PyLops-GPU

May 03, 2021

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 to run operators on GPUs.

As much as numpy and scipy lie at the core of the parent project PyLops, PyLops-GPU heavily builds on top of PyTorch and takes advantage of the same optimized tensor computations used in PyTorch for deep learning using GPUs and CPUs. Doing so, linear operators can be computed on GPUs.

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 = int(1e6)
x = np.ones(n)
d = np.arange(n) + 1.

Dop = Diagonal(d)

# y = Dx
y = Dop*x
```

and similarly using PyLops-GPU:

```
import numpy as np
import torch
from pylops_gpu.utils.backend import device
from pylops_gpu import Diagonal

dev = device() # will return 'gpu' if GPU is available

n = int(1e6)
x = torch.ones(n, dtype=torch.float64).to(dev)
d = (torch.arange(0, n, dtype=torch.float64) + 1.).to(dev)

Dop = Diagonal(d, device=dev)

# y = Dx
y = Dop*x
```

Running these two snippets of code in Google Colab with GPU enabled gives a 50+ speed up for the forward pass.

As a by-product of implementing PyLops linear operators in PyTorch, we can easily chain our operators with any non-linear mathematical operation (e.g., log, sin, tan, pow, ...) as well as with operators from the `torch.nn` submodule and obtain *Automatic Differentiation* (AD) for the entire chain. Since the gradient of a linear operator is simply its *adjoint*, we have implemented a single class, *pylops_gpu.TorchOperator*, which can wrap any linear operator from PyLops and PyLops-gpu libraries and return a `torch.autograd.Function` object.

CHAPTER 1

History

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

1.1 Installation

You will need **Python 3.6 or greater** to get started.

1.1.1 Dependencies

Our mandatory dependencies are limited to:

- numpy
- scipy
- pytorch
- pytorch_complex_tensor
- 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

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

```
>> pip install pylops-gpu
```

Alternatively, to access the latest source from github:

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

or just clone the repository

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

Note: you may see an error if *pytorch-complex-tensor* has not been previously installed. In that case first run *pip install pytorch-complex-tensor* and then install pylops-gpu

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-gpu.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-gpu
```

1.2 Tutorials

1.2.1 01. Automatic Differentiation

This tutorial focuses on one of the two main benefits of re-implementing some of PyLops linear operators within the PyTorch framework, namely the possibility to perform Automatic Differentiation (AD) on chains of operators which can be:

- native PyTorch mathematical operations (e.g., `torch.log`, `torch.sin`, `torch.tan`, `torch.pow`, ...)
- neural network operators in `torch.nn`
- PyLops and/or PyLops-gpu linear operators

This opens up many opportunities, such as easily including linear regularization terms to nonlinear cost functions or using linear preconditioners with nonlinear modelling operators.

```
import numpy as np
import torch
import matplotlib.pyplot as plt
from torch.autograd import gradcheck

import pylops_gpu
from pylops_gpu.utils.backend import device

dev = device()
plt.close('all')
np.random.seed(10)
torch.manual_seed(10)
```

Out:

```
<torch._C.Generator object at 0x7fbc865527b0>
```

In this example we consider a simple multidimensional functional:

$$\mathbf{y} = \mathbf{A} sin(\mathbf{x})$$

and we use AD to compute the gradient with respect to the input vector evaluated at $\mathbf{x} = \mathbf{x}_0$: $\mathbf{g} = d\mathbf{y}/d\mathbf{x}|_{\mathbf{x}=\mathbf{x}_0}$.

Let's start by defining the Jacobian:

$$\mathbf{J} = \begin{bmatrix} dy_1/dx_1 & ... & dy_1/dx_M \\ ... & ... & ... \\ dy_N/dx_1 & ... & dy_N/dx_M \end{bmatrix} = \begin{bmatrix} a_{11}cos(x_1) & ... & a_{1M}cos(x_M) \\ ... & ... & ... \\ a_{N1}cos(x_1) & ... & a_{NM}cos(x_M) \end{bmatrix} = \mathbf{A}cos(\mathbf{x})$$

Since both input and output are multidimensional, PyTorch `backward` actually computes the product between the transposed Jacobian and a vector $\mathbf{v}$: $\mathbf{g} = \mathbf{J^T v}$.

To validate the correctness of the AD result, we can in this simple case also compute the Jacobian analytically and apply it to the same vector $\mathbf{v}$ that we have provided to PyTorch `backward`.

```
nx, ny = 10, 6
x0 = torch.arange(nx, dtype=torch.double, requires_grad=True)

# Forward
A = torch.normal(0., 1., (ny, nx), dtype=torch.double)
Aop = pylops_gpu.TorchOperator(pylops_gpu.MatrixMult(A))
y = Aop.apply(torch.sin(x0))

# AD
v = torch.ones(ny, dtype=torch.double)
y.backward(v, retain_graph=True)
adgrad = x0.grad

# Analytical
J = (A * torch.cos(x0))
anagrad = torch.matmul(J.T, v)

print('Input: ', x0)
print('AD gradient: ', adgrad)
print('Analytical gradient: ', anagrad)
```

Out:

```
Input:  tensor([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.], dtype=torch.float64,
       requires_grad=True)
AD gradient:  tensor([-0.0695,  0.6679, -0.1115,  5.8981, -0.2886,  0.3653,  2.6875, -
↪2.1607,
        -0.4924,  0.2005], dtype=torch.float64)
Analytical gradient:  tensor([-0.0695,  0.6679, -0.1115,  5.8981, -0.2886,  0.3653, ␣
↪2.6875, -2.1607,
        -0.4924,  0.2005], dtype=torch.float64, grad_fn=<MvBackward>)
```

Similarly we can use the `torch.autograd.gradcheck` directly from PyTorch. Note that doubles must be used for this to succeed with very small *eps* and *atol*

```
input = (torch.arange(nx, dtype=torch.double, requires_grad=True),
         Aop.matvec, Aop.rmatvec, Aop.pylops, Aop.device)
test = gradcheck(Aop.Top, input, eps=1e-6, atol=1e-4)
print(test)
```

Out:

```
True
```

Note that while matrix-vector multiplication could have been performed using the native PyTorch operator `torch.matmul`, in this case we have shown that we are also able to use a PyLops-gpu operator wrapped in *pylops_gpu.TorchOperator*. As already mentioned, this gives us the ability to use much more complex linear operators provided by PyLops within a chain of mixed linear and nonlinear AD-enabled operators.

Total running time of the script: (0 minutes 0.008 seconds)

1.2.2 02. Post-stack inversion

This tutorial focuses on extending post-stack seismic inversion to GPU processing. We refer to the equivalent PyLops tutorial for a more detailed description of the theory.

```
# sphinx_gallery_thumbnail_number = 2
import numpy as np
import torch
import matplotlib.pyplot as plt
from scipy.signal import filtfilt
from pylops.utils.wavelets import ricker

import pylops_gpu
from pylops_gpu.utils.backend import device

dev = device()
plt.close('all')
np.random.seed(10)
torch.manual_seed(10)
```

Out:

```
<torch._C.Generator object at 0x7fbc865527b0>
```

We consider the 1d example. A synthetic profile of acoustic impedance is created and data is modelled using both the dense and linear operator version of `pylops_gpu.avo.poststack.PoststackLinearModelling` operator. Both model and wavelet are created as numpy arrays and converted into torch tensors (note that we enforce `float32` for optimal performance on GPU).

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

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

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

# convert to torch tensors
m = torch.from_numpy(m.astype('float32'))
mback = torch.from_numpy(mback.astype('float32'))
wav = torch.from_numpy(wav.astype('float32'))

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

# lop operator
PPop = pylops_gpu.avo.poststack.PoststackLinearModelling(wav / 2, nt0=nt0)

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

# add noise
dn_dense = d_dense + \
           torch.from_numpy(np.random.normal(0, 2e-2, d_dense.shape).astype('float32
↪'))
```

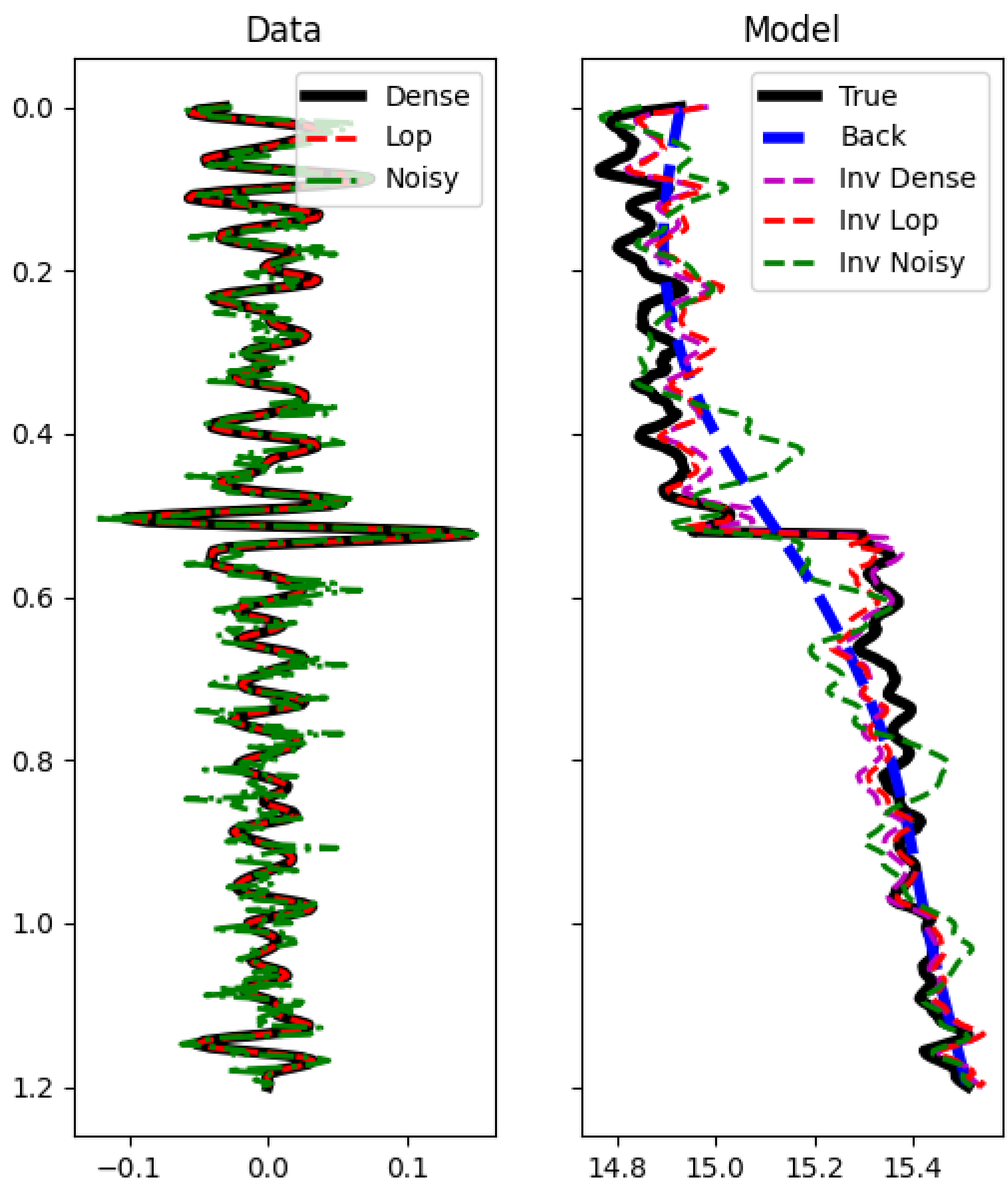
We can now estimate the acoustic profile from band-limited data using either the dense operator or linear operator.

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

# solve lop
minv = \
    pylops_gpu.avo.poststack.PoststackInversion(d_dense, wav / 2, m0=mback,
                                                explicit=False,
                                                simultaneous=False,
                                                **dict(niter=500))[0]
```

(continues on next page)

(continued from previous page)

```
# solve noisy
mn = \
    pylops_gpu.avo.poststack.PoststackInversion(dn_dense, wav / 2, m0=mback,
                                                explicit=True, epsI=1e-4,
                                                epsR=1e0, **dict(niter=100))[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 0x7fbc634cb3c8>
```

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

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

(continues on next page)

(continued from previous page)

```
model = np.load(inputfile)
m = np.log(model['model'][:, ::3])
x, z = model['x'][::3]/1000., model['z']/1000.
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)

# convert to torch tensors
m = torch.from_numpy(m.astype('float32'))
mback = torch.from_numpy(mback.astype('float32'))


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

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

# data
d = (PPop_dense * m.flatten()).reshape(nz, nx)
n = torch.from_numpy(np.random.normal(0, 1e-1, d.shape).astype('float32'))
dn = d + n
```

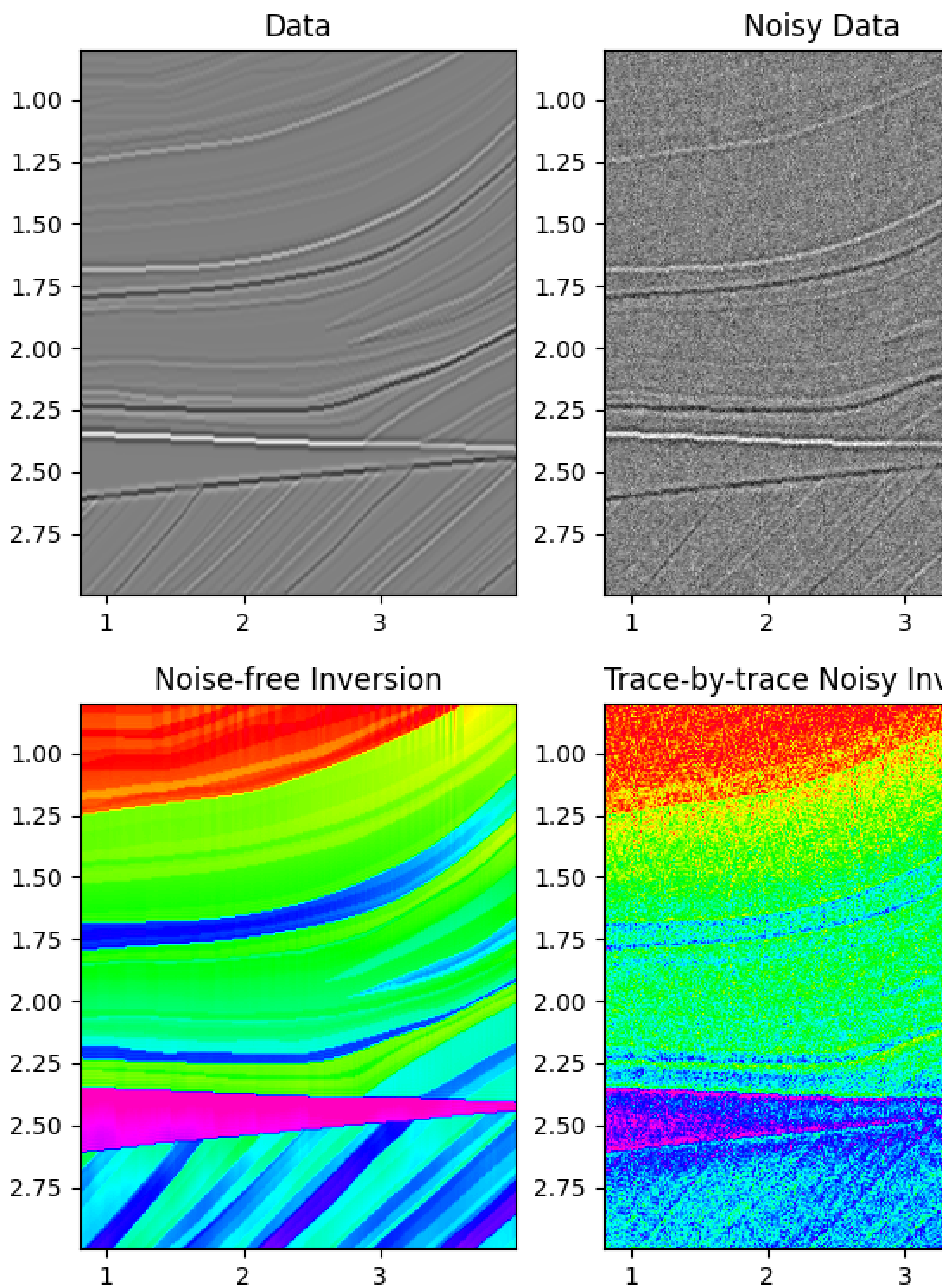
Finally we perform different types of inversion

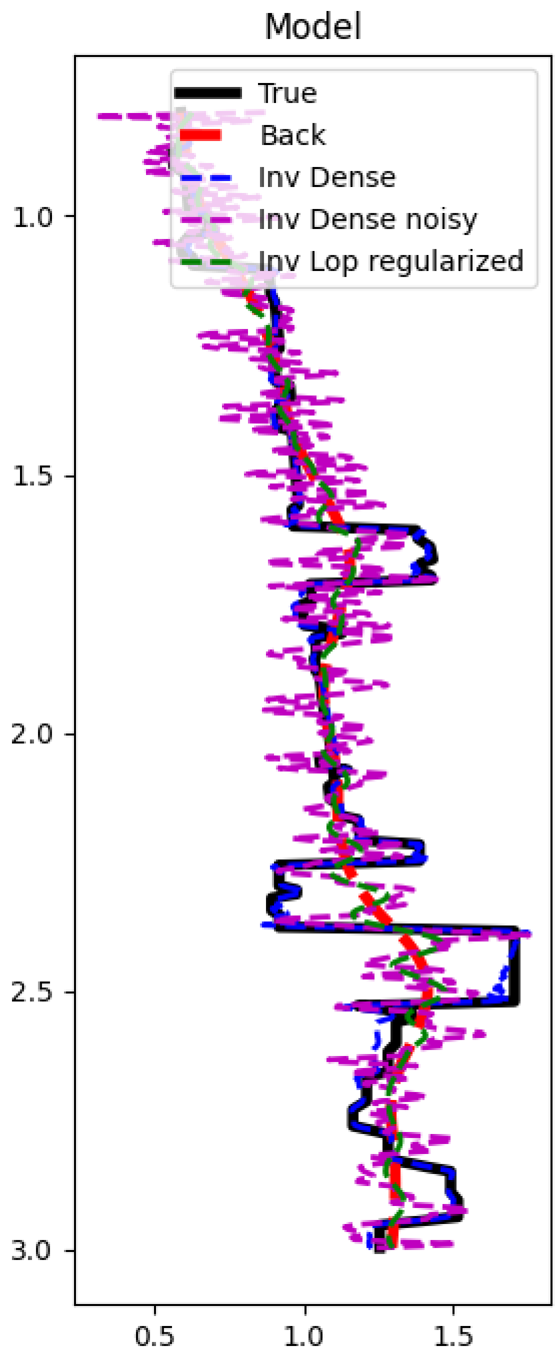
```
# dense inversion with noise-free data
minv_dense = \
    pylops_gpu.avo.poststack.PoststackInversion(d, wav / 2, m0=mback,
                                                explicit=True,
                                                simultaneous=False)[0]

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

# spatially regularized lop inversion with noisy data
minv_lop_reg = \
    pylops_gpu.avo.poststack.PoststackInversion(dn, wav / 2, m0=minv_dense_noisy,
                                                explicit=False,
                                                epsR=5e1, epsI=1e-2,
                                                **dict(niter=80))[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')
```

(continues on next page)

(continued from previous page)

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

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.set_title('Model')
ax.invert_yaxis()
ax.axis('tight')
ax.legend()
plt.tight_layout()
```


Data
Noisy Data
Noise-free Inversion
Trace-by-trace Noisy Inv
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
1
2
3



•

Finally, if you want to run this code on GPUs, take a look at the following notebook and obtain more and more speed-up for problems of increasing size.

Total running time of the script: (0 minutes 4.236 seconds)

1.3 PyLops-GPU API

1.3.1 Linear operators

Templates

`LinearOperator`(shape, dtype[, Op, explicit, ...])	Common interface for performing matrix-vector products.
`TorchOperator`(Op[, batch, pylops, device])	Wrap a PyLops operator into a Torch function.

pylops_gpu.LinearOperator

class pylops_gpu.**LinearOperator**(*shape, dtype, Op=None, explicit=False, device='cpu', togpu=(False, False), tocpu=(False, False)*)

Common interface for performing matrix-vector products.

This class is an overload of the `pylops.LinearOperator` class. It adds functionalities for operators on GPUs; specifically, it allows users specifying when to move model and data from the host to the device and viceversa.

Compared to its equivalent PyLops class `pylops.LinearOperator`, it requires input model and data to be `torch.Tensor` objects.

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-gpu. Find more details regarding implementation of new operators at Implementing new operators.

Parameters

shape [`tuple`] Operator shape

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

Op [`pylops.LinearOperator`] Operator to wrap in `LinearOperator` (if `None`, self must implement `_matvec_` and `_rmatvec_`)

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

device [`str`, optional] Device to be used

togpu [`tuple`, optional] Move model and data from cpu to gpu prior to applying `matvec` and `rmatvec`, respectively (only when `device='gpu'`)

tocpu [`tuple`, optional] Move data and model from gpu to cpu after applying `matvec` and `rmatvec`, respectively (only when `device='gpu'`)

Methods

`__init__`(shape, dtype[, Op, explicit, ...])	Initialize this LinearOperator.
`adjoint`()	Hermitian adjoint.
`apply_columns`(cols)	Apply subset of columns of operator

Continued on next page

Table 2 – continued from previous page

`cond`([uselobpcg])	Condition number of linear operator.
`conj`()	Complex conjugate operator
`div`(y[, niter, tol])	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[, kfirst])	Matrix-matrix multiplication.
`matvec`(x)	Matrix-vector multiplication.
`rmatmat`(X[, kfirst])	Adjoint matrix-matrix multiplication.
`rmatvec`(x)	Adjoint matrix-vector multiplication.
`todense`([backend])	Return dense matrix.
`toimag`([forw, adj])	Imag operator
`toreal`([forw, adj])	Real operator
`tosparse`()	Return sparse matrix.
`transpose`()	Transpose this linear operator.

matvec (*x*)

Matrix-vector multiplication.

Performs the operation `y = A * x` where A is an $N \times M$ linear operator and `x` is a 1-d array.

Parameters

x [`torch.Tensor`] An array with shape (M,)

Returns

y [`torch.Tensor`] An array with shape (N,)

rmatvec (*x*)

Adjoint matrix-vector multiplication.

Performs the operation `y = A^H * x` where A is an $NimesM$ linear operator and `x` is a 1-d array.

Parameters

x [`torch.Tensor`] An array with shape (N,)

Returns

y [`torch.Tensor`] An array with shape (M,)

matmat (*X*, *kfirst=False*)

Matrix-matrix multiplication.

Performs the operation `Y = A * X` where A is an $NimesM$ linear operator and `X` is a 2-d array of size $KimesM$ (`kfirst=True`) or $MimesK$ (`kfirst=False`).

Parameters

x [`torch.Tensor`] An array with shape (M, K) or (K, M)

kfirst [`bool`, optional] Dimension `K` along which the matrix multiplication is performed is in the first dimension (`True`) or in the second dimension (`False`)

Returns

y [`torch.Tensor`] An array with shape (N, K) or (K, N)

rmatmat (*X*, *kfirst=False*)

Adjoint matrix-matrix multiplication.

Performs the operation `Y = A^H * X` where A is an $NimesM$ linear operator and `X` is a 2-d array of size $KimesN$ (`kfirst=True`) or $NimesK$ (`kfirst=False`).

Parameters

x [`torch.Tensor`] An array with shape (N, K) or (K, N)

kfirst [`bool`, optional] Dimension `K` along which the matrix multiplication is performed is in the first dimension (`True`) or in the second dimension (`False`)

Returns

y [`torch.Tensor`] An array with shape (M, K) or (K, M)

`dot`(*x*)

Matrix-vector multiplication.

Parameters

x [`torch.Tensor` or `pytorch_complex_tensor.ComplexTensor`] 1-d or 2-d array, representing a vector or matrix.

Returns

Ax [`torch.Tensor` or `pytorch_complex_tensor.ComplexTensor`] 1-d or 2-d array (depending on the shape of `x`) that represents the result of applying this linear operator on `x`.

`adjoint`()

Hermitian adjoint.

Returns the Hermitian adjoint. Can be abbreviated self.H instead of self.adjoint().

`H`

Hermitian adjoint.

Returns the Hermitian adjoint. Can be abbreviated self.H instead of self.adjoint().

`div`(*y*, *niter=100*, *tol=0.0001*)

Solve the linear problem $\mathbf{y} = \mathbf{A}\mathbf{x}$.

Overloading of operator `/` to improve expressivity of *Pylops_gpu* when solving inverse problems.

Parameters

y [`torch.Tensor`] Data

niter [`int`, optional] Number of iterations (to be used only when `explicit=False`)

tol [`int`] Residual norm tolerance

Returns

xest [`np.ndarray`] Estimated model

Examples using `pylops_gpu.LinearOperator`

- sphx_glr_gallery_plot_convolve.py
- sphx_glr_gallery_plot_derivative.py
- sphx_glr_gallery_plot_diagonal.py
- sphx_glr_gallery_plot_fista.py
- sphx_glr_gallery_plot_identity.py

- sphx_glr_gallery_plot_matrixmult.py
- sphx_glr_gallery_plot_tvreg.py
- *01. Automatic Differentiation*

pylops_gpu.TorchOperator

class pylops_gpu.**TorchOperator**(*Op*, *batch=False*, *pylops=False*, *device='cpu'*)

Wrap a PyLops operator into a Torch function.

This class can be used to wrap a pylops (or pylops-gpu) operator into a torch function. Doing so, users can mix native torch functions (e.g. basic linear algebra operations, neural networks, etc.) and pylops operators.

Since all operators in PyLops are linear operators, a Torch function is simply implemented by using the forward operator for its forward pass and the adjont operator for its backward (gradient) pass.

Parameters

Op [*pylops_gpu.LinearOperator* or pylops.LinearOperator] PyLops operator

batch [bool, optional] Input has single sample (False) or batch of samples (True). If batch==False the input must be a 1-d Torch tensor, if *batch==False*' the input must be a 2-d Torch tensor with batches along the first dimension

pylops [bool, optional] Op is a pylops operator (True) or a pylops-gpu operator (False)

device [str, optional] Device to be used for output vectors when Op is a pylops operator

Returns

y [torch.Tensor] Output array resulting from the application of the operator to x.

Methods

__init__(Op[, batch, pylops, device])	Initialize self.
apply(x)	Apply forward pass to input vector

apply(*x*)

Apply forward pass to input vector

Parameters

x [torch.Tensor] Input array

Returns

y [torch.Tensor] Output array resulting from the application of the operator to x.

Examples using `pylops_gpu.TorchOperator`

- *01. Automatic Differentiation*

Basic operators

MatrixMult(A[, dims, device, togpu, tocpu, . . .])	Matrix multiplication.
Identity(N[, M, inplace, complex, device, . . .])	Identity operator.
Diagonal(diag[, dims, dir, device, togpu, . . .])	Diagonal operator.
VStack(ops[, device, togpu, tocpu, dtype])	Vertical stacking.

pylops_gpu.MatrixMult

class pylops_gpu.**MatrixMult**(*A*, *dims=None*, *device='cpu'*, *togpu=(False*, *False)*, *tocpu=(False*, *False)*, *dtype=torch.float32*)

Matrix multiplication.

Simple wrapper to `torch.matmul` for an input matrix $\mathbf{A}$.

Parameters

A [`torch.Tensor` or `pytorch_complex_tensor.ComplexTensor` or `numpy.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).

device [`str`, optional] Device to be used

togpu [`tuple`, optional] Move model and data from cpu to gpu prior to applying `matvec` and `rmatvec`, respectively (only when `device='gpu'`)

tocpu [`tuple`, optional] Move data and model from gpu to cpu after applying `matvec` and `rmatvec`, respectively (only when `device='gpu'`)

dtype [`torch.dtype` or `np.dtype`, 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, device, togpu, tocpu, 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, tol])	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[, kfirst])	Matrix-matrix multiplication.
`matvec`(x)	Matrix-vector multiplication.
`rmatmat`(X[, kfirst])	Adjoint matrix-matrix multiplication.

Continued on next page

Table 5 – continued from previous page

rmatvec(x)	Adjoint matrix-vector multiplication.
todense([backend])	Return dense matrix.
toimag([forw, adj])	Imag operator
toreal([forw, adj])	Real operator
tosparse()	Return sparse matrix.
transpose()	Transpose this linear operator.

inv()

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

Returns

Ainv [torch.Tensor] Inverse matrix.

Examples using pylops_gpu.MatrixMult

- sphx_glr_gallery_plot_fista.py
- sphx_glr_gallery_plot_matrixmult.py
- *01. Automatic Differentiation*
- *02. Post-stack inversion*

pylops_gpu.Identity

class pylops_gpu.**Identity**(*N*, *M=None*, *inplace=True*, *complex=False*, *device='cpu'*, *togpu=(False, False)*, *tocpu=(False, False)*, *dtype=torch.float32*)

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).

complex [bool, optional] Input model and data are complex arrays

device [str, optional] Device to be used

togpu [tuple, optional] Move model and data from cpu to gpu prior to applying matvec and rmatvec, respectively (only when device='gpu')

tocpu [tuple, optional] Move data and model from gpu to cpu after applying matvec and rmatvec, respectively (only when device='gpu')

dtype [torch.dtype, optional] Type of elements in input array (if complex=True, provide the type of the real component of the array)

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, complex, device, …])	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, tol])	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[, kfirst])	Matrix-matrix multiplication.
matvec(x)	Matrix-vector multiplication.
rmatmat(X[, kfirst])	Adjoint matrix-matrix multiplication.
rmatvec(x)	Adjoint matrix-vector multiplication.
todense([backend])	Return dense matrix.
toimag([forw, adj])	Imag operator
toreal([forw, adj])	Real operator
tosparse()	Return sparse matrix.
transpose()	Transpose this linear operator.

Examples using pylops_gpu.Identity

- sphx_glr_gallery_plot_identity.py
- sphx_glr_gallery_plot_tvreg.py

pylops_gpu.Diagonal

class pylops_gpu.**Diagonal**(*diag, dims=None, dir=0, device='cpu', togpu=(False, False), tocpu=(False, False), dtype=torch.float32*)

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 [numpy.ndarray or torch.Tensor or pytorch_complex_tensor.ComplexTensor] 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.

device [str, optional] Device to be used

togpu [tuple, optional] Move model and data from cpu to gpu prior to applying matvec and rmatvec, respectively (only when device='gpu')

tocpu [tuple, optional] Move data and model from gpu to cpu after applying matvec and rmatvec, respectively (only when device='gpu')

dtype [torch.dtype, 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, device, togpu, . . .])	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, tol])	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[, kfirst])	Matrix-matrix multiplication.
matrix()	Return diagonal matrix as dense torch.Tensor
matvec(x)	Matrix-vector multiplication.
rmatmat(X[, kfirst])	Adjoint matrix-matrix multiplication.
rmatvec(x)	Adjoint matrix-vector multiplication.
todense([backend])	Return dense matrix.
toimag([forw, adj])	Imag operator
toreal([forw, adj])	Real operator
tosparse()	Return sparse matrix.
transpose()	Transpose this linear operator.

matrix()

Return diagonal matrix as dense torch.Tensor

Returns

densemat [torch.Tensor] Dense matrix.

Examples using `pylops_gpu.Diagonal`

- sphx_glr_gallery_plot_diagonal.py

pylops_gpu.VStack

class `pylops_gpu.`**`VStack`**(*ops*, *device='cpu'*, *togpu=(False, False)*, *tocpu=(False, False)*, *dtype=torch.float32*)

Vertical stacking.

Stack a set of N linear operators vertically.

Parameters

ops [`list`] Linear operators to be stacked

device [`str`, optional] Device to be used

togpu [`tuple`, optional] Move model and data from cpu to gpu prior to applying `matvec` and `rmatvec`, respectively (only when `device='gpu'`)

tocpu [`tuple`, optional] Move data and model from gpu to cpu after applying `matvec` and `rmatvec`, respectively (only when `device='gpu'`)

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[, device, togpu, tocpu, 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, tol])	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[, kfirst])	Matrix-matrix multiplication.
`matvec`(x)	Matrix-vector multiplication.
`rmatmat`(X[, kfirst])	Adjoint matrix-matrix multiplication.
`rmatvec`(x)	Adjoint matrix-vector multiplication.
`todense`([backend])	Return dense matrix.
`toimag`([forw, adj])	Imag operator
`toreal`([forw, adj])	Real operator
`tosparse`()	Return sparse matrix.

Continued on next page

Table 8 – continued from previous page

transpose()	Transpose this linear operator.

Smoothing and derivatives

FirstDerivative(N[, dims, dir, sampling, ...])	First derivative.
SecondDerivative(N[, dims, dir, sampling, ...])	Second derivative.
Laplacian(dims[, dirs, weights, sampling, ...])	Laplacian.

pylops_gpu.FirstDerivative

class pylops_gpu.**FirstDerivative**(*N*, *dims=None*, *dir=0*, *sampling=1.0*, *device='cpu'*, *togpu=(False, False)*, *tocpu=(False, False)*, *dtype=torch.float32*)

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.

device [str, optional] Device to be used

togpu [tuple, optional] Move model and data from cpu to gpu prior to applying matvec and rmatvec, respectively (only when device='gpu')

tocpu [tuple, optional] Move data and model from gpu to cpu after applying matvec and rmatvec, respectively (only when device='gpu')

dtype [torch.dtype or np.dtype, optional] Type of elements in input array.

Notes

Refer to pylops.basicoperators.FirstDerivative for implementation details.

Note that since the Torch implementation is based on a convolution with a compact filter $[0.5, 0., -0.5]$, edges are treated differently compared to the PyLops equivalent operator.

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, device, ...])	Initialize this LinearOperator.
adjoint()	Hermitian adjoint.

Continued on next page

Table 10 – continued from previous page

`apply_columns`(cols)	Apply subset of columns of operator
`cond`([uselobpcg])	Condition number of linear operator.
`conj`()	Complex conjugate operator
`div`(y[, niter, tol])	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[, kfirst])	Matrix-matrix multiplication.
`matvec`(x)	Matrix-vector multiplication.
`rmatmat`(X[, kfirst])	Adjoint matrix-matrix multiplication.
`rmatvec`(x)	Adjoint matrix-vector multiplication.
`todense`([backend])	Return dense matrix.
`toimag`([forw, adj])	Imag operator
`toreal`([forw, adj])	Real operator
`tosparse`()	Return sparse matrix.
`transpose`()	Transpose this linear operator.

Examples using `pylops_gpu.FirstDerivative`

- sphx_glr_gallery_plot_derivative.py
- sphx_glr_gallery_plot_tvreg.py

pylops_gpu.SecondDerivative

class `pylops_gpu.`**`SecondDerivative`**(*N, dims=None, dir=0, sampling=1.0, device='cpu', togpu=(False, False), tocpu=(False, False), dtype=torch.float32*)

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`.

device [`str`, optional] Device to be used

togpu [`tuple`, optional] Move model and data from cpu to gpu prior to applying `matvec` and `rmatvec`, respectively (only when `device='gpu'`)

tocpu [`tuple`, optional] Move data and model from gpu to cpu after applying `matvec` and `rmatvec`, respectively (only when `device='gpu'`)

dtype [`torch.dtype` or `np.dtype`, optional] Type of elements in input array.

Notes

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

Note that since the Torch implementation is based on a convolution with a compact filter $[1., -2., 1.]$, edges are treated differently compared to the PyLops equivalent operator.

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, device, . . .])	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, tol])	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[, kfirst])	Matrix-matrix multiplication.
`matvec`(x)	Matrix-vector multiplication.
`rmatmat`(X[, kfirst])	Adjoint matrix-matrix multiplication.
`rmatvec`(x)	Adjoint matrix-vector multiplication.
`todense`([backend])	Return dense matrix.
`toimag`([forw, adj])	Imag operator
`toreal`([forw, adj])	Real operator
`tosparse`()	Return sparse matrix.
`transpose`()	Transpose this linear operator.

Examples using `pylops_gpu.SecondDerivative`

- sphx_glr_gallery_plot_derivative.py
- sphx_glr_gallery_plot_tvreg.py

pylops_gpu.Laplacian

`pylops_gpu.`**`Laplacian`**(*dims*, *dirs=(0, 1)*, *weights=(1, 1)*, *sampling=(1, 1)*, *device='cpu'*, *togpu=(False, False)*, *tocpu=(False, False)*, *dtype=torch.float32*)

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 `dx` and `dy` for each direction

edge [`bool`, optional] Use reduced order derivative at edges (`True`) or ignore them (`False`)

device [str, optional] Device to be used

togpu [tuple, optional] Move model and data from cpu to gpu prior to applying `matvec` and `rmatvec`, respectively (only when `device='gpu'`)

tocpu [tuple, optional] Move data and model from gpu to cpu after applying `matvec` and `rmatvec`, respectively (only when `device='gpu'`)

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.

Note that since the Torch implementation is based on a convolution with a compact filter $[1., -2., 1.]$, edges are treated differently compared to the PyLops equivalent operator.

Examples using `pylops_gpu.Laplacian`

- sphx_glr_gallery_plot_derivative.py

Signal processing

Convolve1D(N, h[, offset, dims, dir, …])	1D convolution operator.

pylops_gpu.signalprocessing.Convolve1D

class pylops_gpu.signalprocessing.**Convolve1D**(*N, h, offset=0, dims=None, dir=0, zero_edges=False, device='cpu', togpu=(False, False), tocpu=(False, False), dtype=torch.float32*)

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 [torch.Tensor or 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

zero_edges [bool, optional] Zero output at edges (*True*) or not (*False*)

device [str, optional] Device to be used

togpu [tuple, optional] Move model and data from cpu to gpu prior to applying matvec and rmatvec, respectively (only when device='gpu')

tocpu [tuple, optional] Move data and model from gpu to cpu after applying matvec and rmatvec, respectively (only when device='gpu')

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

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, …])	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, tol])	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[, kfirst])	Matrix-matrix multiplication.
matvec(x)	Matrix-vector multiplication.
rmatmat(X[, kfirst])	Adjoint matrix-matrix multiplication.
rmatvec(x)	Adjoint matrix-vector multiplication.
todense([backend])	Return dense matrix.
toimag([forw, adj])	Imag operator
toreal([forw, adj])	Real operator
tosparse()	Return sparse matrix.
transpose()	Transpose this linear operator.

Examples using pylops_gpu.signalprocessing.Convolve1D

- sphx_glr_gallery_plot_convolve.py
- sphx_glr_gallery_plot_fista.py

1.3.2 Solvers

Low-level solvers

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

pylops_gpu.optimization.cg.cg

`pylops_gpu.optimization.cg.`**cg**(*A*, *y*, *x=None*, *niter=10*, *tol=1e-10*)

Conjugate gradient

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

Parameters

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

y [`torch.Tensor`] Data of size $[N \times 1]$

x0 [`torch.Tensor`, optional] Initial guess

niter [`int`, optional] Number of iterations

tol [`int`, optional] Residual norm tolerance

Returns

x [`torch.Tensor`] Estimated model

iiter [`torch.Tensor`] Max number of iterations model

Examples using `pylops_gpu.optimization.cg.cg`

- sphx_glr_gallery_plot_convolve.py
- sphx_glr_gallery_plot_fista.py

pylops_gpu.optimization.cg.cgls

`pylops_gpu.optimization.cg.`**cgls**(*A*, *y*, *x=None*, *niter=10*, *damp=0.0*, *tol=1e-10*)

Conjugate gradient least squares

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

Parameters

A [*pylops_gpu.LinearOperator*] Operator to invert of size $[N \times M]$

y [`torch.Tensor`] Data of size $[N \times 1]$

x0 [`torch.Tensor`, optional] Initial guess

niter [`int`, optional] Number of iterations

damp [`float`, optional] Damping coefficient

tol [`int`, optional] Residual norm tolerance

Returns

x [`torch.Tensor`] Estimated model

iiter [`torch.Tensor`] Max number of iterations model

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.

Least-squares

leastsquares.NormalEquationsInversion(Op, ...)	Inversion of normal equations.

pylops_gpu.optimization.leastsquares.NormalEquationsInversion

pylops_gpu.optimization.leastsquares.**NormalEquationsInversion**(*Op*, *Regs*, *data*, *Weight=None*, *dataregs=None*, *epsI=0*, *epsRs=None*, *x0=None*, *returninfo=False*, *device='cpu'*, ***kwargs_cg*)

Inversion of normal equations.

Solve the regularized normal equations for a system of equations given the operator `Op`, a data weighting operator `Weight` and a list of regularization terms `Regs`

Parameters

Op [*pylops_gpu.LinearOperator*] Operator to invert

Regs [`list`] Regularization operators (`None` to avoid adding regularization)

data [`torch.Tensor`] Data

Weight [*pylops_gpu.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 [`torch.Tensor`, optional] Initial guess

returninfo [`bool`, optional] Return info of CG solver

device [`str`, optional] Device to be used

****kwargs_cg** Arbitrary keyword arguments for `pylops_gpu.optimization.leastsquares.cg` solver

Returns

xinv [`numpy.ndarray`] Inverted model.

Notes

Refer to `pylops..optimization.leastsquares.NormalEquationsInversion` for implementation details.

Examples using `pylops_gpu.optimization.leastsquares.NormalEquationsInversion`

- sphx_glr_gallery_plot_tvreg.py

Sparsity

sparsity.FISTA(Op, data, niter[, eps, ...])	Fast Iterative Soft Thresholding Algorithm (FISTA).
sparsity.SplitBregman(Op, RegsL1, data[, ...])	Split Bregman for mixed L2-L1 norms.

pylops_gpu.optimization.sparsity.FISTA

pylops_gpu.optimization.sparsity.**FISTA**(*Op*, *data*, *niter*, *eps=0.1*, *alpha=None*, *eigsiter=1000*, *eigstol=0*, *tol=1e-10*, *returninfo=False*, *show=False*, *device='cpu'*)

Fast Iterative Soft Thresholding Algorithm (FISTA).

Solve an optimization problem with $L1$ 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_gpu.LinearOperator*] Operator to invert

data [`torch.tensor`] Data

niter [`int`] Number of iterations

eps [`float`, optional] Sparsity damping

alpha [`float`, optional] Step size ($\alpha \le 1/\lambda_{max}(\mathbf{Op}^H\mathbf{Op})$ guarantees convergence. If `None`, estimated to satisfy the condition, otherwise the condition will not be checked)

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

device [`str`, optional] Device to be used

Returns

xinv [`numpy.ndarray`] Inverted model

niter [`int`] Number of effective iterations

cost [`numpy.ndarray`, optional] History of cost function

costdata [`numpy.ndarray`, optional] History of data fidelity term in the cost function

costreg [`numpy.ndarray`, optional] History of regularizer term in the cost function

See also:

SplitBregman Split Bregman for mixed L2-L1 norms.

Notes

Solves the following optimization problem for the operator $\mathbf{Op}$ and the data $\mathbf{d}$:

$$J = ||\mathbf{d} - \mathbf{Op}\mathbf{x}||_2^2 + \epsilon||\mathbf{x}||_1$$

using the Fast Iterative Soft Thresholding Algorithm (FISTA) [1]. This is a modified version of ISTA solver with improved convergence properties and limitied additional computational cost.

Examples using `pylops_gpu.optimization.sparsity.FISTA`

- sphx_glr_gallery_plot_fista.py

pylops_gpu.optimization.sparsity.SplitBregman

`pylops_gpu.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*, *device='cpu'*, ***kwargs_cg*)

Split Bregman for mixed L2-L1 norms.

Solve an unconstrained system of equations with mixed L2-L1 regularization terms given the operator `Op`, a list of L1 regularization terms `RegsL1`, and an optional list of L2 regularization terms `RegsL2`.

Parameters

Op [*`pylops_gpu.LinearOperator`*] Operator to invert

RegsL1 [`list`] L1 regularization operators

data [`torch.Tensor`] Data

niter_outer [`int`] Number of iterations of outer loop

niter_inner [`int`] Number of iterations of inner loop

RegsL2 [`list`] Additional L2 regularization operators (if `None`, L2 regularization is not added to the problem)

dataregsL2 [`list`, optional] L2 Regularization data (must have the same number of elements of `RegsL2` or equal to `None` to use a zero data for every regularization operator in `RegsL2`)

mu [`float`, optional] Data term damping

epsRL1s [`list`] L1 Regularization dampings (must have the same number of elements as `RegsL1`)

epsRL2s [list] L2 Regularization dampings (must have the same number of elements as RegsL2)

tol [float, optional] Tolerance. Stop outer iterations if difference between inverted model at subsequent iterations is smaller than tol

tau [float, optional] Scaling factor in the Bregman update (must be close to 1)

x0 [torch.Tensor, 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

device [str, optional] Device to be used

****kwargs_cg** Arbitrary keyword arguments for pylops_gpu.optimization.leastsquares.cg solver

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 $\mathbf{Op}$ and a set of mixed norm (L2 and L1) regularization terms $\mathbf{R_{L2,i}}$ and $\mathbf{R_{L1,i}}$, respectively:

$$J = \mu/2||\textbf{d} - \textbf{Op}\textbf{x}|||_2 + \sum_i \epsilon_{R_{L2,i}}||\mathbf{d_{R_{L2,i}}} - \mathbf{R_{L2,i}}\textbf{x}|||_2 + \sum_i ||\mathbf{R_{L1,i}}\textbf{x}|||_1$$

where μ and $\epsilon_{R_{L2,i}}$ are the damping factors used to weight the different terms of the cost function.

The generalized Split Bergman algorithm is used to solve such cost function: the algorithm is composed of a sequence of unconstrained inverse problems and Bregman updates. Note that the L1 terms are not weighted in the original cost function but are first converted into constraints and then re-inserted in the cost function with Lagrange multipliers $\epsilon_{R_{L1,i}}$, which effectively act as damping factors for those terms. See [1] for detailed derivation.

The scipy.sparse.linalg.lsqr solver and a fast shrinkage algorithm are used within the inner loop to solve the unconstrained inverse problem, and the same procedure is repeated niter_outer times until convergence.

Examples using pylops_gpu.optimization.sparsity.SplitBregman

- sphx_glr_gallery_plot_tvreg.py

1.3.3 Applications

Geophysical subsurface characterization

poststack.PoststackInversion(data, wav[, ...])	Post-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 $[n_{t0}(\times n_x \times n_y)]$

wav [`np.ndarray`] Wavelet in time domain (must have odd number of elements and centered to zero). If 1d, assume stationary wavelet for the entire time axis. If 2d of size $[n_{t0} \times n_h]$ use as non-stationary wavelet

m0 [`np.ndarray`, optional] Background model of size $[n_{t0}(\times n_x \times n_y)]$

explicit [`bool`, optional] Create a chained linear operator (`False`, preferred for large data) or a `MatrixMult` linear operator with dense matrix (`True`, preferred for small data)

simultaneous [`bool`, optional] Simultaneously invert entire data (`True`) or invert trace-by-trace (`False`) when using `explicit` operator (note that the entire data is always inverted when working with linear operator)

epsI [`float`, optional] Damping factor for Tikhonov regularization term

epsR [`float`, optional] Damping factor for additional Laplacian regularization term

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

epsRL1 [`float`, optional] Damping factor for additional blockiness regularization term

****kwargs_solver** Arbitrary keyword arguments for `scipy.linalg.lstsq` solver (if `explicit=True` and `epsR=None`) or `scipy.sparse.linalg.lsqr` solver (if `explicit=False` and/or `epsR` is not `None`)

Returns

minv [`np.ndarray`] Inverted model of size $[n_{t0}(\times n_x \times n_y)]$

datar [`np.ndarray`] Residual data (i.e., data - background data) of size $[n_{t0}(\times n_x \times n_y)]$

Notes

The cost function and solver used in the seismic post-stack inversion module depends on the choice of `explicit`, `simultaneous`, `epsI`, and `epsR` parameters:

- `explicit=True`, `epsI=None` and `epsR=None`: the explicit solver `scipy.linalg.lstsq` is used if `simultaneous=False` (or the iterative solver `scipy.sparse.linalg.lsqr` is used if `simultaneous=True`)
- `explicit=True` with `epsI` and `epsR=None`: the regularized normal equations $\mathbf{W}^T\mathbf{d} = (\mathbf{W}^T\mathbf{W} + \epsilon_I^2\mathbf{I})\mathbf{AI}$ 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.

1.4 PyLops-GPU Utilities

Alongside with its *Linear Operators* and *Solvers*, PyLops-GPU contains also a number of auxiliary routines.

1.4.1 Shared

Backends

`backend.device`()	Automatically identify device to be used with PyTorch

pylops_gpu.utils.backend.device

`pylops_gpu.utils.backend.`**`device`**`()`

Automatically identify device to be used with PyTorch

Returns

device [`str`] Identified device, `cpu` or `gpu`

Examples using `pylops_gpu.utils.backend.device`

- sphx_glr_gallery_plot_convolve.py
- sphx_glr_gallery_plot_derivative.py
- sphx_glr_gallery_plot_fista.py
- sphx_glr_gallery_plot_matrixmult.py
- sphx_glr_gallery_plot_tvreg.py
- *01. Automatic Differentiation*
- *02. Post-stack inversion*

Dot-test

`dottest`(Op, nr, nc[, tol, dtype, …])	Dot test.

pylops_gpu.utils.dottest

`pylops_gpu.utils.dottest`(*Op*, *nr*, *nc*, *tol=1e-06*, *dtype=torch.float32*, *complexflag=0*, *device='cpu'*, *raiseerror=True*, *verb=False*)

Dot test.

Generate random vectors $\mathbf{u}$ and $\mathbf{v}$ 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 [`torch.Tensor`] Linear operator to test.

nr [`int`] Number of rows of operator (i.e., elements in data)

nc [`int`] Number of columns of operator (i.e., elements in model)

tol [`float`, optional] Dottest tolerance

dtype [`torch.dtype`, optional] Type of elements in random vectors

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

device [`str`, optional] Device to be used

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 the following *equality* within a numerical tolerance:

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

Torch2Numpy

`torch2numpy.numpytype_from_torchtype`(torchtype)	Convert torch type into equivalent numpy type
`torch2numpy.torchtype_from_numpytype`(numpytype)	Convert torch type into equivalent numpy type

pylops_gpu.utils.torch2numpy.numpytype_from_torchtype

`pylops_gpu.utils.torch2numpy.numpytype_from_torchtype`(*torchtype*)

Convert torch type into equivalent numpy type

Parameters

torchtype [`torch.dtype`] Torch type

Returns

numpytype [numpy.dtype] Numpy equivalent type

pylops_gpu.utils.torch2numpy.torchtype_from_numpytype

pylops_gpu.utils.torch2numpy.**torchtype_from_numpytype**(*numpytype*)

Convert torch type into equivalent numpy type

Parameters

numpytype [numpy.dtype] Numpy type

Returns

torchtype [torch.dtype] Torch equivalent type

Notes

Given limitations of torch to handle complex numbers, complex numpy types are casted into equivalent real types and the equivalent torch type is returned.

Complex Tensors

complex.complextorch_fromnumpy(x)	Convert complex numpy array into torch ComplexTensor
complex.complexnumpy_fromtorch(xt)	Convert torch ComplexTensor into complex numpy array
complex.conj(x)	Apply complex conjugation to torch ComplexTensor
complex.divide(x, y)	Element-wise division of torch Tensor and torch ComplexTensor.
complex.reshape(x, shape)	Reshape torch ComplexTensor
complex.flatten(x)	Flatten torch ComplexTensor

pylops_gpu.utils.complex.complextorch_fromnumpy

pylops_gpu.utils.complex.**complextorch_fromnumpy**(*x*)

Convert complex numpy array into torch ComplexTensor

Parameters

x [numpy.ndarray] Numpy complex multi-dimensional array

Returns

xt [pytorch_complex_tensor.ComplexTensor] Torch ComplexTensor multi-dimensional array

pylops_gpu.utils.complex.complexnumpy_fromtorch

pylops_gpu.utils.complex.**complexnumpy_fromtorch**(*xt*)

Convert torch ComplexTensor into complex numpy array

Parameters

xt [pytorch_complex_tensor.ComplexTensor] Torch ComplexTensor

Returns

x [numpy.ndarray] Numpy complex multi-dimensional array

pylops_gpu.utils.complex.conj

pylops_gpu.utils.complex.**conj**(*x*)

Apply complex conjugation to torch ComplexTensor

Parameters

x [pytorch_complex_tensor.ComplexTensor] Torch ComplexTensor

Returns

x [pytorch_complex_tensor.ComplexTensor] Complex conjugated Torch ComplexTensor

pylops_gpu.utils.complex.divide

pylops_gpu.utils.complex.**divide**(*x*, *y*)

Element-wise division of torch Tensor and torch ComplexTensor.

Divide each element of x and y, where one or both of them can contain complex numbers.

Parameters

x [pytorch_complex_tensor.ComplexTensor or torch.Tensor] Numerator

y [pytorch_complex_tensor.ComplexTensor] Denominator

Returns

div [pytorch_complex_tensor.ComplexTensor] Complex conjugated Torch ComplexTensor

pylops_gpu.utils.complex.reshape

pylops_gpu.utils.complex.**reshape**(*x*, *shape*)

Reshape torch ComplexTensor

Parameters

x [pytorch_complex_tensor.ComplexTensor] Torch ComplexTensor

shape [tuple] New shape

Returns

xreshaped [pytorch_complex_tensor.ComplexTensor] Reshaped Torch ComplexTensor

pylops_gpu.utils.complex.flatten

pylops_gpu.utils.complex.**flatten**(*x*)

Flatten torch ComplexTensor

Parameters

x [`pytorch_complex_tensor.ComplexTensor`] Torch ComplexTensor

Returns

xflattened [`pytorch_complex_tensor.ComplexTensor`] Flattened Torch ComplexTensor

1.5 Contributing

Contributions are welcome and greatly appreciated!

Follow the instructions in our main repository

1.6 Changelog

1.6.1 Version 0.0.1

Released on: 03/05/2021

- Added *`pylops_gpu.optimization.sparsity.FISTA`* and *`pylops_gpu.optimization.sparsity.SplitBregman`* solvers
- Modified *`pylops_gpu.TorchOperator`* to work with cupy arrays
- Modified `pylops_gpu.avo.poststack._PoststackLinearModelling` to use the code written in pylops library whilst still dealing with torch arrays
- Allowed passing numpy dtypes to operators (automatic conversion to torch types)

1.6.2 Version 0.0.0

Released on: 12/01/2020

- First official release.

1.7 Roadmap

Coming soon…

1.8 Contributors

- Matteo Ravasi, mrava87
- Francesco Picetti, fpicetti

Bibliography

[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] 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.

Python Module Index

p

Index