpy.ndarray] Inverted downgoing focusing functionof 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_distributed.waveeqprocessing.Marchenko`

- *Marchenko redatuming by inversion*

pylops_distributed.waveeqprocessing.Demigration

pylops_distributed.waveeqprocessing.**Demigration**(*z*, *x*, *t*, *srcs*, *recs*, *vel*, *wav*, *wavcenter*, *y=None*, *mode='eikonal'*, *trav=None*, *nprocesses=None*, *client=None*)

Demigration operator.

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 $[2/3 \times n_s]$

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 [str, optional] Computation mode (`analytic`, `eikonal` or `byot`, see Notes for more details)

trav [numpy.ndarray or dask.array.core.Array, optional] Traveltime table of size $[n_r * n_s \times (n_y *) n_x * n_z]$ To be provided only when `mode='byot'`

nprocesses [str, optional] Number of processes to split computations

client [dask.distributed.client.Client, optional] Dask client. If provided, the traveltime computation will be persisted.

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:

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

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: traveltime curves between source to receiver pairs are computed for every subsurface point and Green's functions are implemented from traveltime look-up tables, placing the reflectivity values at corresponding source-to-receiver time in the data.
- byot: bring your own table. Traveltime table provided directly by user using trav input parameter. Green's functions are then implemented in the same way as previous options.

The adjoint of the demigration operator is a *migration* operator which projects data in the model domain creating an image of the subsurface reflectivity.

1.3.2 Solvers

Low-level solvers

cg(A, y[, x, niter, tol, compute, client])	Conjugate gradient
cgls(A, y[, x, niter, damp, tol, compute, ...])	Conjugate gradient least squares

pylops_distributed.optimization.cg.cg

pylops_distributed.optimization.cg.**cg**(*A*, *y*, *x=None*, *niter=10*, *tol=1e-05*, *compute=False*, *client=None*)

Conjugate gradient

Solve a system of equations given the square operator A and data y using conjugate gradient iterations.

Parameters

A [pylops_distributed.LinearOperator] Operator to invert of size $[N \times N]$

y [dask.array] Data of size $[N \times 1]$

x0 [dask.array, optional] Initial guess

niter [int, optional] Number of iterations

tol [float, optional] Tolerance on residual norm

compute [tuple, optional] Compute intermediate results at the end of every iteration

client [dask.distributed.client.Client, optional] Dask client. If provided when compute=None each iteration is persisted. This is the preferred method to avoid repeating computations.

Returns

x [dask.array] Estimated model

iter [int] Number of executed iterations

Notes

Solve the the following problem using conjugate gradient iterations:

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

Note that early stopping based on tol is activated only when client is provided or compute=True. The formed approach is preferred as it avoid repeating computations along the compute tree.

pylops_distributed.optimization.cg.cgls

pylops_distributed.optimization.cg.**cgls**(*A*, *y*, *x=None*, *niter=10*, *damp=0.0*, *tol=0.0001*, *compute=False*, *client=None*)

Conjugate gradient least squares

Solve an overdetermined system of equations given an operator A and data y using conjugate gradient iterations.

Parameters

A [pylops_distributed.LinearOperator] Operator to invert of size $[N \times N]$

y [dask.array] Data of size $[N \times 1]$

x0 [dask.array, optional] Initial guess

niter [int, optional] Number of iterations

damp [float, optional] Damping coefficient

tol [float, optional] Tolerance on residual norm

compute [tuple, optional] Compute intermediate results at the end of every iteration

client [dask.distributed.client.Client, optional] Dask client. If provided when compute=None each iteration is persisted. This is the preferred method to avoid repeating computations.

Returns

x [dask.array] Estimated model

iit [int] Number of executed iterations

Notes

Minimize the following functional using conjugate gradient iterations:

$$J = ||\mathbf{y} - \mathbf{A}\mathbf{x}||^2 + \epsilon||\mathbf{x}||^2$$

where ϵ is the damping coefficient.

Note that early stopping based on `tol` is activated only when client is provided or `compute=True`. The formed approach is preferred as it avoid repeating computations along the compute tree.

1.3.3 Applications

Wave-Equation processing

LSM(z, x, t, srcs, recs, vel, wav, wavcenter)	Least-squares Migration (LSM).

pylops_distributed.waveeqprocessing.LSM

class pylops_distributed.waveeqprocessing.**LSM**(*z*, *x*, *t*, *srcs*, *recs*, *vel*, *wav*, *wavcenter*, *y=None*, *mode='eikonal'*, *trav=None*, *dottest=False*, *nprocesses=None*, *client=None*)

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)

trav [`numpy.ndarray`, optional] Traveltime table of size $[(n_y *) n_x * n_z \times n_r * n_s]$ (to be provided if `mode='byot'`)

dottest [`bool`, optional] Apply dot-test

enginetrav [`str`, optional] Engine used for traveltime computation when `mode='eikonal'` (`numpy` and `dask` supported)

engine [`str`, optional] Engine used for `pylops.basicoperators.Spread` computation in forward and adjoint modelling operations (`numpy`, `numba`, or `dask`)

nprocesses [`str`, optional] Number of processes to split computations on (if `engine=dask`)

See also:

pylops.waveeqprocessing.Demigration Demigration operator

Notes

Inverting a demigration operator is generally referred in the literature as least-squares migration (LSM) as historically a least-squares cost function has been used for this purpose. In practice any other cost function could be used, for examples if `solver='pylops.optimization.sparsity.FISTA'` a sparse representation of reflectivity is produced as result of the inversion.

Finally, it is worth noting that in the first iteration of an iterative scheme aimed at inverting the demigration operator, a projection of the recorded data in the model domain is performed and an approximate (band-limited) image of the subsurface is created. This process is referred to in the literature as *migration*.

Attributes

Demop [`pylops.LinearOperator`] Demigration operator

Methods

`__init__`(z, x, t, srcs, recs, vel, wav, ...)	Initialize self.
`solve`(d[, solver])	Solve least-squares migration equations with chosen `solver`

`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 $[(n_y \times) n_x \times n_z]$

1.4 PyLops-distributed 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.

1.4.1 Shared

Backends

`backend.dask`([hardware, client, processes, ...])	Dask backend initialization.

pylops_distributed.utils.backend.dask

pylops_distributed.utils.backend.**dask**(*hardware='single'*, *client=None*, *processes=False*, *n_workers=1*, *threads_per_worker=1*, ***kwargscluster*)

Dask backend initialization.

Create connection to drive computations using Dask distributed.

Parameters

hardware [str, optional] Hardware used to run Dask distributed. Currently available options are `single` for single-machine distribution, `ssh` for SSH-bases multi-machine distribution and `pbs` for PBS-bases multi-machine distribution

client [str, optional] Name of scheduler (use `None` for `hardware=single`).

processes [str, optional] Whether to use processes (`True`) or threads (`False`).

n_workers [int, optional] Number of workers

threads_per_worker [int, optional] Number of threads per each worker

kwargscluster: Additional parameters to be passed to the cluster creation routine

Returns

client [`dask.distributed.client.Client`] Client

cluster : Cluster

Raises

NotImplementedError If `hardware` is not `single`, `ssh`, or `pbs`

Dot-test

dottest(Op, nr, nc, chunks[, tol, ...])	Dot test.

pylops_distributed.utils.dottest

pylops_distributed.utils.**dottest**(*Op*, *nr*, *nc*, *chunks*, *tol=1e-06*, *complexflag=0*, *raiseerror=True*, *verb=False*)

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)

chunks [tuple, optional] Chunks for data and model

tol [float, optional] Dottest tolerance

complexflag [bool, optional] generate random vectors with real (0) or complex numbers (1: only model, 2: only data, 3:both)

raiseerror [bool, optional] Raise error or simply return False when dottest fails

verb [bool, optional] Verbosity

Raises

ValueError If dot-test is not verified within chosen tolerance.

Notes

A dot-test is mathematical tool used in the development of numerical linear operators.

More specifically, a correct implementation of forward and adjoint for a linear operator should verify the following *equality* within a numerical tolerance:

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

1.5 Contributing

Contributions are welcome and greatly appreciated!

Follow the instructions in our main repository

1.6 Changelog

1.6.1 Version 0.2.0

Released on: 06/06/2020

- Added prescaled input parameter to *pylops_distributed.waveeqprocessing.MDC* and *pylops_distributed.waveeqprocessing.Marchenko*
- Added dtype parameter to the FFT calls in the definition of the *pylops_distributed.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.
- Changed handling of dtype in *pylops_distributed.signalprocessing.FFT* to ensure that the type of the input vector is retained when applying forward and adjoint.
- Added PBS backend to *pylops_distributed.utils.backend.dask*

1.6.2 Version 0.1.0

Released on: 09/02/2020

- Added *pylops_distributed.Restriction* operator
- Added *pylops_distributed.signalprocessing.Convolve1D* and pylops_distributed.signalprocessing.FFT2D operators
- Improved efficiency of *pylops_distributed.signalprocessing.Fredholm1* when saveGt=False
- Adapted *pylops_distributed.optimization.cg.cg* and *pylops_distributed.optimization.cg.cgls* solvers for complex numbers

1.6.3 Version 0.0.0

Released on: 01/09/2019

- First official release.

1.7 Roadmap

Coming soon...

1.8 Contributors

- Matteo Ravasi, mrava87

Index