Getting started:

1 History 3

Bibliography 39

Python Module Index 41

Index 43
**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:

```python
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:

```python
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.
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 3.5 or greater** to get started.

### 1.1.1 Dependencies

Our mandatory dependencies are limited to:

- numpy
- scipy
- numba
- pytorch
- 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:
1.1.3 Step-by-step installation for developers

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

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

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

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

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

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

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

```bash
>> make tests
```

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

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

```bash
>> source activate pylops-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)

In this example we consider a simple multidimensional functional:

\[ y = A \sin(x) \]

and we use AD to compute the gradient with respect to the input vector evaluated at \( x = x_0 \): \( g = \frac{dy}{dx}|_{x=x_0} \).

Let’s start by defining the Jacobian:

\[
J = \begin{bmatrix}
dy_1/dx_1 & \ldots & dy_1/dx_M \\
\vdots & \ddots & \vdots \\
dy_N/dx_1 & \ldots & dy_N/dx_M 
\end{bmatrix}
= \begin{bmatrix}
a_{11}\cos(x_1) & \ldots & a_{1M}\cos(x_M) \\
\vdots & \ddots & \vdots \\
a_{N1}\cos(x_1) & \ldots & a_{NM}\cos(x_M) 
\end{bmatrix}
= A\cos(x)
\]

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

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 \( v \) that we have provided to PyTorch `backward`.

```python
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)
```

Out:

```
Input:  
AD gradient:  
Analytical gradient:  
```
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`

```python
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 0.013 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.
# 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]

# solve noisy

(continues on next page)
mn = \
    pylops_gpu.avo.poststack.PoststackInversion(dn_dense, wav / 2, m0=mback,
        explicit=True, epsI=1e-4,
        epsR=1e0, **dict(niter=100))

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

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

Out:

```python
<matplotlib.legend.Legend object at 0x7f77086935f8>
```

(continues on next page)
model = np.load(inputfile)
m = np.log(model['model'][:, ::3])
x, z = model['x'][:, ::3]/1000., model['z'][:, 0]/1000.
x, nz = len(x), len(z)

# smooth model
nsmodel, 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'))

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

# spatially regularized lop inversion with noisy data

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')
(fig, axs)
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()
Chapter 1. History

PyLops-GPU

Data

Noisy Data

Noise-free Inversion

Trace-by-trace Noisy Inversion
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 5.736 seconds)
1.3 PyLops-GPU API

1.3.1 Linear operators

Templates

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>LinearOperator</code></td>
<td>Common interface for performing matrix-vector products.</td>
</tr>
<tr>
<td><code>TorchOperator</code></td>
<td>Wrap a PyLops operator into a Torch function.</td>
</tr>
</tbody>
</table>

**pylops_gpu.LinearOperator**

**class** pylops_gpu.LinearOperator *(shape, dtype[, Op, explicit, ...])*

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 vice versa.

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
Table 2 – continued from previous page

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

**matvec**($x$)
Matrix-vector multiplication.

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

**Parameters**

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

**Returns**

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

**matmat**($X$, kfirst=False)
Matrix-matrix multiplication.

Performs the operation $Y = A \times X$ 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 $Y = A^H \times X$ 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 (M, K) or (K, M)
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 y = Ax.

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**

```python
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 adjoint 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

```

Ex.

```

**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.

Continued on next page
Table 4 – continued from previous page

| **VStack** (ops[, device, togpu, tocpu, dtype]) | Vertical stacking. |

### pylops_gpu.MatrixMult

**class** pylops_gpu.MatrixMult

Matrix multiplication.

Simple wrapper to `torch.matmul` for an input matrix $A$.

**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 $y = Ax$.
- **dot**(x) Matrix-vector multiplication.
- **eigs**([neigs, symmetric, niter, uselobpcg]) Most significant eigenvalues of linear operator.
- **inv**() Return the inverse of $A$.
- **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.

Continued on next page
pylops-gpu

### Table 5 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tosparse()</td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td>transpose()</td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

#### inv()

Return the inverse of \( A \).

**Returns**

\[ \text{Ainv} \text{ [torch.Tensor]} \] 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 vice versa in adjoint mode if \( M = N \). If \( M > N \) removes last \( M - N \) elements from model in forward and pads with 0 in adjoint. If \( N > M \) removes last \( N - M \) elements from data in adjoint and pads with 0 in forward.

**Parameters**

- \( N \) [int] Number of samples in data (and model, if \( M \) is not provided).
- \( M \) [int, optional] Number of samples in model.
- 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

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

Examples using `pylops_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 performed 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 y = Ax.

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.

tosparse() Return sparse matrix.

transpose() Transpose this linear operator.

matrix() Return diagonal matrix as dense torch.Tensor

Returns

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

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

Smoothing and derivatives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FirstDerivative([N[, dims, dir, sampling,...]])</td>
<td>First derivative.</td>
</tr>
<tr>
<td>SecondDerivative([N[, dims, dir, sampling,...]])</td>
<td>Second derivative.</td>
</tr>
<tr>
<td>Laplacian([dims[, dirs, weights, sampling,...]])</td>
<td>Laplacian.</td>
</tr>
</tbody>
</table>

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.

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 \( y = Ax \).

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.

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
class pylops_gpu.SecondDerivative

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 \(y = Ax\).
- 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.

Continued on next page
Table 11 – continued from previous page

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

Examples using `pylops_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

\texttt{Convolve1D(N, h[, offset, dims, dir, ...])}  
1D convolution operator.

\texttt{pylops_gpu.signalprocessing.Convolve1D}

class \texttt{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 \texttt{dir}.

\textbf{Parameters}

- \texttt{N [int]} Number of samples in model.
- \texttt{h [torch.Tensor or numpy.ndarray]} 1d compact filter to be convolved to input signal
- \texttt{offset [int]} Index of the center of the compact filter
- \texttt{dims [tuple]} Number of samples for each dimension (\texttt{None} if only one dimension is available)
- \texttt{dir [int, optional]} Direction along which convolution is applied
- \texttt{zero_edges [bool, optional]} Zero output at edges (\texttt{True}) or not (\texttt{False})
- \texttt{device [str, optional]} Device to be used
- \texttt{togpu [tuple, optional]} Move model and data from cpu to gpu prior to applying \texttt{matvec} and \texttt{rmatvec}, respectively (only when \texttt{device='gpu'}
- \texttt{tocpu [tuple, optional]} Move data and model from gpu to cpu after applying \texttt{matvec} and \texttt{rmatvec}, respectively (only when \texttt{device='gpu'}
- \texttt{dtype [torch.dtype, optional]} Type of elements in input array.

\textbf{Notes}

Refer to \texttt{pylops.signalprocessing.Convolve1D} for implementation details.

\textbf{Attributes}

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

\textbf{Methods}

- \texttt{__init__}(N, h[, offset, dims, dir, ...]) Initialize this LinearOperator.
- \texttt{adjoint()} Hermitian adjoint.
- \texttt{apply_columns(cols)} Apply subset of columns of operator
- \texttt{cond([uselobpcg])} Condition number of linear operator.
- \texttt{conj()} Complex conjugate operator

Continued on next page
Table 13 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>div(y[, niter, tol])</code></td>
<td>Solve the linear problem ( y = Ax ).</td>
</tr>
<tr>
<td><code>dot(x)</code></td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>eigs([neigs, symmetric, niter, uselobpcg])</code></td>
<td>Most significant eigenvalues of linear operator.</td>
</tr>
<tr>
<td><code>matmat(X[, kfirst])</code></td>
<td>Matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>matvec(x)</code></td>
<td>Matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>rmatmat(X[, kfirst])</code></td>
<td>Adjoint matrix-matrix multiplication.</td>
</tr>
<tr>
<td><code>rmatvec(x)</code></td>
<td>Adjoint matrix-vector multiplication.</td>
</tr>
<tr>
<td><code>todense([backend])</code></td>
<td>Return dense matrix.</td>
</tr>
<tr>
<td><code>tosparse()</code></td>
<td>Return sparse matrix.</td>
</tr>
<tr>
<td><code>transpose()</code></td>
<td>Transpose this linear operator.</td>
</tr>
</tbody>
</table>

### Examples using `pylops_gpu.signalprocessing.Convolve1D`

- sphx_glr_gallery_plot_convolve.py
- sphx_glr_gallery_plot_fista.py

### 1.3.2 Solvers

#### Low-level solvers

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cg(A, y[, x, niter, tol])</code></td>
<td>Conjugate gradient</td>
</tr>
<tr>
<td><code>cglsl(A, y[, x, niter, damp, tol])</code></td>
<td>Conjugate gradient least squares</td>
</tr>
</tbody>
</table>

#### `pylops_gpu.optimization.cg.cg`

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
- \( \text{niter} \) [int, optional] Number of iterations
- \( \text{tol} \) [int, optional] Residual norm tolerance

**Returns**

- \( x \) [torch.Tensor] Estimated model
- \( \text{iiter} \) [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

1.3. PyLops-GPU API
**pylops_gpu.optimization.cg.cgls**

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:

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

where $\epsilon$ is the damping coefficient.

**Least-squares**

**leastsquares.NormalEquationsInversion**

Inversion of normal equations.

**pylops_gpu.optimization.leastsquares.NormalEquationsInversion**

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=0.1, 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 ($\alpha \leq 1/\lambda_{\text{max}}(\text{Op}^H \text{Op})$) 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 Op and the data d:

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

using the Fast Iterative Soft Thresholding Algorithm (FISTA) [1]. This is a modified version of ISTA solver with improved convergence properties and limited 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, 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 $\mathbf{O}p$ and a set of mixed norm (L2 and L1) regularization terms $R_{L2,i}$ and $R_{L1,i}$, respectively:

$$J = \frac{\mu}{2} ||d - \mathbf{O}p\mathbf{x}||_2^2 + \sum_i \epsilon_{R_{L2,i}} ||d_{R_{L2,i}} - R_{L2,i}\mathbf{x}||_2^2 + \sum_i ||R_{L1,i}\mathbf{x}||_1$$

where $\mu$ 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 $niter_{\text{outer}}$ times until convergence.

Examples using `pylops_gpu.optimization.sparsity.SplitBregman`

- sphx_glr_gallery_plot_tvreg.py

1.3.3 Applications

Geophysical subsurface characterization

```python
poststack.PoststackInversion(data, wav[, ...])
```

`pylops洪.poststack.PoststackInversion`

- `pylops洪.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 \(W^Td = (W^TW + \varepsilon^2I)AI\) 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`) (np, nr, nc[, tol, dtype, ...])

Dot test.

**pylops_gpu.utils.dottest**

`pylops_gpu.utils.dottest(Op, nr, nc[, tol, dtype, ...])`

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** [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:

\[(\text{Op} \ast \mathbf{u})^H \ast \mathbf{v} = \mathbf{u}^H \ast (\text{Op}^H \ast \mathbf{v})\]

Torch2Numpy

\[
\text{torch2numpy.numpypype_from_torchtype (torchtype)}
\]

Convert torch type into equivalent numpy type

Parameters

- torchtype [torch.dtype] Torch type

Returns

- numpypype [numpypype] Numpy equivalent type

\[
\text{torch2numpy.torchtype_from_numpypype (numpypype)}
\]

Convert torch type into equivalent numpy type

Parameters

- numpypype [numpypype] Numpy type

Returns

- torchtype [torch.dtype] Torch equivalent type

Notes

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

Complex Tensors
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convert.complextorch_fromnumpy(x)</code></td>
<td>Converts complex numpy array into torch ComplexTensor</td>
</tr>
<tr>
<td><code>convert.complexnumpy_fromtorch(xt)</code></td>
<td>Converts torch ComplexTensor into complex numpy array</td>
</tr>
<tr>
<td><code>convert.conj(x)</code></td>
<td>Applies complex conjugation to torch ComplexTensor</td>
</tr>
<tr>
<td><code>convert.divide(x, y)</code></td>
<td>Element-wise division of torch Tensor and torch ComplexTensor.</td>
</tr>
<tr>
<td><code>convert.reshape(x, shape)</code></td>
<td>Reshape torch ComplexTensor</td>
</tr>
<tr>
<td><code>convert.flatten(x)</code></td>
<td>Flatten torch ComplexTensor</td>
</tr>
</tbody>
</table>

### pylops_gpu.utils.complex.complextorch_fromnumpy

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

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

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

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**
- $\text{div}$ [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**
- $x$reshaped [pytorch_complex_tensor.ComplexTensor] Reshaped Torch Complex-Tensor

**pylops_gpu.utils.complex.flatten**

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

Flatten torch ComplexTensor

**Parameters**
- $x$ [pytorch_complex_tensor.ComplexTensor] Torch ComplexTensor

**Returns**
- $x$flattened [pytorch_complex_tensor.ComplexTensor] Flattened Torch Complex-Tensor

**1.5 Contributing**

Contributions are welcome and greatly appreciated!

Follow the instructions in our main repository

**1.6 Changelog**

**1.6.1 Version 0.0.0**

Released on: 12/01/2020
- First official release.
1.7 Roadmap

Coming soon…

1.8 Contributors

- Matteo Ravasi, mrava87
- Francesco Picetti, fpicetti

p

pylops_gpu, 14
Index

A

adjoint() (pylops_gpu.LinearOperator method), 16
apply() (pylops_gpu.TorchOperator method), 17

C
cg() (in module pylops_gpu.optimization.cg), 27
cgls() (in module pylops_gpu.optimization.cg), 28
complexnumpy_fromtorch() (in module pylops_gpu.utils.complex), 36
complextorch_fromnumpy() (in module pylops_gpu.utils.complex), 36
conj() (in module pylops_gpu.utils.complex), 36
Convolve1D (class in pylops_gpu.signalprocessing), 26

D
device() (in module pylops_gpu.utils.backend), 34
Diagonal (class in pylops_gpu), 20
div() (pylops_gpu.LinearOperator method), 16
divide() (in module pylops_gpu.utils.complex), 36
dot() (pylops_gpu.LinearOperator method), 16
dottest() (in module pylops_gpu.utils), 34

F
FirstDerivative (class in pylops_gpu), 22
FISTA() (in module pylops_gpu.optimization.sparsity), 29
flatten() (in module pylops_gpu.utils.complex), 37

H
H (pylops_gpu.LinearOperator attribute), 16

I
Identity (class in pylops_gpu), 19
inv() (pylops_gpu.MatrixMult method), 19

L
Laplacian() (in module pylops_gpu), 25
LinearOperator (class in pylops_gpu), 14

M
matmat() (pylops_gpu.LinearOperator method), 15
matrix() (pylops_gpu.Diagonal method), 21
MatrixMult (class in pylops_gpu), 18
matvec() (pylops_gpu.LinearOperator method), 15

N
NormalEquationsInversion() (in module pylops_gpu.optimization.leastsquares), 28
numpytype_from_torchtype() (in module pylops_gpu.utils.torch2numpy), 35

P
PoststackInversion() (in module pylops.avo.poststack), 32
pylops_gpu (module), 14

R
reshape() (in module pylops_gpu.utils.complex), 37
rmatmat() (pylops_gpu.LinearOperator method), 15
rmatvec() (pylops_gpu.LinearOperator method), 15

S
SecondDerivative (class in pylops_gpu), 24
SplitBregman() (in module pylops_gpu.optimization.sparsity), 31

T
TorchOperator (class in pylops_gpu), 17
torchtype_from_numpytype() (in module pylops_gpu.utils.torch2numpy), 35

V
VStack (class in pylops_gpu), 21