spyrit

Release 2.1.0

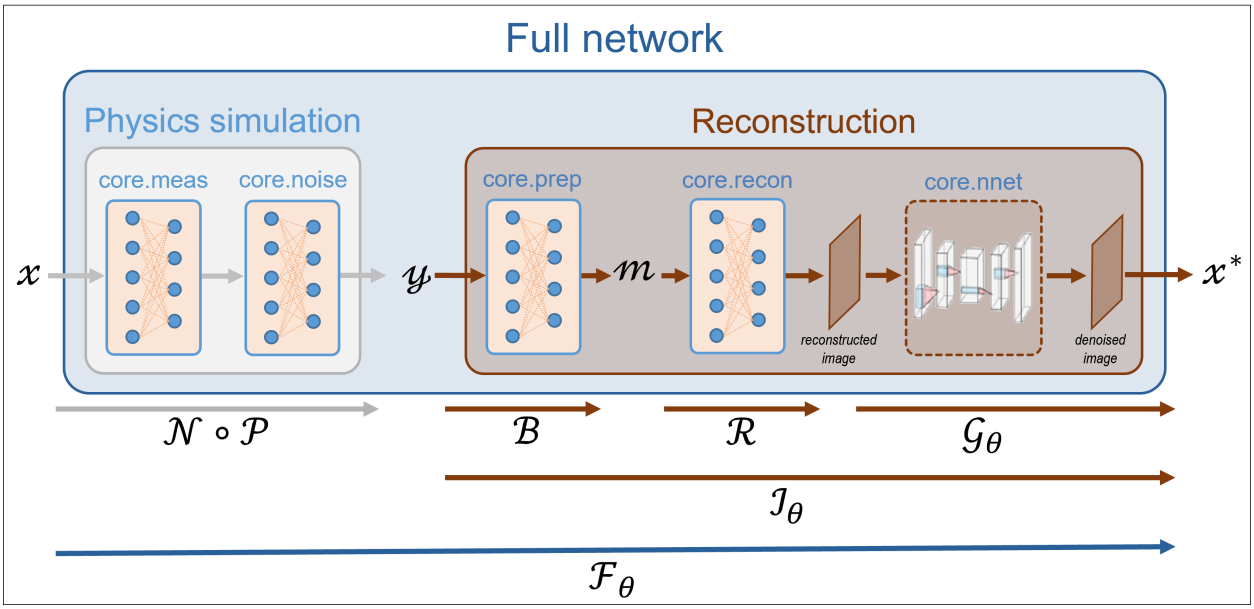
Antonio Tomas Lorente Mur - Nicolas Ducros - Sebastien Crombe

May 14, 2024

SUBPACKAGES

SPyRiT is a PyTorch-based deep image reconstruction package primarily designed for single-pixel imaging.

SPyRiT allows to simulate measurements and perform image reconstruction using a full network structure. It takes a normalized image as input and performs data simulation and image reconstruction in a single forward pass or in separate steps. A full network generally consists of a measurement operator, a noise operator, a preprocessing operator, a reconstruction operator, and a learnable neural network. All operators inherit from PyTorch's *nn.Module* class, which allows them to be easily combined into a full network.



The complete network contains two main parts: a physics simulation part that simulates measurements from images, and a reconstruction part that estimates the unknown image from measurements.

The Physics Simulation part consists of a Measurement operator ($\mathcal{N}$) and a Noise operator ($\mathcal{P}$).

The reconstruction part consists of a preprocessing ($\mathcal{B}$) that produces the pre-processed measurements from the noisy measurements, a reconstruction operator ($\mathcal{R}$) that estimates the unknown image from the pre-processed measurements, and an optional neural network ($\mathcal{G}_\theta$) that can be trained to improve the reconstruction quality.

CHAPTER

ONE

INSTALLATION

The spyrit package is available for Linux, MacOs and Windows:

```
pip install spyrit
```

Advanced installation guidelines are available on GitHub.

CHAPTER

TWO

SINGLE-PIXEL IMAGING

2.1 Modelling of the measurements

Single-pixel imaging aims to recover an image $x \in \Re^N$ from a few noisy scalar products $y \in \Re^M$, where $M \ll N$. We model the acquisition as

$$y = (\mathcal{N} \circ \mathcal{P})(x),$$

where $\mathcal{P}$ is a linear operator, $\mathcal{N}$ is a noise operator, and $\circ$ denotes the composition of operators.

2.2 Image reconstruction

Learning-based reconstruction approaches estimate the unknown image as $x^* = \mathcal{I}_\theta(y)$, where $\mathcal{I}_\theta$ represents the parameters that are learned during a training phase. In the case of supervised learning, **the training phase** solves

$$\min_\theta \sum_i \mathcal{L}\left(x_i, \mathcal{I}_\theta(y_i)\right),$$

where $\mathcal{L}$ is the training loss between the true image x and its estimate, and $\{x_i, y_i\}_i$ is a set of training pairs.

Consider the typical **reconstruction operator** $\mathcal{I}_\theta$ which can be written as:

$$\mathcal{I}_\theta = \mathcal{G}_\theta \circ \mathcal{R} \circ \mathcal{B},$$

where $\mathcal{B}$ is a preprocessing operator, $\mathcal{R}$ is a (standard) linear reconstruction operator, and $\mathcal{G}_\theta$ is a neural network that can be trained during the training phase. Alternatively, $\mathcal{R}$ can be simply "plugged". In this case, it is trained beforehand.

To introduce the **full network**, a forward pass can be written as follows:

$$F_\theta(x) = (\mathcal{G}_\theta \circ \mathcal{R} \circ \mathcal{B} \circ \mathcal{N} \circ \mathcal{P})(x).$$

The full network can be trained using a database containing only images:

$$\min_\theta \sum_i \mathcal{L}\left(x_i, \mathcal{F}_\theta(x_i)\right).$$

This pipeline allows noisy data to be simulated on the fly, providing data augmentation while avoiding storing the measurements.

2.3 Package structure

The main functionalities of SPyRiT are implemented in the subpackage *spyrit.core*, which contains six submodules:

1. **Measurement operators (meas)** compute linear measurements $\mathcal{P}x$ from images x, where $\mathcal{P}$ is a linear operator (matrix) and x is a vectorized image (see *spyrit.core.meas*).
2. **Noise operators (noise)** corrupt measurements $y = (\mathcal{N} \circ \mathcal{P})(x)$ with noise (see *spyrit.core.noise*).
3. **Preprocessing operators (prep)** are used to process noisy measurements, $m = \mathcal{B}(y)$, prior to reconstruction. They typically compensate for the image normalization previously performed (see *spyrit.core.prep*).
4. **Reconstruction operators (recon)** comprise both standard linear reconstruction operators $\mathcal{R}$ and full network definitions $\mathcal{F}_\theta$, which include both forward and reconstruction layers (see *spyrit.core.recon*).
5. **Neural networks (nnet)** include well-known neural networks $\mathcal{G}_\theta$, generally used as denoiser layers (see *spyrit.core.nnet*).
6. **Training (train)** provide the functionalities for training reconstruction networks (see *spyrit.core.train*).

2.4 Subpackages

spyrit.core
spyrit.misc

2.4.1 spyrit.core

Modules

spyrit.core.meas	Measurement operators, static and dynamic.
spyrit.core.nnet	Neural network models for image denoising.
spyrit.core.noise	Noise models for simulating measurements in imaging.
spyrit.core.prep	Preprocessing operators applying affine transformations to the measurements.
spyrit.core.recon	Reconstruction methods and networks.
spyrit.core.time	Stores deformation fields and warps images.
spyrit.core.train	Training functions for deep learning models.

spyrit.core.meas

Measurement operators, static and dynamic.

There are six classes contained in this module, each representing a different type of measurement operator. Three of them are static, i.e. they are used to simulate measurements of still images, and three are dynamic, i.e. they are used to simulate measurements of moving objects, represented as a sequence of images.

Functions

set_dyn_pinv(meas_op, motion[, interp_mode, reg])	

spyrit.core.meas.set_dyn_pinv

spyrit.core.meas.**set_dyn_pinv**(*meas_op:* DynamicLinear, *motion:* DeformationField, *interp_mode: str = 'bilinear'*, *reg: float = 1e-15*) → None

Classes

DynamicHadamSplit(M, h[, Ord])	Simulates the measurement of a moving object using the positive and negative components of a Hadamard matrix.
DynamicLinear(H[, Ord])	Simulates the measurement of a moving object using a measurement matrix.
DynamicLinearSplit(H[, Ord])	Simulates the measurement of a moving object using the positive and negative components of the measurement matrix.
HadamSplit(M, h, Ord)	Simulates the measurement of a still image using the positive and negative components of a Hadamard matrix.
Linear(H[, pinv, reg, Ord])	Simulates the measurement of an still image using a measurement matrix.
LinearSplit(H[, pinv, reg, Ord])	Simulates the measurement of a still image using the computed positive and negative components of the measurement matrix.

spyrit.core.meas.DynamicHadamSplit

class spyrit.core.meas.**DynamicHadamSplit**(*M: int*, *h: int*, *Ord: tensor = None*)

Bases: *DynamicLinearSplit*

Simulates the measurement of a moving object using the positive and negative components of a Hadamard matrix.

Computes linear measurements from incoming images: $y = Px$, where P is a linear operator (matrix) with positive entries and x is a batch of vectorized images representing a motion picture.

The class relies on a Hadamard-based matrix H with shape (M, N) where N represents the number of pixels in the image and $M \leq N$ the number of measurements. H is obtained by selecting a re-ordered subsample of M rows of a "full" Hadamard matrix F with shape (N^2, N^2). N must be a power of 2.

The matrix P is then obtained by splitting the matrix H such that $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$, where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

Args:

`M` (int): Number of measurements

`h` (int): Image height h, must be a power of 2. The image is assumed to be square, so the number of pixels in the image is $N = h^2$.

`Ord` (torch.tensor): Order matrix with shape (h, h) used to select the rows of the full Hadamard matrix F compute the permutation matrix G^T with shape (N, N) (see the *sampling* submodule)

Attributes:

`H` (torch.nn.Parameter): The measurement matrix of shape (M, h^2). It is initialized as a re-ordered subsample of the rows of the "full" Hadamard matrix F with shape (N^2, N^2).

`H_pinv` (torch.nn.Parameter): The pseudo inverse of the measurement matrix of shape (h^2, M). It is initialized as $H^\dagger = \frac{1}{N} H^T$ where $N = h^2$.

`P` (torch.nn.Parameter): The splitted measurement matrix of shape $(2M, h^2)$ initialized as $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$ where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

`Perm` (torch.nn.Parameter): The permutation matrix G^T that is used to re-order the subsample of rows of the "full" Hadamard matrix F according to descreasing value of the order matrix Ord. It has shape (N, N) where $N = h^2$.

`M` (int): Number of measurements performed by the linear operator.

`N` (int): Number of pixels in the image. It is initialized as h^2.

`h` (int): Image height h.

`w` (int): Image width w. The image is assumed to be square, i.e. $w = h$.

Warning: For each call, there must be **exactly** as many images in x as there are measurements in the linear operator used to initialize the class.

Note: The computation of a Hadamard transform Fx benefits a fast algorithm, as well as the computation of inverse Hadamard transforms.

Note: The matrix H has shape (M, N) with $N = h^2$.

Note: $H = H_+ - H_-$

Example:

```
>>> Ord = torch.rand([32,32])
>>> meas_op = HadamSplitDynamic(400, 32, Ord)
>>> print(meas_op)
HadamSplitDynamic(
  (Image pixels): 1024
  (H): torch.Size([400, 1024])
  (P): torch.Size([800, 1024])
  (Perm): torch.Size([1024, 1024])
  )
```

Methods

forward(x)	Simulates the measurement of a motion picture using P.
forward_H(x)	Simulates the measurement of a motion picture using H.
get_H()	Returns the attribute measurement matrix H.
get_H_pinv()	Returns the pseudo inverse of the measurement matrix H.
get_P()	Returns the attribute measurement matrix P.
get_Perm()	
set_H_pinv([reg, pinv])	Stores in self.H_pinv the pseudo inverse of the measurement matrix H.
sort_by_indices(x[, axis, inverse_permutation])	Reorder the rows or columns of a tensor according to the indices.

spyrit.core.meas.DynamicHadamSplit.forward

DynamicHadamSplit.**forward**(*x: tensor*) → tensor

Simulates the measurement of a motion picture using P.

The output y is computed as $y = Px$, where P is the measurement matrix and x is a batch of vectorized (flattened) images.

P contains only positive values and is obtained by splitting a given measurement matrix H such that $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$, where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

The matrix H can contain positive and negative values and is given by the user at initialization.

> **Warning:** There must be **exactly** as many images as there are measurements in the linear operator used to initialize the class, i.e. *P.shape[-2] == x.shape[-2]*

Args:

x: Batch of vectorized (flattened) images of shape $(*, 2M, N)$ where * denotes the batch size, $2M$ the number of measurements in the measurement matrix P and N the number of pixels in the image.

Shape:

x: $(*, 2M, N)$

P has a shape of $(2M, N)$ where M is the number of measurements as defined by the first dimension of H and N is the number of pixels in the image.

output: $(*, 2M)$

Example:

```
>>> x = torch.rand([10, 800, 1600])
>>> H = torch.rand([400, 1600])
>>> meas_op = DynamicLinearSplit(H)
>>> y = meas_op(x)
```

(continues on next page)

(continued from previous page)

```
>>> print(y.shape)
torch.Size([10, 800])
```

spyrit.core.meas.DynamicHadamSplit.forward_H

DynamicHadamSplit.**forward_H**(*x: tensor*) → tensor

Simulates the measurement of a motion picture using H.

The output y is computed as $y = Hx$, where H is the measurement matrix and x is a batch of vectorized (flattened) images. The positive and negative components of the measurement matrix are **not** used in this method.

The matrix H can contain positive and negative values and is given by the user at initialization.

Warning: There must be **exactly** as many images as there are measurements in the linear operator used to initialize the class, i.e. *H.shape[-2:] == x.shape[-2:]*

Args:

x: Batch of vectorized (flatten) images of shape $(*, M, N)$ where * denotes the batch size, and (M, N) is the shape of the measurement matrix H.

Shape:

x: $(*, M, N)$

H has a shape of (M, N) where M is the number of measurements and N is the number of pixels in the image.

output: $(*, M)$

Example:

```
>>> x = torch.rand([10, 400, 1600])
>>> H = torch.rand([400, 1600])
>>> meas_op = LinearDynamicSplit(H)
>>> y = meas_op.forward_H(x)
>>> print(y.shape)
torch.Size([10, 400])
```

spyrit.core.meas.DynamicHadamSplit.get_H

DynamicHadamSplit.**get_H**() → tensor

Returns the attribute measurement matrix H.

Shape:

Output: (M, N)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1)
>>> H2 = meas_op.get_H()
```

(continues on next page)

(continued from previous page)

```
>>> print(H2.shape)
torch.Size([400, 1600])
```

spyrit.core.meas.DynamicHadamSplit.get_H_pinv

DynamicHadamSplit.**get_H_pinv**() → tensor

Returns the pseudo inverse of the measurement matrix H.

Shape:

Output: (N, M)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1, True)
>>> H2 = meas_op.get_H_pinv()
>>> print(H2.shape)
torch.Size([1600, 400])
```

spyrit.core.meas.DynamicHadamSplit.get_P

DynamicHadamSplit.**get_P**() → tensor

Returns the attribute measurement matrix P.

Shape:

Output: $(2M, N)$, where (M, N) is the shape of the measurement matrix H given at initialization.

Example:

```
>>> H = torch.rand([400, 1600])
>>> meas_op = LinearDynamicSplit(H)
>>> P = meas_op.get_P()
>>> print(P.shape)
torch.Size([800, 1600])
```

spyrit.core.meas.DynamicHadamSplit.get_Perm

DynamicHadamSplit.**get_Perm**() → tensor

spyrit.core.meas.DynamicHadamSplit.set_H_pinv

DynamicHadamSplit.**set_H_pinv**(*reg: float = 1e-15*, *pinv: tensor = None*) → None

Stores in self.H_pinv the pseudo inverse of the measurement matrix H.

If `pinv` is given, it is directly stored as the pseudo inverse. The validity of the pseudo inverse is not checked. If `pinv` is `False`, the pseudo inverse is computed from the existing measurement matrix H with regularization parameter `reg`.

Args:

`reg` (float, optional): Cutoff for small singular values.

`H_pinv` (torch.tensor, optional): If given, the tensor is directly stored as the pseudo inverse. No checks are performed. Otherwise, the pseudo inverse is computed from the existing measurement matrix H.

Shape:

`H_pinv`: (N, M), where N is the number of pixels in the image and M the number of measurements.

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> H2 = torch.linalg.pinv(H1)
>>> meas_op = Linear(H1)
>>> meas_op.set_H_pinv(H2)
```

spyrit.core.meas.DynamicHadamSplit.sort_by_indices

`DynamicHadamSplit.`**`sort_by_indices`**(*x: tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → tensor

Reorder the rows or columns of a tensor according to the indices.

The indices are stored in the attribute `self.indices` and are used to reorder the rows or columns of the input tensor x. The indices give the order in which the rows or columns should be reordered.

..note::

This method is identical to the function *sort_by_indices()*.

Args:

x (torch.tensor):
Input tensor to be reordered. The tensor must have the same number of rows or columns as the number of elements in the attribute `self.indices`.

axis (str, optional):
Axis along which to order the tensor. Must be either “rows” or “cols”. Defaults to “rows”.

inverse_permutation (bool, optional): *
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not “rows” or “cols”.

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

torch.tensor:
Tensor x with reordered rows or columns according to the indices.

spyrit.core.meas.DynamicLinear

class spyrit.core.meas.**DynamicLinear**(*H: tensor*, *Ord: tensor = None*)

Bases: Module

Simulates the measurement of a moving object using a measurement matrix.

Computes linear measurements y from incoming images: $y = Hx$, where H is a linear operator (matrix) and x is a batch of vectorized images representing a motion picture.

The class is constructed from a matrix H of shape (M, N), where N represents the number of pixels in the image and M the number of measurements and the number of frames in the animated object.

> **Warning:** For each call, there must be **exactly** as many images in x as there are measurements in the linear operator used to initialize the class.

Args:

H (torch.tensor): measurement matrix (linear operator) with shape (M, N).

Attributes:

H (torch.nn.Parameter): The learnable measurement matrix of shape (M, N) initialized as H.

M (int): Number of measurements performed by the linear operator. It is initialized as the first dimension of H.

N (int): Number of pixels in the image. It is initialized as the second dimension of H.

h (int): Image height h. The image is assumed to be square, i.e. $h = \text{floor}(\sqrt{N})$. If not, please assign h and w manually.

w (int): Image width w. The image is assumed to be square, i.e. $w = \text{floor}(\sqrt{N})$. If not, please assign h and w manually.

Example:

```
>>> H = torch.rand([400, 1600])
>>> meas_op = DynamicLinear(H)
>>> print(meas_op)
DynamicLinear(
  (Image pixels): 1600
  (H): torch.Size([400, 1600])
  )
```

Methods

forward(x)	Simulates the measurement of a motion picture.
get_H()	Returns the attribute measurement matrix H.
get_H_pinv()	Returns the pseudo inverse of the measurement matrix H.
set_H_pinv([reg, pinv])	Stores in self.H_pinv the pseudo inverse of the measurement matrix H.
sort_by_indices(x[, axis, inverse_permutation])	Reorder the rows or columns of a tensor according to the indices.

spyrit.core.meas.DynamicLinear.forward

DynamicLinear.**forward**(*x: tensor*) → tensor

Simulates the measurement of a motion picture.

The output y is computed as $y = Hx$, where H is the measurement matrix and x is a batch of vectorized (flattened) images.

> **Warning:** There must be **exactly** as many images as there are measurements in the linear operator used to initialize the class, i.e. *H.shape[-2] == x.shape[-2]*

Args:

x: Batch of vectorized (flattened) images.

Shape:

x: $(*, M, N)$, where * denotes the batch size and (M, N) is the shape of the measurement matrix H. M is the number of measurements (and frames) and N the number of pixels in the image.

output: $(*, M)$

Example:

```
>>> x = torch.rand([10, 400, 1600])
>>> H = torch.rand([400, 1600])
>>> meas_op = DynamicLinear(H)
>>> y = meas_op(x)
>>> print(y.shape)
torch.Size([10, 400])
```

spyrit.core.meas.DynamicLinear.get_H

DynamicLinear.**get_H**() → tensor

Returns the attribute measurement matrix H.

Shape:

Output: (M, N)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1)
>>> H2 = meas_op.get_H()
>>> print(H2.shape)
torch.Size([400, 1600])
```

spyrit.core.meas.DynamicLinear.get_H_pinv

DynamicLinear.**get_H_pinv**() → tensor

Returns the pseudo inverse of the measurement matrix H.

Shape:
Output: (N, M)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1, True)
>>> H2 = meas_op.get_H_pinv()
>>> print(H2.shape)
torch.Size([1600, 400])
```

spyrit.core.meas.DynamicLinear.set_H_pinv

DynamicLinear.**set_H_pinv**(*reg: float = 1e-15*, *pinv: tensor = None*) → None

Stores in self.H_pinv the pseudo inverse of the measurement matrix H.

If `pinv` is given, it is directly stored as the pseudo inverse. The validity of the pseudo inverse is not checked. If `pinv` is `False`, the pseudo inverse is computed from the existing measurement matrix H with regularization parameter `reg`.

Args:
`reg` (float, optional): Cutoff for small singular values.

`H_pinv` (torch.tensor, optional): If given, the tensor is directly stored as the pseudo inverse. No checks are performed. Otherwise, the pseudo inverse is computed from the existing measurement matrix H.

Shape:
`H_pinv`: (N, M), where N is the number of pixels in the image and M the number of measurements.

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> H2 = torch.linalg.pinv(H1)
>>> meas_op = Linear(H1)
>>> meas_op.set_H_pinv(H2)
```

spyrit.core.meas.DynamicLinear.sort_by_indices

DynamicLinear.**sort_by_indices**(*x: tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → tensor

Reorder the rows or columns of a tensor according to the indices.

The indices are stored in the attribute `self.indices` and are used to reorder the rows or columns of the input tensor x. The indices give the order in which the rows or columns should be reordered.

..note::
This method is identical to the function *sort_by_indices()*.

Args:

x (torch.tensor):
Input tensor to be reordered. The tensor must have the same number of rows or columns as the number of elements in the attribute `self.indices`.

axis (str, optional):
Axis along which to order the tensor. Must be either "rows" or "cols". Defaults to "rows".

inverse_permutation (bool, optional): *
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not "rows" or "cols".

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

torch.tensor:
Tensor x with reordered rows or columns according to the indices.

spyrit.core.meas.DynamicLinearSplit

class spyrit.core.meas.**DynamicLinearSplit**(*H: tensor*, *Ord: tensor = None*)

Bases: *DynamicLinear*

Simulates the measurement of a moving object using the positive and negative components of the measurement matrix.

Computes linear measurements y from incoming images: $y = Px$, where P is a linear operator (matrix) and x is a batch of vectorized images representing a motion picture.

The matrix P contains only positive values and is obtained by splitting a given measurement matrix H such that $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$, where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

The class is constructed from the M by N matrix H, where N represents the number of pixels in the image and M the number of measurements.

Args:
H (torch.tensor): measurement matrix (linear operator) with shape (M, N) where M is the number of measurements and N the number of pixels in the image.

Attributes:
`H` (torch.nn.Parameter): The learnable measurement matrix of shape (M, N).

`P` (torch.nn.Parameter): The splitted measurement matrix of shape $(2M, N)$ initialized as $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$ where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$

`M` (int): Number of measurements performed by the linear operator. It is initialized as the first dimension of H.

`N` (int): Number of pixels in the image. It is initialized as the second dimension of H.

`h` (int): Image height h. The image is assumed to be square, i.e. $h = \text{floor}(\sqrt{N})$. If not, please assign `h` and `w` manually.

`w` (int): Image width w. The image is assumed to be square, i.e. $w = \text{floor}(\sqrt{N})$. If not, please assign `h` and `w` manually.

Warning: For each call, there must be **exactly** as many images in x as there are measurements in the linear operator used to initialize the class.

Example:

```
>>> H = torch.rand([400,1600])
>>> meas_op = DynamicLinearSplit(H)
>>> print(meas_op)
DynamicLinearSplit(
    (Image pixels): 1600
    (H): torch.Size([400, 1600])
    (P): torch.Size([800, 1600])
    )
```

Methods

forward(x)	Simulates the measurement of a motion picture using P.
forward_H(x)	Simulates the measurement of a motion picture using H.
get_H()	Returns the attribute measurement matrix H.
get_H_pinv()	Returns the pseudo inverse of the measurement matrix H.
get_P()	Returns the attribute measurement matrix P.
set_H_pinv([reg, pinv])	Stores in self.H_pinv the pseudo inverse of the measurement matrix H.
sort_by_indices(x[, axis, inverse_permutation])	Reorder the rows or columns of a tensor according to the indices.

spyrit.core.meas.DynamicLinearSplit.forward

DynamicLinearSplit.**forward**(*x: tensor*) → tensor

Simulates the measurement of a motion picture using P.

The output y is computed as $y = Px$, where P is the measurement matrix and x is a batch of vectorized (flattened) images.

P contains only positive values and is obtained by splitting a given measurement matrix H such that $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$, where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

The matrix H can contain positive and negative values and is given by the user at initialization.

Warning: There must be **exactly** as many images as there are measurements in the linear operator used to initialize the class, i.e. *P.shape[-2] == x.shape[-2]*

Args:

x: Batch of vectorized (flattened) images of shape $(*, 2M, N)$ where * denotes the batch size, $2M$ the number of measurements in the measurement matrix P and N the number of pixels in the image.

Shape:

x: $(*, 2M, N)$

P has a shape of $(2M, N)$ where M is the number of measurements as defined by the first dimension of H and N is the number of pixels in the image.

output: $(*, 2M)$

Example:

```
>>> x = torch.rand([10, 800, 1600])
>>> H = torch.rand([400, 1600])
>>> meas_op = DynamicLinearSplit(H)
>>> y = meas_op(x)
>>> print(y.shape)
torch.Size([10, 800])
```

spyrit.core.meas.DynamicLinearSplit.forward_H

DynamicLinearSplit.**forward_H**(*x: tensor*) → tensor

Simulates the measurement of a motion picture using H.

The output y is computed as $y = Hx$, where H is the measurement matrix and x is a batch of vectorized (flattened) images. The positive and negative components of the measurement matrix are **not** used in this method.

The matrix H can contain positive and negative values and is given by the user at initialization.

> **Warning:** There must be **exactly** as many images as there are measurements in the linear operator used to initialize the class, i.e. *H.shape[-2:] == x.shape[-2:]*

Args:

x: Batch of vectorized (flatten) images of shape $(*, M, N)$ where * denotes the batch size, and (M, N) is the shape of the measurement matrix H.

Shape:

x: $(*, M, N)$

H has a shape of (M, N) where M is the number of measurements and N is the number of pixels in the image.

output: $(*, M)$

Example:

```
>>> x = torch.rand([10, 400, 1600])
>>> H = torch.rand([400, 1600])
>>> meas_op = LinearDynamicSplit(H)
>>> y = meas_op.forward_H(x)
>>> print(y.shape)
torch.Size([10, 400])
```

spyrit.core.meas.DynamicLinearSplit.get_H

DynamicLinearSplit.**get_H**() → tensor

Returns the attribute measurement matrix H.

Shape:

Output: (M, N)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1)
>>> H2 = meas_op.get_H()
>>> print(H2.shape)
torch.Size([400, 1600])
```

spyrit.core.meas.DynamicLinearSplit.get_H_pinv

DynamicLinearSplit.**get_H_pinv**() → tensor

Returns the pseudo inverse of the measurement matrix H.

Shape:

Output: (N, M)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1, True)
>>> H2 = meas_op.get_H_pinv()
>>> print(H2.shape)
torch.Size([1600, 400])
```

spyrit.core.meas.DynamicLinearSplit.get_P

DynamicLinearSplit.**get_P**() → tensor

Returns the attribute measurement matrix P.

Shape:

Output: $(2M, N)$, where (M, N) is the shape of the measurement matrix H given at initialization.

Example:

```
>>> H = torch.rand([400, 1600])
>>> meas_op = LinearDynamicSplit(H)
>>> P = meas_op.get_P()
>>> print(P.shape)
torch.Size([800, 1600])
```

spyrit.core.meas.DynamicLinearSplit.set_H_pinv

DynamicLinearSplit.**set_H_pinv**(*reg: float = 1e-15*, *pinv: tensor = None*) → None

Stores in self.H_pinv the pseudo inverse of the measurement matrix H.

If `pinv` is given, it is directly stored as the pseudo inverse. The validity of the pseudo inverse is not checked. If `pinv` is `False`, the pseudo inverse is computed from the existing measurement matrix H with regularization parameter `reg`.

Args:

`reg` (float, optional): Cutoff for small singular values.

`H_pinv` (torch.tensor, optional): If given, the tensor is directly stored as the pseudo inverse. No checks are performed. Otherwise, the pseudo inverse is computed from the existing measurement matrix H.

Shape:

`H_pinv`: (N, M), where N is the number of pixels in the image and M the number of measurements.

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> H2 = torch.linalg.pinv(H1)
>>> meas_op = Linear(H1)
>>> meas_op.set_H_pinv(H2)
```

spyrit.core.meas.DynamicLinearSplit.sort_by_indices

DynamicLinearSplit.**sort_by_indices**(*x: tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → tensor

Reorder the rows or columns of a tensor according to the indices.

The indices are stored in the attribute `self.indices` and are used to reorder the rows or columns of the input tensor x. The indices give the order in which the rows or columns should be reordered.

..note::

This method is identical to the function *sort_by_indices()*.

Args:

x (torch.tensor):
Input tensor to be reordered. The tensor must have the same number of rows or columns as the number of elements in the attribute `self.indices`.

axis (str, optional):
Axis along which to order the tensor. Must be either “rows” or “cols”. Defaults to “rows”.

inverse_permutation (bool, optional): *
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not “rows” or “cols”.

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

torch.tensor:
Tensor x with reordered rows or columns according to the indices.

spyrit.core.meas.HadamSplit

class spyrit.core.meas.**HadamSplit**(*M: int*, *h: int*, *Ord: tensor*)

Bases: *LinearSplit*, *DynamicHadamSplit*

Simulates the measurement of a still image using the positive and negative components of a Hadamard matrix.

Computes linear measurements from incoming images: $y = Px$, where P is a linear operator (matrix) with positive entries and x is a vectorized image or a batch of images.

The class relies on a Hadamard-based matrix H with shape (M, N) where N represents the number of pixels in the image and $M \leq N$ the number of measurements. H is obtained by selecting a re-ordered subsample of M rows of a "full" Hadamard matrix F with shape (N^2, N^2). N must be a power of 2.

The matrix P is then obtained by splitting the matrix H such that $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$, where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

Args:
`M` (int): Number of measurements

`h` (int): Image height h, must be a power of 2. The image is assumed to be square, so the number of pixels in the image is $N = h^2$.

`Ord` (torch.tensor): Order matrix with shape (h, h) used to compute the permutation matrix G^T with shape (N, N) (see the *sampling* submodule)

Attributes:
`H` (torch.nn.Parameter): The measurement matrix of shape (M, h^2). It is initialized as a re-ordered subsample of the rows of the "full" Hadamard matrix F with shape (N^2, N^2).

`H_pinv` (torch.nn.Parameter): The pseudo inverse of the measurement matrix of shape (h^2, M). It is initialized as $H^\dagger = \frac{1}{N} H^T$ where $N = h^2$.

`P` (torch.nn.Parameter): The splitted measurement matrix of shape $(2M, h^2)$ initialized as $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$ where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

`Perm` (torch.nn.Parameter): The permutation matrix G^T that is used to re-order the subsample of rows of the "full" Hadamard matrix F according to descreasing value of the order matrix Ord. It has shape (N, N) where $N = h^2$.

`M` (int): Number of measurements performed by the linear operator.

`N` (int): Number of pixels in the image. It is initialized as h^2.

`h` (int): Image height h.

`w` (int): Image width w. The image is assumed to be square, i.e. $w = h$.

Note: The computation of a Hadamard transform Fx benefits a fast algorithm, as well as the computation of inverse Hadamard transforms.

Note: The matrix H has shape (M, N) with $N = h^2$.

Note: $H = H_+ - H_-$

Example:

```
>>> h = 32
>>> Ord = torch.randn(h, h)
>>> meas_op = HadamSplit(400, h, Ord)
>>> print(meas_op)
HadamSplit(
  (Image pixels): 1024
  (H): torch.Size([400, 1024])
  (P): torch.Size([800, 1024])
  (Perm): torch.Size([1024, 1024])
  (H_pinv): torch.Size([1024, 400])
  )
```

Methods

adjoint(x)	Applies adjoint transform to incoming measurements $y = H^T x$
forward(x)	Applies linear transform to incoming images: $y = Px$.
forward_H(x)	Applies linear transform to incoming images: $m = Hx$.
get_H()	Returns the attribute measurement matrix H.
get_H_T()	Returns the transpose of the measurement matrix H.
get_H_pinv()	Returns the pseudo inverse of the measurement matrix H.
get_P()	Returns the attribute measurement matrix P.
get_Perm()	
inverse(x)	Inverse transform of Hadamard-domain images $x = H_{had}^{-1} Gy$ is a Hadamard matrix.
pinv(x)	Computes the pseudo inverse solution $y = H^{\dagger} x$
set_H_pinv([reg, pinv])	Stores in self.H_pinv the pseudo inverse of the measurement matrix H.
sort_by_indices(x[, axis, inverse_permutation])	Reorder the rows or columns of a tensor according to the indices.

spyrit.core.meas.HadamSplit.adjoint

HadamSplit.**adjoint**(*x: tensor*) → tensor

Applies adjoint transform to incoming measurements $y = H^T x$

Args:

x (torch.tensor): batch of measurement vectors. If x has more than 1 dimension, the adjoint measurement is applied to each measurement in the batch.

Shape:

x: $(*, M)$

Output: $(*, N)$

Example:

```
>>> H = torch.randn([400, 1600])
>>> meas_op = Linear(H)
>>> x = torch.randn([10, 400]
>>> y = meas_op.adjoint(x)
>>> print(y.shape)
torch.Size([10, 1600])
```

spyrit.core.meas.HadamSplit.forward

HadamSplit.**forward**(*x: tensor*) → tensor

Applies linear transform to incoming images: $y = Px$.

This method uses the splitted measurement matrix P to compute the linear measurements from incoming images. P contains only positive values and is obtained by splitting a given measurement matrix H such that $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$, where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

Args:

x (torch.tensor): Batch of vectorized (flattened) images. If x has more than 1 dimension, the linear measurement is applied to each image in the batch.

Shape:

x: $(*, N)$ where * denotes the batch size and *N* the total number of pixels in the image.

Output: $(*, 2M)$ where * denotes the batch size and *M* the number of measurements.

Example:

```
>>> H = torch.randn(400, 1600)
>>> meas_op = LinearSplit(H)
>>> x = torch.randn(10, 1600)
>>> y = meas_op(x)
>>> print(y.shape)
torch.Size([10, 800])
```

spyrit.core.meas.HadamSplit.forward_H

HadamSplit.**forward_H**(*x: tensor*) → tensor

Applies linear transform to incoming images: $m = Hx$.

This method uses the measurement matrix H to compute the linear measurements from incoming images.

Args:

`x` (torch.tensor): Batch of vectorized (flatten) images. If x has more than 1 dimension, the linear measurement is applied to each image in the batch.

Shape:

`x`: $(*, N)$ where * denotes the batch size and *N* the total number of pixels in the image.

Output: $(*, M)$ where * denotes the batch size and *M* the number of measurements.

Example:

```
>>> H = torch.randn(400, 1600)
>>> meas_op = LinearSplit(H)
>>> x = torch.randn(10, 1600)
>>> y = meas_op.forward_H(x)
>>> print(y.shape)
torch.Size([10, 400])
```

spyrit.core.meas.HadamSplit.get_H

HadamSplit.**get_H**() → tensor

Returns the attribute measurement matrix H.

Shape:

Output: (M, N)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1)
>>> H2 = meas_op.get_H()
>>> print(H2.shape)
torch.Size([400, 1600])
```

spyrit.core.meas.HadamSplit.get_H_T

HadamSplit.**get_H_T**() → tensor

Returns the transpose of the measurement matrix H.

Shape:

Output: (N, M), where N is the number of pixels in the image and M the number of measurements.

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1)
>>> H2 = meas_op.get_H_T()
>>> print(H2.shape)
torch.Size([400, 1600])
```

spyrit.core.meas.HadamSplit.get_H_pinv

HadamSplit.**get_H_pinv**() → tensor

Returns the pseudo inverse of the measurement matrix H.

Shape:

Output: (N, M)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1, True)
>>> H2 = meas_op.get_H_pinv()
>>> print(H2.shape)
torch.Size([1600, 400])
```

spyrit.core.meas.HadamSplit.get_P

HadamSplit.**get_P**() → tensor

Returns the attribute measurement matrix P.

Shape:

Output: $(2M, N)$, where (M, N) is the shape of the measurement matrix H given at initialization.

Example:

```
>>> H = torch.rand([400, 1600])
>>> meas_op = LinearDynamicSplit(H)
>>> P = meas_op.get_P()
>>> print(P.shape)
torch.Size([800, 1600])
```

spyrit.core.meas.HadamSplit.get_Perm

HadamSplit.**get_Perm**() → tensor

spyrit.core.meas.HadamSplit.inverse

HadamSplit.**inverse**(*x: tensor*) → tensor

Inverse transform of Hadamard-domain images $x = H_{had}^{-1} G y$ is a Hadamard matrix.

Args:

x: batch of images in the Hadamard domain

Shape:

x: $(b * c, N)$ with b the batch size, c the number of channels, and N the number of pixels in the image.

Output: math:*(b*c, N)*

Example:

```
>>> h = 32
>>> Ord = torch.randn(h, h)
>>> meas_op = HadamSplit(400, h, Ord)
>>> y = torch.randn(10, h**2)
>>> x = meas_op.inverse(y)
>>> print(x.shape)
torch.Size([10, 1024])
```

spyrit.core.meas.HadamSplit.pinv

`HadamSplit.pinv`(*x: tensor*) → tensor

Computes the pseudo inverse solution $y = H^{\dagger}x$

Args:

x (torch.tensor): batch of measurement vectors. If x has more than 1 dimension, the pseudo inverse is applied to each image in the batch.

Shape:

x: $(*, M)$

Output: $(*, N)$

Example:

```
>>> H = torch.randn([400, 1600])
>>> meas_op = Linear(H, True)
>>> x = torch.randn([10, 400])
>>> y = meas_op.pinv(x)
>>> print(y.shape)
torch.Size([10, 1600])
```

spyrit.core.meas.HadamSplit.set_H_pinv

`HadamSplit.set_H_pinv`(*reg: float = 1e-15*, *pinv: tensor = None*) → None

Stores in self.H_pinv the pseudo inverse of the measurement matrix H.

If *pinv* is given, it is directly stored as the pseudo inverse. The validity of the pseudo inverse is not checked. If *pinv* is `False`, the pseudo inverse is computed from the existing measurement matrix H with regularization parameter `reg`.

Args:

`reg` (float, optional): Cutoff for small singular values.

`H_pinv` (torch.tensor, optional): If given, the tensor is directly stored as the pseudo inverse. No checks are performed. Otherwise, the pseudo inverse is computed from the existing measurement matrix H.

Shape:

`H_pinv`: (N, M), where N is the number of pixels in the image and M the number of measurements.

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> H2 = torch.linalg.pinv(H1)
>>> meas_op = Linear(H1)
>>> meas_op.set_H_pinv(H2)
```

spyrit.core.meas.HadamSplit.sort_by_indices

HadamSplit.**sort_by_indices**(*x: tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → tensor

Reorder the rows or columns of a tensor according to the indices.

The indices are stored in the attribute `self.indices` and are used to reorder the rows or columns of the input tensor x. The indices give the order in which the rows or columns should be reordered.

..note::
This method is identical to the function *sort_by_indices()*.

Args:

x (torch.tensor):
Input tensor to be reordered. The tensor must have the same number of rows or columns as the number of elements in the attribute `self.indices`.

axis (str, optional):
Axis along which to order the tensor. Must be either "rows" or "cols". Defaults to "rows".

inverse_permutation (bool, optional): *
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not "rows" or "cols".

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

torch.tensor:
Tensor x with reordered rows or columns according to the indices.

spyrit.core.meas.Linear

class spyrit.core.meas.**Linear**(*H: tensor*, *pinv: bool = False*, *reg: float = 1e-15*, *Ord: tensor = None*)

Bases: *DynamicLinear*

Simulates the measurement of an still image using a measurement matrix.

Computes linear measurements from incoming images: $y = Hx$, where H is a given linear operator (matrix) and x is a vectorized image or batch of images.

The class is constructed from a M by N matrix H, where N represents the number of pixels in the image and M the number of measurements.

Args:
`H` (`torch.tensor`): measurement matrix (linear operator) with shape (M, N).

pinv (bool): Option to have access to pseudo inverse solutions. If *True*, the pseudo inverse is initialized as $H^\dagger$ and stored in the attribute `H_pinv`. Defaults to *False* (the pseudo inverse is not initiliazed).

`reg` (float, optional): Regularization parameter (cutoff for small singular values, see `numpy.linal.pinv`). Only relevant when *pinv* is *True*.

Attributes:
`H` (torch.tensor): The learnable measurement matrix of shape (M, N) initialized as H

`H_pinv` (torch.tensor, optional): The learnable adjoint measurement matrix of shape (N, M) initialized as $H^\dagger$. Only relevant when *pinv* is *True*.

`M` (int): Number of measurements performed by the linear operator. It is initialized as the first dimension of H.

`N` (int): Number of pixels in the image. It is initialized as the second dimension of H.

`h` (int): Image height h. The image is assumed to be square, i.e. $h = \text{floor}(\sqrt{N})$. If not, please assign `h` and `w` manually.

`w` (int): Image width w. The image is assumed to be square, i.e. $w = \text{floor}(\sqrt{N})$. If not, please assign `h` and `w` manually.

Note: If you know the pseudo inverse of H and want to store it, it is best to initialize the class with *pinv* set to *False* and then call *set_H_pinv()* to store the pseudo inverse.

Example 1:

```
>>> H = torch.rand([400, 1600])
>>> meas_op = Linear(H, pinv=False)
>>> print(meas_op)
Linear(
  (Image pixels): 1600
  (H): torch.Size([400, 1600])
  (H_pinv): None
  )
```

Example 2:

```
>>> H = torch.rand([400, 1600])
>>> meas_op = Linear(H, True)
>>> print(meas_op)
Linear(
  (Image pixels): 1600
  (H): torch.Size([400, 1600])
  (H_pinv): torch.Size([1600, 400])
  )
```

Methods

adjoint(x)	Applies adjoint transform to incoming measurements $y = H^T x$
forward(x)	Applies linear transform to incoming images: $y = Hx$.
get_H()	Returns the attribute measurement matrix H.
get_H_T()	Returns the transpose of the measurement matrix H.
get_H_pinv()	Returns the pseudo inverse of the measurement matrix H.
pinv(x)	Computes the pseudo inverse solution $y = H^{\dagger} x$
set_H_pinv([reg, pinv])	Stores in self.H_pinv the pseudo inverse of the measurement matrix H.
sort_by_indices(x[, axis, inverse_permutation])	Reorder the rows or columns of a tensor according to the indices.

spyrit.core.meas.Linear.adjoint

Linear.**adjoint**(*x: tensor*) → tensor

Applies adjoint transform to incoming measurements $y = H^T x$

Args:

x (torch.tensor): batch of measurement vectors. If x has more than 1 dimension, the adjoint measurement is applied to each measurement in the batch.

Shape:

x: $(*, M)$

Output: $(*, N)$

Example:

```
>>> H = torch.randn([400, 1600])
>>> meas_op = Linear(H)
>>> x = torch.randn([10, 400]
>>> y = meas_op.adjoint(x)
>>> print(y.shape)
torch.Size([10, 1600])
```

spyrit.core.meas.Linear.forward

Linear.**forward**(*x: tensor*) → tensor

Applies linear transform to incoming images: $y = Hx$.

Args:

x (torch.tensor): Batch of vectorized (flattened) images. If x has more than 1 dimension, the linear measurement is applied to each image in the batch.

Shape:

x: $(*, N)$ where * denotes the batch size and *N* the total number of pixels in the image.

Output: $(*, M)$ where * denotes the batch size and *M* the number of measurements.

Example:

```
>>> H = torch.randn([400, 1600])
>>> meas_op = Linear(H)
>>> x = torch.randn([10, 1600])
>>> y = meas_op(x)
>>> print(y.shape)
torch.Size([10, 400])
```

spyrit.core.meas.Linear.get_H

`Linear.get_H()` → tensor

Returns the attribute measurement matrix H.

Shape:

Output: (M, N)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1)
>>> H2 = meas_op.get_H()
>>> print(H2.shape)
torch.Size([400, 1600])
```

spyrit.core.meas.Linear.get_H_T

`Linear.get_H_T()` → tensor

Returns the transpose of the measurement matrix H.

Shape:

Output: (N, M), where N is the number of pixels in the image and M the number of measurements.

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1)
>>> H2 = meas_op.get_H_T()
>>> print(H2.shape)
torch.Size([400, 1600])
```

spyrit.core.meas.Linear.get_H_pinv

`Linear.get_H_pinv()` → tensor

Returns the pseudo inverse of the measurement matrix H.

Shape:

Output: (N, M)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1, True)
>>> H2 = meas_op.get_H_pinv()
>>> print(H2.shape)
torch.Size([1600, 400])
```

spyrit.core.meas.Linear.pinv

`Linear.`**`pinv`**(*x: tensor*) → tensor

Computes the pseudo inverse solution $y = H^{\dagger}x$

Args:

x (torch.tensor): batch of measurement vectors. If x has more than 1 dimension, the pseudo inverse is applied to each image in the batch.

Shape:

x: $(*, M)$

Output: $(*, N)$

Example:

```
>>> H = torch.randn([400, 1600])
>>> meas_op = Linear(H, True)
>>> x = torch.randn([10, 400])
>>> y = meas_op.pinv(x)
>>> print(y.shape)
torch.Size([10, 1600])
```

spyrit.core.meas.Linear.set_H_pinv

`Linear.`**`set_H_pinv`**(*reg: float = 1e-15*, *pinv: tensor = None*) → None

Stores in self.H_pinv the pseudo inverse of the measurement matrix H.

If *pinv* is given, it is directly stored as the pseudo inverse. The validity of the pseudo inverse is not checked. If *pinv* is `False`, the pseudo inverse is computed from the existing measurement matrix H with regularization parameter `reg`.

Args:

`reg` (float, optional): Cutoff for small singular values.

`H_pinv` (torch.tensor, optional): If given, the tensor is directly stored as the pseudo inverse. No checks are performed. Otherwise, the pseudo inverse is computed from the existing measurement matrix H.

Shape:

`H_pinv`: (N, M), where N is the number of pixels in the image and M the number of measurements.

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> H2 = torch.linalg.pinv(H1)
>>> meas_op = Linear(H1)
>>> meas_op.set_H_pinv(H2)
```

spyrit.core.meas.Linear.sort_by_indices

Linear.**sort_by_indices**(*x: tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → tensor

Reorder the rows or columns of a tensor according to the indices.

The indices are stored in the attribute `self.indices` and are used to reorder the rows or columns of the input tensor x. The indices give the order in which the rows or columns should be reordered.

..note::
This method is identical to the function *sort_by_indices()*.

Args:

x (torch.tensor):
Input tensor to be reordered. The tensor must have the same number of rows or columns as the number of elements in the attribute `self.indices`.

axis (str, optional):
Axis along which to order the tensor. Must be either "rows" or "cols". Defaults to "rows".

inverse_permutation (bool, optional): *
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not "rows" or "cols".

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

torch.tensor:
Tensor x with reordered rows or columns according to the indices.

spyrit.core.meas.LinearSplit

class spyrit.core.meas.**LinearSplit**(*H: tensor*, *pinv: bool = False*, *reg: float = 1e-15*, *Ord: tensor = None*)

Bases: *Linear*, *DynamicLinearSplit*

Simulates the measurement of a still image using the computed positive and negative components of the measurement matrix.

Computes linear measurements from incoming images: $y = Px$, where P is a linear operator (matrix) and x is a vectorized image or batch of vectorized images.

The matrix P contains only positive values and is obtained by splitting a measurement matrix H such that $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$, where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

The class is constructed from the M by N matrix H, where N represents the number of pixels in the image and M the number of measurements.

Args:
`H` (torch.tensor): measurement matrix (linear operator) with shape (M, N), where M is the number of measurements and N the number of pixels in the image.

pinv (Any): Option to have access to pseudo inverse solutions. If *True*, the pseudo inverse is initialized as $H^\dagger$ and stored in the attribute `H_pinv`. Defaults to *False* (the pseudo inverse is not initiliazed).

`reg` (float, optional): Regularization parameter (cutoff for small singular values, see `torch.linalg.pinv`). Only relevant when *pinv* is *True*.

Attributes:

`H` (torch.nn.Parameter): The learnable measurement matrix of shape (M, N).

`P` (torch.nn.Parameter): The splitted measurement matrix of shape $(2M, N)$ initialized as $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$ where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$

`M` (int): Number of measurements performed by the linear operator. It is initialized as the first dimension of H.

`N` (int): Number of pixels in the image. It is initialized as the second dimension of H.

`h` (int): Image height h. The image is assumed to be square, i.e. $h = \text{floor}(\sqrt{N})$. If not, please assign `h` and `w` manually.

`w` (int): Image width w. The image is assumed to be square, i.e. $w = \text{floor}(\sqrt{N})$. If not, please assign `h` and `w` manually.

Note: If you know the pseudo inverse of H and want to store it, it is best to initialize the class with *pinv* set to *False* and then call *set_H_pinv()* to store the pseudo inverse.

Example:

```
>>> H = torch.randn(400, 1600)
>>> meas_op = LinearSplit(H, False)
>>> print(meas_op)
LinearSplit(
  (Image pixels): 1600
  (H): torch.Size([400, 1600])
  (P): torch.Size([800, 1600])
  (H_pinv): None
  )
```

Methods

adjoint(x)	Applies adjoint transform to incoming measurements $y = H^T x$
forward(x)	Applies linear transform to incoming images: $y = Px$.
forward_H(x)	Applies linear transform to incoming images: $m = Hx$.
get_H()	Returns the attribute measurement matrix H.
get_H_T()	Returns the transpose of the measurement matrix H.
get_H_pinv()	Returns the pseudo inverse of the measurement matrix H.
get_P()	Returns the attribute measurement matrix P.
pinv(x)	Computes the pseudo inverse solution $y = H^\dagger x$
set_H_pinv([reg, pinv])	Stores in self.H_pinv the pseudo inverse of the measurement matrix H.
sort_by_indices(x[, axis, inverse_permutation])	Reorder the rows or columns of a tensor according to the indices.

spyrit.core.meas.LinearSplit.adjoint

LinearSplit.**adjoint**(*x: tensor*) → tensor

Applies adjoint transform to incoming measurements $y = H^T x$

Args:

x (torch.tensor): batch of measurement vectors. If x has more than 1 dimension, the adjoint measurement is applied to each measurement in the batch.

Shape:

x: $(*, M)$

Output: $(*, N)$

Example:

```
>>> H = torch.randn([400, 1600])
>>> meas_op = Linear(H)
>>> x = torch.randn([10, 400]
>>> y = meas_op.adjoint(x)
>>> print(y.shape)
torch.Size([10, 1600])
```

spyrit.core.meas.LinearSplit.forward

LinearSplit.**forward**(*x: tensor*) → tensor

Applies linear transform to incoming images: $y = Px$.

This method uses the splitted measurement matrix P to compute the linear measurements from incoming images. P contains only positive values and is obtained by splitting a given measurement matrix H such that $P = \begin{bmatrix} H_+ \\ H_- \end{bmatrix}$, where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

Args:

x (torch.tensor): Batch of vectorized (flattened) images. If x has more than 1 dimension, the linear measurement is applied to each image in the batch.

Shape:

x: $(*, N)$ where * denotes the batch size and *N* the total number of pixels in the image.

Output: $(*, 2M)$ where * denotes the batch size and *M* the number of measurements.

Example:

```
>>> H = torch.randn(400, 1600)
>>> meas_op = LinearSplit(H)
>>> x = torch.randn(10, 1600)
>>> y = meas_op(x)
>>> print(y.shape)
torch.Size([10, 800])
```

spyrit.core.meas.LinearSplit.forward_H

LinearSplit.**forward_H**(*x: tensor*) → tensor

Applies linear transform to incoming images: $m = Hx$.

This method uses the measurement matrix H to compute the linear measurements from incoming images.

Args:

x (torch.tensor): Batch of vectorized (flatten) images. If x has more than 1 dimension, the linear measurement is applied to each image in the batch.

Shape:

x: $(*, N)$ where * denotes the batch size and *N* the total number of pixels in the image.

Output: $(*, M)$ where * denotes the batch size and *M* the number of measurements.

Example:

```
>>> H = torch.randn(400, 1600)
>>> meas_op = LinearSplit(H)
>>> x = torch.randn(10, 1600)
>>> y = meas_op.forward_H(x)
>>> print(y.shape)
torch.Size([10, 400])
```

spyrit.core.meas.LinearSplit.get_H

LinearSplit.**get_H**() → tensor

Returns the attribute measurement matrix H.

Shape:

Output: (M, N)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1)
```

(continues on next page)

(continued from previous page)

```
>>> H2 = meas_op.get_H()
>>> print(H2.shape)
torch.Size([400, 1600])
```

spyrit.core.meas.LinearSplit.get_H_T

`LinearSplit.get_H_T()` → tensor

Returns the transpose of the measurement matrix H.

Shape:

Output: (N, M), where N is the number of pixels in the image and M the number of measurements.

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1)
>>> H2 = meas_op.get_H_T()
>>> print(H2.shape)
torch.Size([400, 1600])
```

spyrit.core.meas.LinearSplit.get_H_pinv

`LinearSplit.get_H_pinv()` → tensor

Returns the pseudo inverse of the measurement matrix H.

Shape:

Output: (N, M)

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> meas_op = Linear(H1, True)
>>> H2 = meas_op.get_H_pinv()
>>> print(H2.shape)
torch.Size([1600, 400])
```

spyrit.core.meas.LinearSplit.get_P

`LinearSplit.get_P()` → tensor

Returns the attribute measurement matrix P.

Shape:

Output: $(2M, N)$, where (M, N) is the shape of the measurement matrix H given at initialization.

Example:

```
>>> H = torch.rand([400, 1600])
>>> meas_op = LinearDynamicSplit(H)
>>> P = meas_op.get_P()
>>> print(P.shape)
torch.Size([800, 1600])
```

spyrit.core.meas.LinearSplit.pinv

`LinearSplit.`**`pinv`**(*x: tensor*) → tensor

Computes the pseudo inverse solution $y = H^{\dagger}x$

Args:

x (torch.tensor): batch of measurement vectors. If x has more than 1 dimension, the pseudo inverse is applied to each image in the batch.

Shape:

x: $(*, M)$

Output: $(*, N)$

Example:

```
>>> H = torch.randn([400, 1600])
>>> meas_op = Linear(H, True)
>>> x = torch.randn([10, 400])
>>> y = meas_op.pinv(x)
>>> print(y.shape)
torch.Size([10, 1600])
```

spyrit.core.meas.LinearSplit.set_H_pinv

`LinearSplit.`**`set_H_pinv`**(*reg: float = 1e-15*, *pinv: tensor = None*) → None

Stores in self.H_pinv the pseudo inverse of the measurement matrix H.

If *pinv* is given, it is directly stored as the pseudo inverse. The validity of the pseudo inverse is not checked. If *pinv* is `False`, the pseudo inverse is computed from the existing measurement matrix H with regularization parameter `reg`.

Args:

`reg` (float, optional): Cutoff for small singular values.

`H_pinv` (torch.tensor, optional): If given, the tensor is directly stored as the pseudo inverse. No checks are performed. Otherwise, the pseudo inverse is computed from the existing measurement matrix H.

Shape:

`H_pinv`: (N, M), where N is the number of pixels in the image and M the number of measurements.

Example:

```
>>> H1 = torch.rand([400, 1600])
>>> H2 = torch.linalg.pinv(H1)
>>> meas_op = Linear(H1)
>>> meas_op.set_H_pinv(H2)
```

spyrit.core.meas.LinearSplit.sort_by_indices

LinearSplit.**sort_by_indices**(*x: tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → tensor

Reorder the rows or columns of a tensor according to the indices.

The indices are stored in the attribute `self.indices` and are used to reorder the rows or columns of the input tensor x. The indices give the order in which the rows or columns should be reordered.

..note::
This method is identical to the function *sort_by_indices()*.

Args:

x (torch.tensor):
Input tensor to be reordered. The tensor must have the same number of rows or columns as the number of elements in the attribute `self.indices`.

axis (str, optional):
Axis along which to order the tensor. Must be either "rows" or "cols". Defaults to "rows".

inverse_permutation (bool, optional): *
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not "rows" or "cols".

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

torch.tensor:
Tensor x with reordered rows or columns according to the indices.

spyrit.core.nnet

Neural network models for image denoising.

Classes

ConvNet()	
ConvNetBN()	
DConvNet()	
Identity()	
List_denoi(Denoi, n_denoi)	
Unet([in_channel, out_channel])	

spyrit.core.nnet.ConvNet

class spyrit.core.nnet.**ConvNet**

Bases: Module

Methods

forward(x)	Define the computation performed at every call.

spyrit.core.nnet.ConvNet.forward

ConvNet.**forward**(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

spyrit.core.nnet.ConvNetBN

class spyrit.core.nnet.**ConvNetBN**

Bases: Module

Methods

forward(x)	Define the computation performed at every call.

spyrit.core.nnet.ConvNetBN.forward

ConvNetBN.**forward**(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

spyrit.core.nnet.DConvNet

class spyrit.core.nnet.**DConvNet**

Bases: Module

Methods

forward(x)	Define the computation performed at every call.

spyrit.core.nnet.DConvNet.forward

DConvNet.**forward**(x)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

spyrit.core.nnet.Identity

class spyrit.core.nnet.**Identity**

Bases: Module

Methods

forward(x)	Define the computation performed at every call.

spyrit.core.nnet.Identity.forward

Identity.**forward**(x)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

spyrit.core.nnet.List_denoi

class spyrit.core.nnet.**List_denoi**(*Denoi*, *n_denoi*)

Bases: Module

Methods

forward(x, iterate)	Define the computation performed at every call.

spyrit.core.nnet.List_denoi.forward

List_denoi.**forward**(*x*, *iterate*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

spyrit.core.nnet.Unet

class spyrit.core.nnet.**Unet**(*in_channel=1*, *out_channel=1*)

Bases: Module

Methods

bottle_neck(in_channels[, kernel_size, padding])	
concat(upsampled, bypass)	
contract(in_channels, out_channels[, ...])	
expans(in_channels, mid_channel, out_channels)	
final_block(in_channels, mid_channel, ...[, ...])	
forward(x)	Define the computation performed at every call.

spyrit.core.nnet.Unet.bottle_neck

Unet.**bottle_neck**(*in_channels*, *kernel_size=3*, *padding=1*)

spyrit.core.nnet.Unet.concat

Unet.**concat**(*upsampled*, *bypass*)

spyrit.core.nnet.Unet.contract

Unet.**contract**(*in_channels*, *out_channels*, *kernel_size=3*, *padding=1*)

spyrit.core.nnet.Unet.expans

Unet.**expans**(*in_channels*, *mid_channel*, *out_channels*, *kernel_size=3*, *padding=1*)

spyrit.core.nnet.Unet.final_block

Unet.**final_block**(*in_channels*, *mid_channel*, *out_channels*, *kernel_size=3*)

spyrit.core.nnet.Unet.forward

Unet.**forward**(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

spyrit.core.noise

Noise models for simulating measurements in imaging.

There are four classes in this module, that each simulate a different type of noise in the measurements. The classes simulate the following types of noise:

- NoNoise: Simulates measurements with no noise
- **Poisson: Simulates measurements corrupted by Poisson noise (each pixel** receives a number of photons that follows a Poisson distribution)
- **PoissonApproxGauss: Simulates measurements corrupted by Poisson noise, but** approximates the Poisson distribution with a Gaussian distribution
- **PoissonApproxGaussSameNoise: Simulates measurements corrupted by Poisson** noise, but all measurements in a batch are corrupted with the same noise sample (approximated by a Gaussian distribution)

Classes

NoNoise(meas_op)	Simulates measurements from images in the range [0;1] by computing $y = \frac{1}{2}H(1 + x)$.
Poisson(meas_op[, alpha])	Simulates measurements corrupted by Poisson noise
PoissonApproxGauss(meas_op, alpha)	Simulates measurements corrupted by Poisson noise.
PoissonApproxGaussSameNoise(meas_op, alpha)	Simulates measurements corrupted by Poisson noise.

spyrit.core.noise.NoNoise

class spyrit.core.noise.**NoNoise**(*meas_op:* Linear | LinearSplit | HadamSplit)

Bases: Module

Simulates measurements from images in the range [0;1] by computing $y = \frac{1}{2}H(1 + x)$.

Note: Assumes that the incoming images x are in the range [-1;1]

The class is constructed from a measurement operator (see the *meas* submodule)

Args:

meas_op : Measurement operator (see the *meas* submodule)

Example 1: Using a *Linear* measurement operator

```
>>> H = torch.rand([400,32*32])
>>> linear_op = Linear(H)
>>> linear_acq = NoNoise(linear_op)
```

Example 2: Using a *HadamSplit* measurement operator

```
>>> H = torch.rand([400,32*32])
>>> Perm = torch.rand([32*32,32*32])
>>> split_op = HadamSplit(H, Perm, 32, 32)
>>> split_acq = NoNoise(split_op)
```

Methods

forward(x)	Simulates measurements
sort_by_indices(x[, axis, inverse_permutation])	Reorder the rows or columns of a tensor according to the indices.

spyrit.core.noise.NoNoise.forward

`NoNoise.forward`(*x: tensor*) → tensor

Simulates measurements

Args:
`x`: Batch of images

Shape:

- `x`: $(*, N)$
- `Output`: $(*, M)$

Example 1: Using a *Linear* measurement operator

```
>>> x = torch.FloatTensor(10, 32*32).uniform_(-1, 1)
>>> y = linear_acq(x)
>>> print(y.shape)
torch.Size([10, 400])
```

Example 2: Using a *HadamSplit* measurement operator

```
>>> x = torch.FloatTensor(10, 32*32).uniform_(-1, 1)
>>> y = split_acq(x)
>>> print(y.shape)
torch.Size([10, 800])
```

spyrit.core.noise.NoNoise.sort_by_indices

`NoNoise.sort_by_indices`(*x: tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → tensor

Reorder the rows or columns of a tensor according to the indices.

The indices are stored in the attribute `self.meas_op.indices` and are used to reorder the rows or columns of the input tensor x. The indices give the order in which the rows or columns should be reordered.

..note::
This method is identical to the function *sort_by_indices()*.

Args:

x (torch.tensor):
Input tensor to be reordered. The tensor must have the same number of rows or columns as the number of elements in the attribute `self.indices`.

axis (str, optional):
Axis along which to order the tensor. Must be either “rows” or “cols”. Defaults to “rows”.

inverse_permutation (bool, optional): *
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not “rows” or “cols”.

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

torch.tensor:
Tensor x with reordered rows or columns according to the indices.

spyrit.core.noise.Poisson

class spyrit.core.noise.**Poisson**(*meas_op:* Linear | LinearSplit | HadamSplit, *alpha=50.0*)

Bases: *NoNoise*

Simulates measurements corrupted by Poisson noise

Assuming incoming images x in the range [-1;1], measurements are first simulated for images in the range [0; α]. Then, Poisson noise is applied: $y = \mathcal{P}(\frac{\alpha}{2} H(1+x))$.

Note: Assumes that the incoming images x are in the range [-1;1]

The class is constructed from a measurement operator and an image intensity α that controls the noise level.

Args:
`meas_op`: Measurement operator H (see the *meas* submodule)

`alpha` (float): Image intensity (in photoelectrons)

Example 1: Using a *Linear* measurement operator

```
>>> H = torch.rand([400,32*32])
>>> linear_op = Linear(H)
>>> linear_acq = Poisson(linear_op, 10.0)
```

Example 2: Using a *HadamSplit* measurement operator

```
>>> H = torch.rand([400,32*32])
>>> Perm = torch.rand([32*32,32*32])
>>> split_op = HadamSplit(H, Perm, 32, 32)
>>> split_acq = Poisson(split_op, 200.0)
```

Example 3: Using a *LinearSplit* measurement operator

```
>>> H = torch.rand(24,64)
>>> split_row_op = LinearSplit(H)
>>> split_acq = Poisson(split_row_op, 50.0)
```

Methods

forward(x)	Simulates measurements corrupted by Poisson noise
sort_by_indices(x[, axis, inverse_permutation])	Reorder the rows or columns of a tensor according to the indices.

spyrit.core.noise.Poisson.forward

`Poisson.`**`forward`**(x)

Simulates measurements corrupted by Poisson noise

Args:

`x`: Batch of images

Shape:

- `x`: $(*, N)$
- `Output`: $(*, M)$

Example 1: Two noisy measurement vectors from a *`Linear`* measurement operator

```
>>> H = torch.rand([400,32*32])
>>> meas_op = Linear(H)
>>> noise_op = Poisson(meas_op, 10.0)
>>> x = torch.FloatTensor(10, 32*32).uniform_(-1, 1)
>>> y = noise_op(x)
>>> print(y.shape)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
>>> y = noise_op(x)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
torch.Size([10, 400])
Measurements in (2249.00 , 2896.00)
Measurements in (2237.00 , 2880.00)
```

Example 2: Two noisy measurement vectors from a *`HadamSplit`* operator

```
>>> Perm = torch.rand([32*32,32*32])
>>> meas_op = HadamSplit(H, Perm, 32, 32)
>>> noise_op = Poisson(meas_op, 200.0)
>>> x = torch.FloatTensor(10, 32*32).uniform_(-1, 1)
>>> y = noise_op(x)
>>> print(y.shape)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
>>> y = noise_op(x)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
torch.Size([10, 800])
Measurements in (0.00 , 55338.00)
Measurements in (0.00 , 55077.00)
```

Example 3: Two noisy measurement vectors from a *`LinearSplit`* operator

```
>>> H = torch.rand(24,64)
>>> meas_op = LinearSplit(H)
>>> noise_op = Poisson(meas_op, 50.0)
>>> x = torch.FloatTensor(10, 64, 92).uniform_(-1, 1)
>>> y = noise_op(x)
>>> print(y.shape)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
>>> y = noise_op(x)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
torch.Size([10, 48, 92])
```

(continues on next page)

(continued from previous page)

```
Measurements in (500.00 , 1134.00)
Measurements in (465.00 , 1140.00)
```

spyrit.core.noise.Poisson.sort_by_indices

Poisson.**sort_by_indices**(*x: tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → tensor

Reorder the rows or columns of a tensor according to the indices.

The indices are stored in the attribute `self.meas_op.indices` and are used to reorder the rows or columns of the input tensor x. The indices give the order in which the rows or columns should be reordered.

..note::
This method is identical to the function *sort_by_indices()*.

Args:

x (torch.tensor):
Input tensor to be reordered. The tensor must have the same number of rows or columns as the number of elements in the attribute `self.indices`.

axis (str, optional):
Axis along which to order the tensor. Must be either "rows" or "cols". Defaults to "rows".

inverse_permutation (bool, optional): *
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not "rows" or "cols".

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

torch.tensor:
Tensor x with reordered rows or columns according to the indices.

spyrit.core.noise.PoissonApproxGauss

class spyrit.core.noise.**PoissonApproxGauss**(*meas_op:* Linear | LinearSplit | HadamSplit, *alpha: float*)

Bases: *NoNoise*

Simulates measurements corrupted by Poisson noise. To accelerate the computation, we consider a Gaussian approximation to the Poisson distribution.

Assuming incoming images x in the range [-1;1], measurements are first simulated for images in the range [0; α]: $y = \frac{\alpha}{2} P(1 + x)$. Then, Gaussian noise is added: $y + \sqrt{y} \cdot \mathcal{G}(\mu = 0, \sigma^2 = 1)$.

The class is constructed from a measurement operator P and an image intensity α that controls the noise level.

> **Warning:** Assumes that the incoming images x are in the range [-1;1]

Args:

`meas_op`: Measurement operator H (see the *meas* submodule)

`alpha` (float): Image intensity (in photoelectrons)

Example 1: Using a *Linear* measurement operator

```
>>> H = torch.rand([400,32*32])
>>> meas_op = Linear(H)
>>> noise_op = PoissonApproxGauss(meas_op, 10.0)
```

Example 2: Using a *HadamSplit* operator

```
>>> Perm = torch.rand([32*32,32*32])
>>> meas_op = HadamSplit(H, Perm, 32, 32)
>>> noise_op = PoissonApproxGauss(meas_op, 200.0)
```

Example 3: Using a *LinearSplit* operator

```
>>> H = torch.rand(24,64)
>>> meas_op = LinearSplit(H)
>>> noise_op = PoissonApproxGauss(meas_op, 50.0)
```

Methods

forward(x)	Simulates measurements corrupted by Poisson noise
sort_by_indices(x[, axis, inverse_permutation])	Reorder the rows or columns of a tensor according to the indices.

spyrit.core.noise.PoissonApproxGauss.forward

`PoissonApproxGauss.`**`forward`**(*x: tensor*) → tensor

Simulates measurements corrupted by Poisson noise

Args:

`x`: Batch of images

Shape:

- `x`: $(*, N)$
- `Output`: $(*, M)$

Example 1: Two noisy measurement vectors from a *Linear* measurement operator

```
>>> H = torch.rand([400,32*32])
>>> meas_op = Linear(H)
>>> noise_op = PoissonApproxGauss(meas_op, 10.0)
>>> x = torch.FloatTensor(10, 32*32).uniform_(-1, 1)
>>> y = noise_op(x)
>>> print(y.shape)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
>>> y = noise_op(x)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
```

(continues on next page)

(continued from previous page)

```
torch.Size([10, 400])
Measurements in (2255.57 , 2911.18)
Measurements in (2226.49 , 2934.42)
```

Example 2: Two noisy measurement vectors from a *HadamSplit* operator

```
>>> Perm = torch.rand([32*32,32*32])
>>> meas_op = HadamSplit(H, Perm, 32, 32)
>>> noise_op = PoissonApproxGauss(meas_op, 200.0)
>>> x = torch.FloatTensor(10, 32*32).uniform_(-1, 1)
>>> y = noise_op(x)
>>> print(y.shape)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
>>> y = noise_op(x)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
torch.Size([10, 800])
Measurements in (0.00 , 55951.41)
Measurements in (0.00 , 56216.86)
```

Example 3: Two noisy measurement vectors from a *LinearSplit* operator

```
>>> H = torch.rand(24,64)
>>> meas_op = LinearSplit(H)
>>> noise_op = PoissonApproxGauss(meas_op, 50.0)
>>> x = torch.FloatTensor(10, 64, 92).uniform_(-1, 1)
>>> y = noise_op(x)
>>> print(y.shape)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
>>> y = noise_op(x)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
torch.Size([10, 48, 92])
Measurements in (460.43 , 1216.94)
Measurements in (441.85 , 1230.43)
```

spyrit.core.noise.PoissonApproxGauss.sort_by_indices

PoissonApproxGauss.**sort_by_indices**(*x: tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → tensor

Reorder the rows or columns of a tensor according to the indices.

The indices are stored in the attribute `self.meas_op.indices` and are used to reorder the rows or columns of the input tensor x. The indices give the order in which the rows or columns should be reordered.

..note::
This method is identical to the function *sort_by_indices()*.

Args:

x (torch.tensor):
Input tensor to be reordered. The tensor must have the same number of rows or columns as the number of elements in the attribute `self.indices`.

axis (str, optional):
Axis along which to order the tensor. Must be either “rows” or “cols”. Defaults to “rows”.

inverse_permutation (bool, optional): *
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not “rows” or “cols”.

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

torch.tensor:
Tensor x with reordered rows or columns according to the indices.

spyrit.core.noise.PoissonApproxGaussSameNoise

`class spyrit.core.noise.`**`PoissonApproxGaussSameNoise`**(*meas_op:* Linear | LinearSplit | HadamSplit, *alpha: float*)

Bases: *NoNoise*

Simulates measurements corrupted by Poisson noise. To accelerate the computation, we consider a Gaussian approximation to the Poisson distribution. Contrary to *PoissonApproxGauss*, all measurements in a batch are corrupted with the same noise sample.

Assuming incoming images x in the range [-1;1], measurements are first simulated for images in the range [0; α]: $y = \frac{\alpha}{2} P(1+x)$. Then, Gaussian noise is added: $y + \sqrt{y} \cdot \mathcal{G}(\mu = 0, \sigma^2 = 1)$.

The class is constructed from a measurement operator P and an image intensity α that controls the noise level.

Warning: Assumes that the incoming images x are in the range [-1;1]

Args:
`meas_op`: Measurement operator H (see the *meas* submodule)

`alpha` (float): Image intensity (in photoelectrons)

Example 1: Using a *Linear* measurement operator

```
>>> H = torch.rand([400,32*32])
>>> meas_op = Linear(H)
>>> noise_op = PoissonApproxGaussSameNoise(meas_op, 10.0)
```

Example 2: Using a *HadamSplit* operator

```
>>> Perm = torch.rand([32*32,32*32])
>>> meas_op = HadamSplit(H, Perm, 32, 32)
>>> noise_op = PoissonApproxGaussSameNoise(meas_op, 200.0)
```

Methods

forward(x)	Simulates measurements corrupted by Poisson noise
sort_by_indices(x[, axis, inverse_permutation])	Reorder the rows or columns of a tensor according to the indices.

spyrit.core.noise.PoissonApproxGaussSameNoise.forward

PoissonApproxGaussSameNoise.**forward**(*x: tensor*) → tensor

Simulates measurements corrupted by Poisson noise

Args:

x: Batch of images

Shape:

- x: $(*, N)$
- Output: $(*, M)$

Example 1: Two noisy measurement vectors from a *Linear* measurement operator

```
>>> H = torch.rand([400,32*32])
>>> meas_op = Linear(H)
>>> noise_op = PoissonApproxGaussSameNoise(meas_op, 10.0)
>>> x = torch.FloatTensor(10, 32*32).uniform_(-1, 1)
>>> y = noise_op(x)
>>> print(y.shape)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
>>> y = noise_op(x)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
torch.Size([10, 400])
Measurements in (2255.57 , 2911.18)
Measurements in (2226.49 , 2934.42)
```

Example 2: Two noisy measurement vectors from a *HadamSplit* operator

```
>>> Perm = torch.rand([32*32,32*32])
>>> meas_op = HadamSplit(H, Perm, 32, 32)
>>> noise_op = PoissonApproxGaussSameNoise(meas_op, 200.0)
>>> x = torch.FloatTensor(10, 32*32).uniform_(-1, 1)
>>> y = noise_op(x)
>>> print(y.shape)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
>>> y = noise_op(x)
>>> print(f"Measurements in ({torch.min(y):.2f} , {torch.max(y):.2f})")
torch.Size([10, 800])
Measurements in (0.00 , 55951.41)
Measurements in (0.00 , 56216.86)
```

spyrit.core.noise.PoissonApproxGaussSameNoise.sort_by_indices

PoissonApproxGaussSameNoise.**sort_by_indices**(*x: tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → tensor

Reorder the rows or columns of a tensor according to the indices.

The indices are stored in the attribute `self.meas_op.indices` and are used to reorder the rows or columns of the input tensor x. The indices give the order in which the rows or columns should be reordered.

..note::
This method is identical to the function *sort_by_indices()*.

Args:

x (torch.tensor):
Input tensor to be reordered. The tensor must have the same number of rows or columns as the number of elements in the attribute `self.indices`.

axis (str, optional):
Axis along which to order the tensor. Must be either "rows" or "cols". Defaults to "rows".

inverse_permutation (bool, optional): *
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not "rows" or "cols".

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

torch.tensor:
Tensor x with reordered rows or columns according to the indices.

spyrit.core.prep

Preprocessing operators applying affine transformations to the measurements.

There are two classes in this module: *DirectPoisson* and *SplitPoisson*. The first one is used for direct measurements (i.e. without splitting the measurement matrix in its positive and negative parts), while the second one is used for split measurements.

Classes

DirectPoisson(alpha, meas_op)	Preprocess the raw data acquired with a direct measurement operator assuming Poisson noise.
SplitPoisson(alpha, meas_op)	Preprocess the raw data acquired with a split measurement operator assuming Poisson noise.

spyrit.core.prep.DirectPoisson

class spyrit.core.prep.**DirectPoisson**(*alpha: float*, *meas_op*)

Bases: Module

Preprocess the raw data acquired with a direct measurement operator assuming Poisson noise. It also compensates for the affine transformation applied to the images to get positive intensities.

It computes $m = \frac{2}{\alpha} y - H1$ and the variance $\sigma^2 = 4\frac{y}{\alpha^2}$, where $y = Hx$ are obtained using a direct linear measurement operator (see `spyrit.core.Linear`), α is the image intensity, and 1 is the all-ones vector.

Args:

`alpha`: maximun image intensity α (in counts)

`meas_op`: measurement operator (see *meas*)

Example:

```
>>> H = torch.rand([400,32*32])
>>> meas_op =  Linear(H)
>>> prep_op = DirectPoisson(1.0, meas_op)
```

Methods

denormalize_expe(x, beta, h, w)	Denormalize images from the range [-1;1] to the range [0; β]
forward(x)	Preprocess measurements to compensate for the affine image normalization
sigma(x)	Estimates the variance of the preprocessed measurements

spyrit.core.prep.DirectPoisson.denormalize_expe

DirectPoisson.**denormalize_expe**(*x*, *beta*, *h*, *w*)

Denormalize images from the range [-1;1] to the range [0; β]

It computes $m = \frac{\beta}{2}(x + 1)$, where β is the normalization factor.

Args:

`x`: Batch of images

`beta`: Normalizarion factor

`h`: Image height

`w`: Image width

Shape:

`x`: $(*, 1, h, w)$

`beta`: $(*)$ or $(*, 1)$

`h`: int

`w`: int

`Output`: $(*, 1, h, w)$

Example:

```
>>> x = torch.rand([10, 1, 32,32], dtype=torch.float)
>>> beta = 9*torch.rand([10])
>>> y = prep_op.denormalize_expe(x, beta, 32, 32)
>>> print(y.shape)
torch.Size([10, 1, 32, 32])
```

spyrit.core.prep.DirectPoisson.forward

`DirectPoisson.`**`forward`**(*x: tensor*) → tensor

Preprocess measurements to compensate for the affine image normalization

It computes $\frac{2}{\alpha}x - H1$, where H1 represents the all-ones vector.

Args:
`x`: batch of measurement vectors

Shape:
x: (B, M) where B is the batch dimension

meas_op: the number of measurements `meas_op.M` should match M.

Output: (B, M)

Example:

```
>>> x = torch.rand([10,400], dtype=torch.float)
>>> H = torch.rand([400,32*32])
>>> meas_op =  Linear(H)
>>> prep_op = DirectPoisson(1.0, meas_op)
>>> m = prep_op(x)
>>> print(m.shape)
torch.Size([10, 400])
```

spyrit.core.prep.DirectPoisson.sigma

`DirectPoisson.`**`sigma`**(*x: tensor*) → tensor

Estimates the variance of the preprocessed measurements

The variance is estimated as $\frac{4}{\alpha^2}x$

Args:
`x`: batch of measurement vectors

Shape:
`x`: (B, M) where B is the batch dimension

Output: (B, M)

Example:

```
>>> x = torch.rand([10,400], dtype=torch.float)
>>> v = prep_op.sigma(x)
>>> print(v.shape)
torch.Size([10, 400])
```

spyrit.core.prep.SplitPoisson

class spyrit.core.prep.**SplitPoisson**(*alpha: float*, *meas_op*)

Bases: Module

Preprocess the raw data acquired with a split measurement operator assuming Poisson noise. It also compensates for the affine transformation applied to the images to get positive intensities.

It computes $m = \frac{y_+ - y_-}{\alpha} - H1$ and the variance $var = \frac{2(y_+ + y_-)}{\alpha^2}$, where $y_+ = H_+ x$ and $y_- = H_- x$ are obtained using a split measurement operator (see `spyrit.core.LinearSplit`), α is the image intensity, and 1 is the all-ones vector.

Args:

alpha (float): maximun image intensity α (in counts)

`meas_op`: measurement operator (see *meas*)

Example:

```
>>> H = torch.rand([400,32*32])
>>> meas_op =  LinearSplit(H)
>>> split_op = SplitPoisson(10, meas_op)
```

Example 2:

```
>>> Perm = torch.rand([32,32])
>>> meas_op = HadamSplit(400, 32,  Perm)
>>> split_op = SplitPoisson(10, meas_op)
```

Methods

denormalize_expe(x, beta, h, w)	Denormalize images from the range [-1;1] to the range [0; β]
forward(x)	Preprocess to compensates for image normalization and splitting of the measurement operator.
forward_expe(x, meas_op)	Preprocess to compensate for image normalization and splitting of the measurement operator.
set_expe([gain, mudark, sigdark, nbin])	Sets experimental parameters of the sensor
sigma(x)	Estimates the variance of the preprocessed measurements
sigma_expe(x)	Estimates the variance of the measurements that are compensated for splitting but **NOT** for image normalization
sigma_from_image(x, meas_op)	Estimates the variance of the preprocessed measurements corresponding to images through a measurement operator

spyrit.core.prep.SplitPoisson.denormalize_expe

SplitPoisson.**denormalize_expe**(*x*, *beta*, *h*, *w*)

Denormalize images from the range [-1;1] to the range [0; β]

It computes $m = \frac{\beta}{2}(x+1)$, where β is the normalization factor.

Args:

- `x`: Batch of images
- `beta`: Normalizarion factor
- `h`: Image height
- `w`: Image width

Shape:

- `x`: $(*, 1, h, w)$
- `beta`: $(*)$ or $(*, 1)$
- `h`: int
- `w`: int
- Output: $(*, 1, h, w)$

Example:

```
>>> x = torch.rand([10, 1, 32,32], dtype=torch.float)
>>> beta = 9*torch.rand([10])
>>> y = split_op.denormalize_expe(x, beta, 32, 32)
>>> print(y.shape)
torch.Size([10, 1, 32, 32])
```

spyrit.core.prep.SplitPoisson.forward

SplitPoisson.**forward**(*x: tensor*) $\rightarrow$ tensor

Preprocess to compensates for image normalization and splitting of the measurement operator.

It computes $\frac{x[0::2]-x[1::2]}{\alpha} - H1$

Args:

`x`: batch of measurement vectors

Shape:

x: $(*, 2M)$ where $*$ indicates one or more dimensions

meas_op: the number of measurements `meas_op.M` should match M.

Output: $(*, M)$

Example:

```
>>> x = torch.rand([10,2*400], dtype=torch.float)
>>> H = torch.rand([400,32*32])
>>> meas_op =  LinearSplit(H)
>>> split_op = SplitPoisson(10, meas_op)
>>> m = split_op(x)
```

(continues on next page)

(continued from previous page)

```
>>> print(m.shape)
torch.Size([10, 400])
```

Example 2:

```
>>> x = torch.rand([10,2*400], dtype=torch.float)
>>> Perm = torch.rand([32,32])
>>> meas_op = HadamSplit(400, 32,  Perm)
>>> split_op = SplitPoisson(10, meas_op)
>>> m = split_op(x)
>>> print(m.shape)
torch.Size([10, 400])
```

spyrit.core.prep.SplitPoisson.forward_expe

SplitPoisson.**forward_expe**(*x: tensor*, *meas_op:* LinearSplit | HadamSplit) → Tuple[tensor, tensor]

Preprocess to compensate for image normalization and splitting of the measurement operator.

It computes $m = \frac{x[0::2]-x[1::2]}{\alpha}$, where $\alpha = \max H^{\dagger}(x[0::2] - x[1::2])$.

Contrary to *forward()*, the image intensity α is estimated from the pseudoinverse of the unsplit measurements. This method is typically called for the reconstruction of experimental measurements, while *forward()* is called in simulations.

The method returns a tuple containing both m and α

Args:

`x`: batch of measurement vectors

`meas_op`: measurement operator (required to estimate α)

Output (m, α): preprocess measurement and estimated intensities.

Shape:

x: $(B, 2M)$ where B is the batch dimension

meas_op: the number of measurements `meas_op.M` should match M.

m: (B, M)

α: (B)

Example:

```
>>> x = torch.rand([10,2*400], dtype=torch.float)
>>> Perm = torch.rand([32,32])
>>> meas_op = HadamSplit(400, 32,  Perm)
>>> split_op = SplitPoisson(10, meas_op)
>>> m, alpha = split_op.forward_expe(x, meas_op)
>>> print(m.shape)
>>> print(alpha.shape)
torch.Size([10, 400])
torch.Size([10])
```

spyrit.core.prep.SplitPoisson.set_expe

SplitPoisson.**set_expe**(*gain=1.0*, *mudark=0.0*, *sigdark=0.0*, *nbin=1.0*)

Sets experimental parameters of the sensor

Args:

- `gain` (float): gain (in count/electron)
- `mudark` (float): average dark current (in counts)
- `sigdark` (float): standard deviation or dark current (in counts)
- `nbin` (float): number of raw bin in each spectral channel (if input x results from the sommation/binning of the raw data)

Example:

```
>>> split_op.set_expe(gain=1.6)
>>> print(split_op.gain)
1.6
```

spyrit.core.prep.SplitPoisson.sigma

SplitPoisson.**sigma**(*x: tensor*) → tensor

Estimates the variance of the preprocessed measurements

The variance is estimated as $\frac{4}{\alpha^2} H(x[0::2] + x[1::2])$

Args:

`x`: batch of images in the Hadamard domain

Shape:

- Input: $(*, 2 * M)$ $*$ indicates one or more dimensions
- Output: $(*, M)$

Example:

```
>>> x = torch.rand([10,2*400], dtype=torch.float)
>>> v = split_op.sigma(x)
>>> print(v.shape)
torch.Size([10, 400])
```

spyrit.core.prep.SplitPoisson.sigma_expe

SplitPoisson.**sigma_expe**(*x: tensor*) → tensor

Estimates the variance of the measurements that are compensated for splitting but **NOT** for image normalization

Args:

`x`: Batch of images in the Hadamard domain.

Shape:

Input: $(B, 2 * M)$ where B is the batch dimension

Output: (B, M)

Example:

```
>>> x = torch.rand([10,2*32*32], dtype=torch.float)
>>> split_op.set_expe(gain=1.6)
>>> v = split_op.sigma_expe(x)
>>> print(v.shape)
torch.Size([10, 400])
```

spyrit.core.prep.SplitPoisson.sigma_from_image

SplitPoisson.**sigma_from_image**(*x: tensor*, *meas_op:* LinearSplit | HadamSplit) → tensor

Estimates the variance of the preprocessed measurements corresponding to images through a measurement operator

The variance is estimated as $\frac{4}{\alpha}\{(Px)[0::2] + (Px)[1::2]\}$

Args:

`x`: Batch of images

`meas_op`: Measurement operator

Shape:

`x`: $(*, N)$

`meas_op`: An operator such that `meas_op.N` $= N$ and `meas_op.M` $= M$

Output: $(*, M)$

Example:

```
>>> x = torch.rand([10,2*400], dtype=torch.float)
>>> Perm = torch.rand([32,32])
>>> meas_op = HadamSplit(400, 32,  Perm)
>>> split_op = SplitPoisson(10, meas_op)
>>> v = split_op.sigma_from_image(x, meas_op)
>>> print(v.shape)
torch.Size([10, 400])
```

spyrit.core.recon

Reconstruction methods and networks.

Classes

DCDRUNet(noise, prep, sigma[, denoi, ...])	Denoised completion reconstruction network based on DRUNet wich concatenates a
DCNet(noise, prep, sigma[, denoi])	Denoised completion reconstruction network
Denoise_layer(M)	Wiener filter that assumes additive white Gaussian noise.
PinvNet(noise, prep[, denoi])	Pseudo inverse reconstruction network
PositiveMonoIncreaseParameters(size[, val_min])	
PositiveParameters(size[, val_min])	
PseudoInverse()	Moore-Penrose pseudoinverse.
TikhonovMeasurementPriorDiag(sigma, M)	Tikhonov regularization with prior in the measurement domain.
UPGD(noise, prep[, denoi, num_iter, lamb, ...])	

spyrit.core.recon.DCDRUNet

class spyrit.core.recon.**DCDRUNet**(*noise*, *prep*, *sigma*, *denoi=Identity()*, *noise_level=5*)

Bases: *DCNet*

Denoised completion reconstruction network based on DRUNet wich concatenates a noise level map to the input

Args:

`noise`: Acquisition operator (see *noise*)

`prep`: Preprocessing operator (see *prep*)

`sigma`: UPDATE!! Tikhonov reconstruction operator of type *TikhonovMeasurementPriorDiag()*

`denoi` (optional): Image denoising operator (see *nnet*). Default *Identity*

`noise_level` (optional): Noise level in the range [0, 255], default is noise_level=5

Input / Output:

`input`: Ground-truth images with concatenated noise level map with shape $(B, C+1, H, W)$

`output`: Reconstructed images with shape (B, C, H, W)

Attributes:

`Acq`: Acquisition operator initialized as `noise`

`PreP`: Preprocessing operator initialized as `prep`

`DC_Layer`: Data consistency layer initialized as `tikho`

`Denoi`: Image (DRUNet architecture type) denoising operator initialized as `denoi`

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
```

(continues on next page)

(continued from previous page)

```
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> sigma = torch.rand([H**2, H**2])
>>> n_channels = 1                    # 1 for grayscale image
>>> model_drunet_path = './spyrit/drunet/model_zoo/drunet_gray.pth'
>>> denoi_drunet = drunet(in_nc=n_channels+1, out_nc=n_channels, nc=[64, 128,␣
↪256, 512], nb=4, act_mode='R',
    downsample_mode="strideconv", upsample_mode="convtranspose")
>>> recnet = DCDRUNet(noise,prep,sigma,denoi_drunet)
>>> z = recnet(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

Methods

acquire(x)	Simulate data acquisition
concat_noise_map(x)	Concatenation of noise level map to reconstructed images
forward(x)	Full pipeline of the reconstruction network
reconstruct(x)	Reconstruction step of a reconstruction network
reconstruct_expe(x)	Reconstruction step of a reconstruction network
set_noise_level(noise_level)	Reset noise level value

spyrit.core.recon.DCDRUNet.acquire

DCDRUNet.**acquire**(x)

Simulate data acquisition

Args:

`x`: ground-truth images

Shape:

`x`: ground-truth images with shape (B, C, H, W)

`output`: measurement vectors with shape $(BC, 2M)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> sigma = torch.rand([H**2, H**2])
>>> recnet = DCNet(noise,prep,sigma)
>>> x = torch.FloatTensor(B,C,H,H).uniform_(-1, 1)
>>> z = recnet.acquire(x)
>>> print(z.shape)
torch.Size([10, 8192])
```

spyrit.core.recon.DCDRUNet.concat_noise_map

DCDRUNet.**concat_noise_map**(x)

Concatenation of noise level map to reconstructed images

Args:

x: reconstructed images from the reconstruction layer

Shape:

x: reconstructed images with shape $(BC, 1, H, W)$

output: reconstructed images with concatenated noise level map with shape $(BC, 2, H, W)$

spyrit.core.recon.DCDRUNet.forward

DCDRUNet.**forward**(x)

Full pipeline of the reconstruction network

Args:

x: ground-truth images

Shape:

x: ground-truth images with shape (B, C, H, W)

output: reconstructed images with shape (B, C, H, W)

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> sigma = torch.rand([H**2, H**2])
>>> recnet = DCNet(noise,prep,sigma)
>>> x = torch.FloatTensor(B,C,H,H).uniform_(-1, 1)
>>> z = recnet(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

spyrit.core.recon.DCDRUNet.reconstruct

DCDRUNet.**reconstruct**(x)

Reconstruction step of a reconstruction network

Args:

x: raw measurement vectors

Shape:

x: raw measurement vectors with shape $(BC, 2M)$

output: reconstructed images with shape $(BC, 1, H, W)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> sigma = torch.rand([H**2, H**2])
>>> n_channels = 1                    # 1 for grayscale image
>>> model_drunet_path = './spyrit/drunet/model_zoo/drunet_gray.pth'
>>> denoi_drunet = drunet(in_nc=n_channels+1, out_nc=n_channels, nc=[64,␣
↪128, 256, 512], nb=4, act_mode='R',
    downsample_mode="strideconv", upsample_mode="convtranspose")
>>> recnet = DCDRUNet(noise,prep,sigma,denoi_drunet)
>>> x = torch.rand((B*C,2*M), dtype=torch.float)
>>> z = recnet.reconstruct(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

spyrit.core.recon.DCDRUNet.reconstruct_expe

DCDRUNet.**reconstruct_expe**(*x*)

Reconstruction step of a reconstruction network

Same as *reconstruct()* reconstruct except that:

1. The preprocessing step estimates the image intensity. The estimated intensity is used for both normalizing the raw data and computing the variance of the normalized data.
2. The output images are "denormalized", i.e., have units of photon counts

Args:

`x`: raw measurement vectors

Shape:

`x`: $(BC, 2M)$

`output`: $(BC, 1, H, W)$

spyrit.core.recon.DCDRUNet.set_noise_level

DCDRUNet.**set_noise_level**(*noise_level*)

Reset noise level value

Args:

`noise_level`: noise level value in the range [0, 255]

Shape:

`noise_level`: float value noise level (1)

`output`: noise level tensor with shape (1)

spyrit.core.recon.DCNet

class spyrit.core.recon.**DCNet**(*noise:* NoNoise, *prep:* DirectPoisson | SplitPoisson, *sigma: tensor*, *denoi=Identity()*)

Bases: `Module`

Denoised completion reconstruction network

Args:

`noise`: Acquisition operator (see *noise*)

`prep`: Preprocessing operator (see *prep*)

`sigma`: UPDATE!! Tikhonov reconstruction operator of type *TikhonovMeasurementPriorDiag()*

`denoi` (optional): Image denoising operator (see *nnet*). Default *Identity*

Input / Output:

`input`: Ground-truth images with shape (B, C, H, W)

`output`: Reconstructed images with shape (B, C, H, W)

Attributes:

`Acq`: Acquisition operator initialized as `noise`

`PreP`: Preprocessing operator initialized as `prep`

`DC_Layer`: Data consistency layer initialized as `tikho`

`Denoi`: Image denoising operator initialized as `denoi`

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> sigma = torch.rand([H**2, H**2])
>>> recnet = DCNet(noise,prep,sigma)
>>> x = torch.FloatTensor(B,C,H,H).uniform_(-1, 1)
>>> z = recnet(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

Methods

acquire(x)	Simulate data acquisition
forward(x)	Full pipeline of the reconstruction network
reconstruct(x)	Reconstruction step of a reconstruction network
reconstruct_expe(x)	Reconstruction step of a reconstruction network

spyrit.core.recon.DCNet.acquire

DCNet.**acquire**(x)

Simulate data acquisition

Args:

x: ground-truth images

Shape:

x: ground-truth images with shape (B, C, H, W)

output: measurement vectors with shape $(BC, 2M)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> sigma = torch.rand([H**2, H**2])
>>> recnet = DCNet(noise,prep,sigma)
>>> x = torch.FloatTensor(B,C,H,H).uniform_(-1, 1)
>>> z = recnet.acquire(x)
>>> print(z.shape)
torch.Size([10, 8192])
```

spyrit.core.recon.DCNet.forward

DCNet.**forward**(x)

Full pipeline of the reconstruction network

Args:

x: ground-truth images

Shape:

x: ground-truth images with shape (B, C, H, W)

output: reconstructed images with shape (B, C, H, W)

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> sigma = torch.rand([H**2, H**2])
>>> recnet = DCNet(noise,prep,sigma)
>>> x = torch.FloatTensor(B,C,H,H).uniform_(-1, 1)
>>> z = recnet(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

spyrit.core.recon.DCNet.reconstruct

DCNet.**reconstruct**(x)

Reconstruction step of a reconstruction network

Args:

x: raw measurement vectors

Shape:

x: raw measurement vectors with shape $(BC, 2M)$

output: reconstructed images with shape $(BC, 1, H, W)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> sigma = torch.rand([H**2, H**2])
>>> recnet = DCNet(noise,prep,sigma)
>>> x = torch.rand((B*C,2*M), dtype=torch.float)
>>> z = recnet.reconstruct(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

spyrit.core.recon.DCNet.reconstruct_expe

DCNet.**reconstruct_expe**(x)

Reconstruction step of a reconstruction network

Same as *reconstruct()* reconstruct except that:

1. The preprocessing step estimates the image intensity. The estimated intensity is used for both normalizing the raw data and computing the variance of the normalized data.

2. The output images are “denormalized”, i.e., have units of photon counts

Args:

x: raw measurement vectors

Shape:

x: $(BC, 2M)$

output: $(BC, 1, H, W)$

spyrit.core.recon.Denoise_layer

class spyrit.core.recon.**Denoise_layer**(*M: int*)

Bases: Module

Wiener filter that assumes additive white Gaussian noise.

$y = \sigma^2_{\text{prior}}/(\sigma^2_{\text{prior}} + \sigma^2_{\text{meas}})x, where : math : `\sigma^2_{\text{prior}}` is the variance prior and : math : `\sigma^2_{\text{meas}}` is the variance of the measuremen$

Args:

`M` (int): size of incoming vector

Shape:

- Input: $(*, M)$.
- Output: $(*, M)$.

Attributes:

`weight`: The learnable standard deviation prior σ_{prior} of shape $(M, 1)$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = 1/M$.

`in_features`: The number of input features equal to M.

Example:

```
>>> m = Denoise_layer(30)
>>> input = torch.randn(128, 30)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

Methods

forward(inputs)	Applies a transformation to the incoming data: $y = A^2/(A^2 + x)$.
reset_parameters()	Resets the standard deviation prior σ_{prior}.
tikho(inputs, weight)	Applies a transformation to the incoming data: $y = A^2/(A^2 + x)$.

spyrit.core.recon.Denoise_layer.forward

Denoise_layer.**forward**(*inputs: tensor*) → tensor

Applies a transformation to the incoming data: $y = A^2/(A^2 + x)$.

x is the input tensor (see `inputs`) and A is the standard deviation prior (see `self.weight`).

Args:

`inputs` (torch.tensor): input tensor x of shape $(N, *, in\_features)$

Returns:

torch.tensor: The transformed data y of shape $(N, in\_features)$

Shape:

spyrit.core.recon.Denoise_layer.reset_parameters

Denoise_layer.**reset_parameters**()

Resets the standard deviation prior σ_{prior}.

The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = 1/M$. They are stored in the `weight` attribute.

spyrit.core.recon.Denoise_layer.tikho

static Denoise_layer.**tikho**(*inputs: tensor*, *weight: tensor*) → tensor

Applies a transformation to the incoming data: $y = A^2/(A^2 + x)$.

x is the input tensor (see `inputs`) and A is the standard deviation prior (see `weight`).

Args:

`inputs` (torch.tensor): input tensor x of shape $(N, *, in\_features)$

`weight` (torch.tensor): standard deviation prior A of shape $(in\_features)$

Returns:

torch.tensor: The transformed data y of shape $(N, in\_features)$

Shape:

- `inputs`: $(N, *, in\_features)$ where * means any number of additional dimensions - Variance of measurements
- `weight`: $(in\_features)$ - corresponds to the standard deviation of our prior.
- `output`: $(N, in\_features)$

spyrit.core.recon.PinvNet

class spyrit.core.recon.**PinvNet**(*noise*, *prep*, *denoi=Identity()*)

Bases: `Module`

Pseudo inverse reconstruction network

Args:

`noise`: Acquisition operator (see *noise*)

`prep`: Preprocessing operator (see *prep*)

`denoi` (optional): Image denoising operator (see *nnet*). Default *Identity*

Input / Output:

`input`: Ground-truth images with shape (B, C, H, W) corresponding to the batch size, number of channels, height, and width.

`output`: Reconstructed images with shape (B, C, H, W) corresponding to the batch size, number of channels, height, and width.

Attributes:

`Acq`: Acquisition operator initialized as `noise`

`prep`: Preprocessing operator initialized as `prep`

`pinv`: Analytical reconstruction operator initialized as *PseudoInverse()*

`Denoi`: Image denoising operator initialized as `denoi`

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> recnet = PinvNet(noise, prep)
>>> x = torch.FloatTensor(B,C,H,H).uniform_(-1, 1)
>>> z = recnet(x)
>>> print(z.shape)
>>> print(torch.linalg.norm(x - z)/torch.linalg.norm(x))
torch.Size([10, 1, 64, 64])
tensor(5.8912e-06)
```

Methods

acquire(x)	Simulates data acquisition
forward(x)	Full pipeline of reconstrcution network
meas2img(y)	Returns images from raw measurement vectors
reconstruct(x)	Preprocesses, reconstructs, and denoises raw measurement vectors.
reconstruct_expe(x)	Reconstruction step of a reconstruction network
reconstruct_pinv(x)	Preprocesses and reconstructs raw measurement vectors.

spyrit.core.recon.PinvNet.acquire

PinvNet.**acquire**(x)

Simulates data acquisition

Args:

x: ground-truth images

Shape:

x: ground-truth images with shape (B, C, H, W)

output: measurement vectors with shape $(BC, 2M)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> recnet = PinvNet(noise, prep)
>>> x = torch.FloatTensor(B,C,H,H).uniform_(-1, 1)
>>> z = recnet.acquire(x)
>>> print(z.shape)
torch.Size([10, 8192])
```

spyrit.core.recon.PinvNet.forward

PinvNet.**forward**(x)

Full pipeline of reconstrcution network

Args:

x: ground-truth images

Shape:

x: ground-truth images with shape (B, C, H, W)

output: reconstructed images with shape (B, C, H, W)

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> recnet = PinvNet(noise, prep)
>>> x = torch.FloatTensor(B,C,H,H).uniform_(-1, 1)
>>> z = recnet(x)
>>> print(z.shape)
>>> print(torch.linalg.norm(x - z)/torch.linalg.norm(x))
torch.Size([10, 1, 64, 64])
tensor(5.8912e-06)
```

spyrit.core.recon.PinvNet.meas2img

PinvNet.**meas2img**(y)

Returns images from raw measurement vectors

Args:

x: raw measurement vectors

Shape:

x: $(BC, 2M)$

output: $(BC, 1, H, W)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H**2)
>>> recnet = PinvNet(noise, prep)
>>> x = torch.rand((B*C,2*M), dtype=torch.float)
>>> z = recnet.reconstruct(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

spyrit.core.recon.PinvNet.reconstruct

PinvNet.**reconstruct**(x)

Preprocesses, reconstructs, and denoises raw measurement vectors.

Args:

x: raw measurement vectors

Shape:

x: $(BC, 2M)$

output: $(BC, 1, H, W)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H**2)
>>> recnet = PinvNet(noise, prep)
>>> x = torch.rand((B*C,2*M), dtype=torch.float)
>>> z = recnet.reconstruct(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

spyrit.core.recon.PinvNet.reconstruct_expe

PinvNet.**reconstruct_expe**(x)

Reconstruction step of a reconstruction network

Same as *reconstruct()* reconstruct except that:

1. The preprocessing step estimates the image intensity for normalization
2. The output images are "denormalized", i.e., have units of photon counts

Args:

x: raw measurement vectors

Shape:

x: $(BC, 2M)$

output: $(BC, 1, H, W)$

spyrit.core.recon.PinvNet.reconstruct_pinv

PinvNet.**reconstruct_pinv**(x)

Preprocesses and reconstructs raw measurement vectors.

Args:

x: raw measurement vectors

Shape:

x: $(BC, 2M)$

output: $(BC, 1, H, W)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H**2)
>>> recnet = PinvNet(noise, prep)
>>> x = torch.rand((B*C,2*M), dtype=torch.float)
>>> z = recnet.reconstruct_pinv(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

spyrit.core.recon.PositiveMonoIncreaseParameters

class spyrit.core.recon.**PositiveMonoIncreaseParameters**(*size*, *val_min=1e-06*)

Bases: *PositiveParameters*

Methods

forward()	Define the computation performed at every call.

spyrit.core.recon.PositiveMonoIncreaseParameters.forward

PositiveMonoIncreaseParameters.**forward**()

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

spyrit.core.recon.PositiveParameters

class spyrit.core.recon.**PositiveParameters**(*size*, *val_min=1e-06*)

Bases: `Module`

Methods

forward()	Define the computation performed at every call.

spyrit.core.recon.PositiveParameters.forward

PositiveParameters.**forward**()

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

spyrit.core.recon.PseudoInverse

class spyrit.core.recon.**PseudoInverse**

Bases: Module

Moore-Penrose pseudoinverse.

Considering linear measurements $y = Hx$, where H is the measurement matrix and x is a vectorized image, it estimates x from y by computing $\hat{x} = H^{\dagger}y$, where H is the Moore-Penrose pseudo inverse of H.

Example:

```
>>> H = torch.rand([400,32*32])
>>> Perm = torch.rand([32*32,32*32])
>>> meas_op =  HadamSplit(H, Perm, 32, 32)
>>> y = torch.rand([85,400], dtype=torch.float)
>>> pinv_op = PseudoInverse()
>>> x = pinv_op(y, meas_op)
>>> print(x.shape)
torch.Size([85, 1024])
```

Methods

forward(x, meas_op)	Computes pseudo-inverse of measurements.

spyrit.core.recon.PseudoInverse.forward

`PseudoInverse.forward`(*x: tensor*, *meas_op*) → tensor

Computes pseudo-inverse of measurements.

Args:

`x`: Batch of measurement vectors.

`meas_op`: Measurement operator. Any class that implements a `pinv()` method can be used, e.g., `HadamSplit`.

Shape:

`x`: $(*, M)$

`meas_op`: not applicable

`output`: $(*, N)$

Example:

```
>>> H = torch.rand([400,32*32])
>>> Perm = torch.rand([32*32,32*32])
>>> meas_op =  HadamSplit(H, Perm, 32, 32)
>>> y = torch.rand([85,400], dtype=torch.float)
>>> pinv_op = PseudoInverse()
>>> x = pinv_op(y, meas_op)
>>> print(x.shape)
torch.Size([85, 1024])
```

spyrit.core.recon.TikhonovMeasurementPriorDiag

class spyrit.core.recon.**TikhonovMeasurementPriorDiag**(*sigma: tensor*, *M: int*)

Bases: `Module`

Tikhonov regularization with prior in the measurement domain.

Considering linear measurements $y = Hx$, where $H = GF$ is the measurement matrix and x is a vectorized image, it estimates x from y by approximately minimizing

$$\|y - GFx\|^2_{\Sigma_\alpha^{-1}} + \|F(x - x_0)\|^2_{\Sigma^{-1}}$$

where x_0 is a mean image prior, Σ is a covariance prior, and Σ_α is the measurement noise covariance.

The class is constructed from Σ.

Args:

- `sigma`: covariance prior with shape (N, N)
- `M`: number of measurements

Attributes:

`comp`: The learnable completion layer initialized as $\Sigma_1\Sigma_{21}^{-1}$. This layer is a `nn.Linear`

`denoi`: The learnable denoising layer initialized from Σ_1.

Example:

```
>>> sigma = torch.rand([32*32, 32*32])
>>> recon_op = TikhonovMeasurementPriorDiag(sigma, 400)
```

Methods

forward(x, x_0, var, meas_op)	Computes the Tikhonov regularization with prior in the measurement domain.

spyrit.core.recon.TikhonovMeasurementPriorDiag.forward

TikhonovMeasurementPriorDiag.**forward**(*x: tensor*, *x_0: tensor*, *var: tensor*, *meas_op:* HadamSplit) → tensor

Computes the Tikhonov regularization with prior in the measurement domain.

We approximate the solution as:

$$\hat{x} = x_0 + F^{-1} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

with $y_1 = D_1(D_1+\Sigma_\alpha)^{-1}(y-GFx_0)$ and $y_2 = \Sigma_1\Sigma_{21}^{-1}y_1$, where $\Sigma = \begin{bmatrix} \Sigma_1 & \Sigma_{21}^\top \\ \Sigma_{21} & \Sigma_2 \end{bmatrix}$ and $D_1 = \text{Diag}(\Sigma_1)$. Assuming the noise covariance Σ_α is diagonal, the matrix inversion involded in the computation of y_1 is straigtforward.

This is an approximation to the exact solution

$$\hat{x} = x_0 + F^{-1} \begin{bmatrix} \Sigma_1 \\ \Sigma_{21} \end{bmatrix} [\Sigma_1 + \Sigma_\alpha]^{-1}(y - GFx_0)$$

See Lemma B.0.5 of the PhD dissertation of A. Lorente Mur (2021): https://theses.hal.science/tel-03670825v1/file/these.pdf

Args:

- `x`: A batch of measurement vectors y
- `x_0`: A batch of prior images x_0
- `var`: A batch of measurement noise variances Σ_α
- `meas_op`: A measurement operator that provides GF and F^{-1}

Shape:

- `x`: $(*, M)$
- `x_0`: $(*, N)$
- `var` $(*, M)$
- Output: $(*, N)$

Example:

```
>>> B, H, M = 85, 32, 512
>>> sigma = torch.rand([H**2, H**2])
>>> recon_op = TikhonovMeasurementPriorDiag(sigma, M)
>>> Ord = torch.ones((H,H))
>> meas = HadamSplit(M, H, Ord)
>>> y = torch.rand([B,M], dtype=torch.float)
>>> x_0 = torch.zeros((B, H**2), dtype=torch.float)
>>> var = torch.zeros((B, M), dtype=torch.float)
>>> x = recon_op(y, x_0, var, meas)
torch.Size([85, 1024])
```

spyrit.core.recon.UPGD

class spyrit.core.recon.**UPGD**(*noise*, *prep*, *denoi=Identity()*, *num_iter=6*, *lamb=1e-05*, *lamb_min=1e-06*, *split=False*)

Bases: *PinvNet*

Methods

acquire(x)	Simulates data acquisition
forward(x)	Full pipeline of reconstrcution network
meas2img(y)	Returns images from raw measurement vectors
reconstruct(x)	Reconstruction step of a reconstruction network
reconstruct_expe(x)	Reconstruction step of a reconstruction network
reconstruct_pinv(x)	Preprocesses and reconstructs raw measurement vectors.

spyrit.core.recon.UPGD.acquire

UPGD.**acquire**(*x*)

Simulates data acquisition

Args:

x: ground-truth images

Shape:

x: ground-truth images with shape (B, C, H, W)

output: measurement vectors with shape $(BC, 2M)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> recnet = PinvNet(noise, prep)
>>> x = torch.FloatTensor(B,C,H,H).uniform_(-1, 1)
>>> z = recnet.acquire(x)
```

(continues on next page)

(continued from previous page)

```
>>> print(z.shape)
torch.Size([10, 8192])
```

spyrit.core.recon.UPGD.forward

UPGD.**forward**(x)

Full pipeline of reconstrcution network

Args:

x: ground-truth images

Shape:

x: ground-truth images with shape (B, C, H, W)

output: reconstructed images with shape (B, C, H, W)

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H*H)
>>> recnet = PinvNet(noise, prep)
>>> x = torch.FloatTensor(B,C,H,H).uniform_(-1, 1)
>>> z = recnet(x)
>>> print(z.shape)
>>> print(torch.linalg.norm(x - z)/torch.linalg.norm(x))
torch.Size([10, 1, 64, 64])
tensor(5.8912e-06)
```

spyrit.core.recon.UPGD.meas2img

UPGD.**meas2img**(y)

Returns images from raw measurement vectors

Args:

x: raw measurement vectors

Shape:

x: $(BC, 2M)$

output: $(BC, 1, H, W)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H**2)
>>> recnet = PinvNet(noise, prep)
>>> x = torch.rand((B*C,2*M), dtype=torch.float)
```

(continues on next page)

(continued from previous page)

```
>>> z = recnet.reconstruct(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

spyrit.core.recon.UPGD.reconstruct

UPGD.**reconstruct**(x)

Reconstruction step of a reconstruction network

Same as *reconstruct()* reconstruct except that:

1. The regularization parameter is trainable

Args:
x: raw measurement vectors

Shape:
x: $(BC, 2M)$

output: $(BC, 1, H, W)$

spyrit.core.recon.UPGD.reconstruct_expe

UPGD.**reconstruct_expe**(x)

Reconstruction step of a reconstruction network

Same as *reconstruct()* reconstruct except that:

1. The preprocessing step estimates the image intensity for normalization
2. The output images are "denormalized", i.e., have units of photon counts

Args:
x: raw measurement vectors

Shape:
x: $(BC, 2M)$

output: $(BC, 1, H, W)$

spyrit.core.recon.UPGD.reconstruct_pinv

UPGD.**reconstruct_pinv**(x)

Preprocesses and reconstructs raw measurement vectors.

Args:
x: raw measurement vectors

Shape:
x: $(BC, 2M)$

output: $(BC, 1, H, W)$

Example:

```
>>> B, C, H, M = 10, 1, 64, 64**2
>>> Ord = torch.ones((H,H))
>>> meas = HadamSplit(M, H, Ord)
>>> noise = NoNoise(meas)
>>> prep = SplitPoisson(1.0, M, H**2)
>>> recnet = PinvNet(noise, prep)
>>> x = torch.rand((B*C,2*M), dtype=torch.float)
>>> z = recnet.reconstruct_pinv(x)
>>> print(z.shape)
torch.Size([10, 1, 64, 64])
```

spyrit.core.time

Stores deformation fields and warps images.

Contains *DeformationField* and *AffineDeformationField*, a subclass of the former. These classes are used to warp images according to a deformation field that is stored as as class attribute. They can be fed an image (called "*original image*") and will return the warped image ("*deformed image*").

The function that maps the *original image* pixel coordinates to the *deformed image* pixel coordinates is called the "*deformation field*" and is noted v. The function that maps the pixels of the *deformed image* to the pixels of the *original image* is called the "*inverse deformation field*" and is noted u. The *deformation field* and the *inverse deformation field* are related by the equation $v = u^{-1}$.

Here, the two classes use and store the *inverse deformation field* u as a class attribute.

Classes

AffineDeformationField(inverse_field_matrix, ...)	Stores an affine deformation field andn uses it to compute a discrete deformation field *DeformationField*.
DeformationField(inverse_grid_frames[, ...])	Stores a discrete deformation field as a $(b, Nx, Ny, 2)$ tensor.

spyrit.core.time.AffineDeformationField

class spyrit.core.time.**AffineDeformationField**(*inverse_field_matrix: tensor*, *t0: float*, *t1: float*, *n_frames: int*, *img_size: tuple*, *align_corners=False*)

Bases: *DeformationField*

Stores an affine deformation field andn uses it to compute a discrete deformation field *DeformationField*.

Warps an image or batch of images according to an *inverse affine deformation field* u, i.e. the field that maps the *deformed image* pixel coordinates to the *original image* pixel coordinates.

It is constructed from a function of one parameter (time) that returns a tensor of shape $(3, 3)$ representing a 2D affine homogeneous transformation matrix. The homogeneous transformation matrix corresponds to the *inverse deformation field* u, i.e. the field that maps the pixels of the *deformed image* to the pixels of the *original image*.

To warp an image, the affine transformation matrix is evaluated at each time corresponding to the frames of the animation. The *inverse deformation field* u is then computed from the inverse of the affine transformation matrix, and the image is warped according to the *inverse deformation field* u.

Contrary to *DeformationField*, this class can warp images of variable sizes, as the *inverse deformation field* u is computed from the affine transformation matrix at the desired spatial resolution.

Note: The coordinates are given in the range [-1;1]. When referring to a pixel, its position is the position of its center.

Note: The position [-1;-1] corresponds to the top-left corner of the top-left pixel if `align_corners` is set to *False* (default), and to the center of the top-left pixel if `align_corners` is set to *True*.

Args:

`inverse_field_matrix` (torch.tensor): Function of one parameter (time) that returns a tensor of shape $(3, 3)$ representing a 2D affine homogeneous transformation matrix. That matrix is the *inverse deformation field* u, i.e. the field that maps the pixels of the *deformed image* to the pixels of the *original image*.

`align_corners` (bool, optional): Geometrically, we consider the pixels of the input as squares rather than points. If set to *True*, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to *False*, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic. Default: *False*. See `torch.nn.functional.grid_sample()` for more details.

Attributes:

`self.inverse_field_matrix` (function of one parameter): Function of one parameter (time) that returns a tensor of shape $(3, 3)$ representing a 2D affine homogeneous transformation matrix.

`t0` (float): First time at which the inverse deformation field is computed.

`t1` (float): Last time at which the inverse deformation field is computed.

`n_frames` (int): Number of frames in the animation.

`self.inverse_grid_frames` (torch.tensor): Inverse grid frames that are computed from thr attribute `inverse_field_matrix` upon calling the method `save_inv_grid_frames()`. If set manually, the dtype should be *torch.float32*. Default: *None*.

`self.align_corners` (bool): Should the extrema (-1 and 1) be considered as referring to the center points of the input's corner pixels? Default: *False*.

Example 1: Progressive zooming in

```
>>> def u(t):
...     return torch.tensor([[1-t/10, 0, 0], [0, 1-t/10, 0], [0, 0, 1]])
>>> field = AffineDeformationField(u, align_corners=False)
```

Example 2: Rotation of an image counter-clockwise, at a frequency of 1Hz

```
>>> import numpy as np
>>> def s(t):
...     return np.sin(2*np.pi*t)
>>> def c(t):
...     return np.cos(2*np.pi*t)
>>> def u(t):
...     return torch.tensor([[c(t), s(t), 0], [-s(t), c(t), 0], [0, 0, 1]])
>>> field = AffineDeformationField(u, align_corners=False)
```

Methods

forward(img, n0, n1[, mode])	Warps an image or batch of images with the stored *inverse deformation field* u.
get_inv_grid_frames()	Returns the *inverse deformation field* u.

spyrit.core.time.AffineDeformationField.forward

AffineDeformationField.**forward**(*img: tensor*, *n0: int*, *n1: int*, *mode: str = 'bilinear'*) → tensor

Warps an image or batch of images with the stored *inverse deformation field* u.

Deforms the image or batch of images according to the *inverse deformation field* u contained in the attribute `inverse_grid_frames`, sliced between the frames $n0$ (included) and $n1$ (excluded). u is the field that maps the pixels of the *deformed image* to the pixels of the *original image*.

Args:

`img` (torch.tensor): The image or batch of images to deform of shape (c, Nx, Ny) or (B, c, Nx, Ny), where B is the number of images in the batch, c is the number of channels (usually 1 or 3), and Nx and Ny are the number of pixels along the x-axis and y-axis respectively.

`n0` (int): The index of the first frame to use in the *inverse deformation field*.

`n1` (int): The index of the first frame to exclude in the *inverse deformation field*.

`mode` (str, optional): The interpolation mode to use. It is directly passed to the function `torch.nn.functional.grid_sample()`. It must be one of the following: 'nearest', 'bilinear', 'bicubic'. Defaults to 'bilinear'.

Important: The input shape must be either (c, Nx, Ny) or (B, c, Nx, Ny).

Note: If $n0 < n1$, `inverse_grid_frames` is sliced as follows: `inv_grid_frames[n0:n1, :, :, :]`

Note: If $n0 > n1$, `inverse_grid_frames` is sliced "backwards". The first frame of the warped animation corresponds to the index $n0$, and the last frame corresponds to the index $n1 + 1$. This behavior is identical to slicing a list with a step of -1.

Returns:

`output` (torch.tensor): The deformed batch of images of shape $(|n1 - n0|, c, Nx, Ny)$ or $(B, |n1 - n0|, c, Nx, Ny)$ depending on the input shape, where each image in the batch is deformed according to the *inverse deformation field* u contained in the attribute `inverse_grid_frames`.

Shape:

`img`: (c, Nx, Ny) or (B, c, Nx, Ny), where B is the batch size, c is the number of channels, and Nx and Ny are the number of pixels along the x-axis and y-axis respectively.

`output`: $(|n1 - n0|, c, Nx, Ny)$ or $(B, |n1 - n0|, c, Nx, Ny)$, depending on the input shape.

Example 1: Rotating a 2x2 B&W image by 90 degrees counter-clockwise, using one frame and align_corners=True

```
>>> v = torch.tensor([[[[ 1., -1.], [ 1., 1.]],
                      [[-1., -1.], [-1., 1.]]]])
>>> field = DeformationField(v, align_corners=True)
>>> image = torch.tensor([[[0. , 0.3],
                           [0.7, 1. ]]])
>>> deformed_image = field(image, 0, 1)
>>> print(deformed_image)
tensor([[[[0.3000, 1.0000],
          [0.0000, 0.7000]]]])
```

spyrit.core.time.AffineDeformationField.get_inv_grid_frames

`AffineDeformationField.`**`get_inv_grid_frames`**`()`

Returns the *inverse deformation field* u.

Returns the *inverse deformation field* u contained in the attribute `inverse_grid_frames`.

Returns:

`self.inverse_grid_frames` (torch.tensor): *Inverse deformation field* u of shape $(n\_frames, Nx, Ny, 2)$.

Example 1: Get the inverse deformation field of a 2x2 B&W image rotated by 90 degrees counter-clockwise

```
>>> u = torch.tensor([[[[ 0.5, -0.5], [ 0.5, 0.5]], [[-0.5, -0.5], [-0.5, 0.
↪5]]]])
>>> field = DeformationField(u, align_corners=False)
>>> print(field.get_inv_grid_frames())
tensor([[[[ 0.5, -0.5], [ 0.5, 0.5]], [[-0.5, -0.5], [-0.5, 0.5]]]])
```

spyrit.core.time.DeformationField

`class` `spyrit.core.time.`**`DeformationField`**(*inverse_grid_frames: tensor*, *align_corners=False*)

Bases: `Module`

Stores a discrete deformation field as a $(b, Nx, Ny, 2)$ tensor.

Warps a single image according to an *inverse deformation field* u, i.e. the field that maps the *deformed image* pixel coordinates to the *original image* pixel coordinates.

It is constructed from a tensor of shape $(n\_frames, Nx, Ny, 2)$, where n_frames is the number of frames in the animation, Nx and Ny are the number of pixels along the x-axis and y-axis respectively. The last dimension contains the x and y coordinates of the original image pixel that is displayed in the warped image, at the position corresponding to the indices in the dimension (Nx, Ny) of the tensor.

Note: The coordinates are given in the range [-1;1]. When referring to a pixel, its position is the position of its center.

Note: The position [-1;-1] corresponds to the top-left corner of the top-left pixel if `align_corners` is set to *False* (default), and to the center of the top-left pixel if `align_corners` is set to *True*.

Args:

`inverse_grid_frames` (torch.tensor, optional): *Inverse deformation field* u of shape $(n\_frames, Nx, Ny, 2)$. Default: *None*.

`align_corners` (bool, optional): Option passed to `torch.nn.functional.grid_sample()`. Geometrically, we consider the pixels of the input as squares rather than points. If set to *True*, the extrema (-1 and 1) are considered as referring to the center points of the input's corner pixels. If set to *False*, they are instead considered as referring to the corner points of the input's corner pixels, making the sampling more resolution agnostic. Default: *False*.

Attributes:

`self.inverse_grid_frames` (torch.tensor): *Inverse deformation field* u of shape $(n\_frames, Nx, Ny, 2)$. If set manually, the dtype should be *torch.float32*. Default: *None*.

`self.align_corners` (bool): Should the extrema (-1 and 1) be considered as referring to the center points of the input's corner pixels? Default: *False*.

Example 1: Rotating a 2x2 B&W image by 90 degrees counter-clockwise, using one frame and align_corners=False

```
>>> u = torch.tensor([[[[ 0.5, -0.5], [ 0.5, 0.5]], [[-0.5, -0.5], [-0.5, 0.
↪5]]]])
>>> field = DeformationField(u, align_corners=False)
>>> print(field.inverse_grid_frames)
tensor([[[[ 0.5, -0.5], [ 0.5, 0.5]], [[-0.5, -0.5], [-0.5, 0.5]]]])
```

Example 2: Rotating a 2x2 B&W image by 90 degrees clockwise, using one frame and align_corners=True

```
>>> u = torch.tensor([[[[-1, 1], [-1, -1]], [[ 1, 1], [ 1, -1]]]])
>>> field = DeformationField(u, align_corners=True)
>>> print(field.inverse_grid_frames)
tensor([[[[-1, 1], [-1, -1]], [[ 1, 1], [ 1, -1]]]])
```

Methods

forward(img, n0, n1[, mode])	Warps an image or batch of images with the stored *inverse deformation field* u.
get_inv_grid_frames()	Returns the *inverse deformation field* u.

spyrit.core.time.DeformationField.forward

`DeformationField.`**`forward`**(*img: tensor*, *n0: int*, *n1: int*, *mode: str = 'bilinear'*) → tensor

Warps an image or batch of images with the stored *inverse deformation field* u.

Deforms the image or batch of images according to the *inverse deformation field* u contained in the attribute `inverse_grid_frames`, sliced between the frames $n0$ (included) and $n1$ (excluded). u is the field that maps the pixels of the *deformed image* to the pixels of the *original image*.

Args:

`img` (torch.tensor): The image or batch of images to deform of shape (c, Nx, Ny) or (B, c, Nx, Ny), where B is the number of images in the batch, c is the number of channels (usually 1 or 3), and Nx and Ny are the number of pixels along the x-axis and y-axis respectively.

`n0` (int): The index of the first frame to use in the *inverse deformation field*.

`n1` (int): The index of the first frame to exclude in the *inverse deformation field*.

`mode` (str, optional): The interpolation mode to use. It is directly passed to the function `torch.nn.functional.grid_sample()`. It must be one of the following: 'nearest', 'bilinear', 'bicubic'. Defaults to 'bilinear'.

Important: The input shape must be either (c, Nx, Ny) or (B, c, Nx, Ny).

Note: If $n0 < n1$, `inverse_grid_frames` is sliced as follows: `inv_grid_frames[n0:n1, :, :, :]`

Note: If $n0 > n1$, `inverse_grid_frames` is sliced "backwards". The first frame of the warped animation corresponds to the index $n0$, and the last frame corresponds to the index $n1 + 1$. This behavior is identical to slicing a list with a step of -1.

Returns:

`output` (torch.tensor): The deformed batch of images of shape $(|n1 - n0|, c, Nx, Ny)$ or $(B, |n1 - n0|, c, Nx, Ny)$ depending on the input shape, where each image in the batch is deformed according to the *inverse deformation field* u contained in the attribute `inverse_grid_frames`.

Shape:

`img`: (c, Nx, Ny) or (B, c, Nx, Ny), where B is the batch size, c is the number of channels, and Nx and Ny are the number of pixels along the x-axis and y-axis respectively.

`output`: $(|n1 - n0|, c, Nx, Ny)$ or $(B, |n1 - n0|, c, Nx, Ny)$, depending on the input shape.

Example 1: Rotating a 2x2 B&W image by 90 degrees counter-clockwise, using one frame and align_corners=True

```
>>> v = torch.tensor([[[[ 1., -1.], [ 1., 1.]],
                       [[-1., -1.], [-1., 1.]]]])
>>> field = DeformationField(v, align_corners=True)
>>> image = torch.tensor([[[0. , 0.3],
                           [0.7, 1. ]]])
>>> deformed_image = field(image, 0, 1)
>>> print(deformed_image)
tensor([[[[0.3000, 1.0000],
          [0.0000, 0.7000]]]])
```

spyrit.core.time.DeformationField.get_inv_grid_frames

DeformationField.**get_inv_grid_frames**()

Returns the *inverse deformation field* u.

Returns the *inverse deformation field* u contained in the attribute `inverse_grid_frames`.

Returns:

`self.inverse_grid_frames` (torch.tensor): *Inverse deformation field* u of shape $(n\_frames, Nx, Ny, 2)$.

Example 1: Get the inverse deformation field of a 2x2 B&W image rotated by 90 degrees counter-clockwise

```
>>> u = torch.tensor([[[[ 0.5, -0.5], [ 0.5, 0.5]], [[-0.5, -0.5], [-0.5, 0.
↪5]]]])
>>> field = DeformationField(u, align_corners=False)
>>> print(field.get_inv_grid_frames())
tensor([[[[ 0.5, -0.5], [ 0.5, 0.5]], [[-0.5, -0.5], [-0.5, 0.5]]])
```

spyrit.core.train

Training functions for deep learning models.

Functions

attr_removal(old_key, old_name)	
attr_transformation(old_key, old_name, new_name)	
boxplot(model1, model2, model3, criterion, ...)	
boxplotconsist(model1, model2, model3, ...)	
checkpoint(root, epoch, model)	Saves the dictionaries of a given pytorch model for the right epoch
compare_model(model1, model2, model3, ...[, ...])	Compare three models
count_memory(model)	
count_param(model)	
count_trainable_param(model)	
images_norm(images)	
imshow(img[, title])	
load_net(title, model[, device, strict])	Loads net defined by title
multiplot(train_info1, train_info2, train_info3)	
read_param(path)	
remove_model_attributes(source, old_name[, ...])	Remove some attributes of a saved model (nn.module)
rename_model_attributes(source, old_name, ...)	Rename the name of the attributes of a saved model (nn.module)
save_net(title, model)	Saves dictionaries of a given pytorch model in the place defined by title
tb_profiler(path_prof, model, criterion, ...)	Tensorboard profiler: Profile code execution
tb_writer_add_image(writer, name_metric, ...)	Tensorboard writer: Add an image)
tb_writer_add_scalar(writer, name_metric, ...)	Tensorboard writer: Add a scalar (loss)
tb_writer_init(tb_path[, samples])	Tensorboard log for torch
train_model(model, criterion, optimizer, ...)	Trains the pytorch model
train_model_supervised(model, criterion, ...)	Trains the pytorch model in a supervised way
visualize_conv_layers(conv_layer[, ...])	Displays the 8 first filters of the convolution layer conv_layer
visualize_model(model, dataloaders, device)	Takes 8 images from the dataloader and shows side by side the input image and the reconstructed image

spyrit.core.train.attr_removal

spyrit.core.train.**attr_removal**(*old_key*, *old_name*)

spyrit.core.train.attr_transformation

spyrit.core.train.**attr_transformation**(*old_key*, *old_name*, *new_name*)

spyrit.core.train.boxplot

spyrit.core.train.**boxplot**(*model1*, *model2*, *model3*, *criterion*, *dataloaders*, *device*)

spyrit.core.train.boxplotconsist

spyrit.core.train.**boxplotconsist**(*model1*, *model2*, *model3*, *criterion*, *dataloaders*, *device*)

spyrit.core.train.checkpoint

spyrit.core.train.**checkpoint**(*root*, *epoch*, *model*)

Saves the dictionaries of a given pytorch model for the right epoch

spyrit.core.train.compare_model

spyrit.core.train.**compare_model**(*model1*, *model2*, *model3*, *dataloaders*, *device*, *suptitle=''*, *colormap=<matplotlib.colors.LinearSegmentedColormap object>*)

Compare three models

spyrit.core.train.count_memory

spyrit.core.train.**count_memory**(*model*)

spyrit.core.train.count_param

spyrit.core.train.**count_param**(*model*)

spyrit.core.train.count_trainable_param

spyrit.core.train.**count_trainable_param**(*model*)

spyrit.core.train.images_norm

spyrit.core.train.**images_norm**(*images*)

spyrit.core.train.imshow

spyrit.core.train.**imshow**(*img*, *title=''*)

spyrit.core.train.load_net

spyrit.core.train.**load_net**(*title*, *model*, *device=None*, *strict=True*)

Loads net defined by title

spyrit.core.train.multiplot

spyrit.core.train.**multiplot**(*train_info1*, *train_info2*, *train_info3*, *start=0*)

spyrit.core.train.read_param

spyrit.core.train.**read_param**(*path*)

spyrit.core.train.remove_model_attributes

spyrit.core.train.**remove_model_attributes**(*source*, *old_name*, *target=None*)

Remove some attributes of a saved model (nn.module)

Parameters

- **source** (*str*) – Path to the saved model.
- **old_name** (*str*) – source pattern for the attributes of the model to be removed.
- **target** (*str, optional*) – Path to model with remaned attributes. The default is source.

Returns

None.

Example

Remove the attribute *Denoi* of the model saved as *source*. The resulting model is saved as *target.pth*

```
>>> rename_model_attributes('model.pth', 'Denoi.', 'target.pth')
```

spyrit.core.train.rename_model_attributes

spyrit.core.train.**rename_model_attributes**(*source*, *old_name*, *new_name*, *target=None*)

Rename the name of the attributes of a saved model (nn.module)

Parameters

- **source** (*str*) – Path to the saved model.
- **old_name** (*str*) – source pattern for the attributes of the model to be renamed.
- **new_name** (*str*) – destination pattern for the attributes of the model to be renamed.
- **target** (*str, optional*) – Path to model with remaned attributes. The default is source.

Returns

None.

Example

Rename the key *Denoi.layer.0.weight* and *Denoi.layer.0.weight* as *denoi.layer.0.weight* and *Denoi.layer.0.weight* and save the resulting model as *target.pth*

```
>>> rename_model_attributes('model.pth', 'Denoi.', 'denoi.', 'target.pth')
```

Adapted from https://gist.github.com/the-bass/0bf8aaa302f9ba0d26798b11e4dd73e3

spyrit.core.train.save_net

spyrit.core.train.**save_net**(*title*, *model*)

Saves dictionaries of a given pytorch model in the place defined by title

spyrit.core.train.tb_profiler

spyrit.core.train.**tb_profiler**(*path_prof*, *model*, *criterion*, *optimizer*, *dataloader*, *device*, *wait=1*, *warmup=1*, *active=3*, *repeat=2*)

Tensorboard profiler: Profile code execution

spyrit.core.train.tb_writer_add_image

spyrit.core.train.**tb_writer_add_image**(*writer*, *name_metric*, *images*, *step*)

Tensorboard writer: Add an image)

spyrit.core.train.tb_writer_add_scalar

spyrit.core.train.**tb_writer_add_scalar**(*writer*, *name_metric*, *val_metric*, *step*)

Tensorboard writer: Add a scalar (loss)

spyrit.core.train.tb_writer_init

spyrit.core.train.**tb_writer_init**(*tb_path*, *samples=None*)

Tensorboard log for torch

spyrit.core.train.train_model

spyrit.core.train.**train_model**(*model*, *criterion*, *optimizer*, *scheduler*, *dataloaders*, *device*, *root*, *num_epochs=25*, *disp=False*, *do_checkpoint=0*, *tb_path=False*, *tb_prof=False*, *tb_freq=20*)

Trains the pytorch model

spyrit.core.train.train_model_supervised

spyrit.core.train.**train_model_supervised**(*model*, *criterion*, *optimizer*, *scheduler*, *dataloaders*, *device*, *root*, *num_epochs=25*, *disp=False*, *do_checkpoint=0*)

Trains the pytorch model in a supervised way

spyrit.core.train.visualize_conv_layers

spyrit.core.train.**visualize_conv_layers**(*conv_layer*, *suptitle=''*, *colormap=<matplotlib.colors.LinearSegmentedColormap object>*)

Displays the 8 first filters of the convolution layer conv_layer

spyrit.core.train.visualize_model

spyrit.core.train.**visualize_model**(*model*, *dataloaders*, *device*, *suptitle=''*, *colormap=<matplotlib.colors.LinearSegmentedColormap object>*)

Takes 8 images from the dataloader and shows side by side the input image and the reconstructed image

Classes

Train_par(batch_size, learning_rate, img_size)	
Weight_Decay_Loss(loss)	

spyrit.core.train.Train_par

class spyrit.core.train.**Train_par**(*batch_size*, *learning_rate*, *img_size*, *reg=0*)

Bases: object

Methods

get_loss()	
plot([start])	
plot_log([start])	
set_loss(train_info)	
title()	

spyrit.core.train.Train_par.get_loss

Train_par.**get_loss**()

spyrit.core.train.Train_par.plot

Train_par.**plot**(*start=0*)

spyrit.core.train.Train_par.plot_log

Train_par.**plot_log**(*start=0*)

spyrit.core.train.Train_par.set_loss

Train_par.**set_loss**(*train_info*)

spyrit.core.train.Train_par.title

Train_par.**title**()

spyrit.core.train.Weight_Decay_Loss

class spyrit.core.train.**Weight_Decay_Loss**(*loss*)

Bases: Module

Methods

forward(x, y, net)	Define the computation performed at every call.

spyrit.core.train.Weight_Decay_Loss.forward

Weight_Decay_Loss.**forward**(*x*, *y*, *net*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

2.4.2 spyrit.misc

Modules

spyrit.misc.data_visualisation	Created on Thu Jan 16 08:56:13 2020
spyrit.misc.disp	
spyrit.misc.examples	
spyrit.misc.load_data	Created on Wed Jan 15 17:06:19 2020
spyrit.misc.matrix_tools	Created on Wed Jan 15 16:37:27 2020
spyrit.misc.metrics	
spyrit.misc.pattern_choice	
spyrit.misc.sampling	
spyrit.misc.statistics	
spyrit.misc.walsh_hadamard	Walsh-ordered Hadamard tranforms.

spyrit.misc.data_visualisation

Created on Thu Jan 16 08:56:13 2020

@author: crombez

Functions

plot_im2D(Im[, fig, title, xlabel, ylabel, cmap])	
show_image_and_infos(path, file)	
show_images_infos(path, file)	
simple_plot_2D(Lx, Ly[, fig, title, xlabel, ...])	

spyrit.misc.data_visualisation.plot_im2D

spyrit.misc.data_visualisation.**plot_im2D**(*Im*, *fig=None*, *title=None*, *xlabel=None*, *ylabel=None*, *cmap='viridis'*)

spyrit.misc.data_visualisation.show_image_and_infos

spyrit.misc.data_visualisation.**show_image_and_infos**(*path*, *file*)

spyrit.misc.data_visualisation.show_images_infos

spyrit.misc.data_visualisation.**show_images_infos**(*path*, *file*)

spyrit.misc.data_visualisation.simple_plot_2D

spyrit.misc.data_visualisation.**simple_plot_2D**(*Lx*, *Ly*, *fig=None*, *title=None*, *xlabel=None*, *ylabel=None*, *style_color='b'*)

spyrit.misc.disp

Functions

Multi_plots(img_list, title_list, shape[, ...])	
add_colorbar(mappable[, position, size])	Example:
compare_video_frames(vid_list, ...[, ...])	
display_rgb_vid(video, fps[, title])	video is a numpy array of shape [nb_frames, 3, nx, ny]
display_vid(video, fps[, title, colormap])	video is a numpy array of shape [nb_frames, 1, nx, ny]
fitPlots(N[, aspect])	
histogram(s)	
imagecomp(Img1, Img2[, suptitle, title1, ...])	
imagepanel(Img1, Img2, Img3, Img4[, ...])	
imagesc(Img[, title, colormap, show, ...])	imagesc(IMG) Display image Img with scaled colors with greyscale colormap and colorbar imagesc(IMG, title=ttl) Display image Img with scaled colors with greyscale colormap and colorbar, with the title ttl imagesc(IMG, title=ttl, colormap=cmap) Display image Img with scaled colors with colormap and colorbar specified by cmap (choose between 'plasma', 'jet', and 'grey'), with the title ttl
noaxis(axs)	
plot(x, y[, title, xlabel, ylabel, color])	
pre_process_video(video, crop_patch, kernel_size)	
print_mean_std(x[, tag, prec])	
string_mean_std(x[, prec])	
torch2numpy(torch_tensor)	
uint8(dsp)	
vid2batch(root, img_dim, start_frame, end_frame)	

spyrit.misc.disp.Multi_plots

spyrit.misc.disp.**Multi_plots**(*img_list*, *title_list*, *shape*, *suptitle=''*, *colormap=<matplotlib.colors.LinearSegmentedColormap object>*, *axis_off=True*, *aspect=(16, 9)*, *savefig=''*, *fontsize=14*)

spyrit.misc.disp.add_colorbar

spyrit.misc.disp.**add_colorbar**(*mappable*, *position='right'*, *size='5%'*)

Example:

f, axs = plt.subplots(1, 2) im = axs[0].imshow(img1, cmap='gray') add_colorbar(im) im = axs[0].imshow(img2, cmap='gray') add_colorbar(im)

spyrit.misc.disp.compare_video_frames

spyrit.misc.disp.**compare_video_frames**(*vid_list*, *nb_disp_frames*, *title_list*, *suptitle=''*, *colormap=<matplotlib.colors.LinearSegmentedColormap object>*, *aspect=(16, 9)*, *savefig=''*, *fontsize=14*)

spyrit.misc.disp.display_rgb_vid

spyrit.misc.disp.**display_rgb_vid**(*video*, *fps*, *title=''*)

video is a numpy array of shape [nb_frames, 3, nx, ny]

spyrit.misc.disp.display_vid

spyrit.misc.disp.**display_vid**(*video*, *fps*, *title=''*, *colormap=<matplotlib.colors.LinearSegmentedColormap object>*)

video is a numpy array of shape [nb_frames, 1, nx, ny]

spyrit.misc.disp.fitPlots

spyrit.misc.disp.**fitPlots**(*N*, *aspect=(16, 9)*)

spyrit.misc.disp.histogram

spyrit.misc.disp.**histogram**(*s*)

spyrit.misc.disp.imagecomp

spyrit.misc.disp.**imagecomp**(*Img1*, *Img2*, *suptitle=''*, *title1=''*, *title2=''*, *colormap1=<matplotlib.colors.LinearSegmentedColormap object>*, *colormap2=<matplotlib.colors.LinearSegmentedColormap object>*)

spyrit.misc.disp.imagepanel

spyrit.misc.disp.**imagepanel**(*Img1*, *Img2*, *Img3*, *Img4*, *suptitle=''*, *title1=''*, *title2=''*, *title3=''*, *title4=''*, *colormap1=<matplotlib.colors.LinearSegmentedColormap object>*, *colormap2=<matplotlib.colors.LinearSegmentedColormap object>*, *colormap3=<matplotlib.colors.LinearSegmentedColormap object>*, *colormap4=<matplotlib.colors.LinearSegmentedColormap object>*)

spyrit.misc.disp.imagesc

spyrit.misc.disp.**imagesc**(*Img*, *title=''*, *colormap=<matplotlib.colors.LinearSegmentedColormap object>*, *show=True*, *figsize=None*, *cbar_pos=None*, *title_fontsize=16*)

imagesc(IMG) Display image Img with scaled colors with greyscale colormap and colorbar imagesc(IMG, title=ttl) Display image Img with scaled colors with greyscale colormap and colorbar, with the title ttl imagesc(IMG, title=ttl, colormap=cmap) Display image Img with scaled colors with colormap and colorbar specified by cmap (choose between 'plasma', 'jet', and 'grey'), with the title ttl

spyrit.misc.disp.noaxis

spyrit.misc.disp.**noaxis**(*axs*)

spyrit.misc.disp.plot

spyrit.misc.disp.**plot**(*x*, *y*, *title=''*, *xlabel=''*, *ylabel=''*, *color='black'*)

spyrit.misc.disp.pre_process_video

spyrit.misc.disp.**pre_process_video**(*video*, *crop_patch*, *kernel_size*)

spyrit.misc.disp.print_mean_std

spyrit.misc.disp.**print_mean_std**(*x*, *tag=''*, *prec=3*)

spyrit.misc.disp.string_mean_std

spyrit.misc.disp.**string_mean_std**(*x*, *prec=3*)

spyrit.misc.disp.torch2numpy

spyrit.misc.disp.**torch2numpy**(*torch_tensor*)

spyrit.misc.disp.uint8

spyrit.misc.disp.**uint8**(*dsp*)

spyrit.misc.disp.vid2batch

spyrit.misc.disp.**vid2batch**(*root*, *img_dim*, *start_frame*, *end_frame*)

spyrit.misc.examples

Functions

circle(img_size, R, x_max)
permutation_matrix(A, B)
translation_matrix(img_size, nb_pixels)

spyrit.misc.examples.circle

spyrit.misc.examples.**circle**(*img_size*, *R*, *x_max*)

spyrit.misc.examples.permutation_matrix

spyrit.misc.examples.**permutation_matrix**(*A*, *B*)

spyrit.misc.examples.translation_matrix

spyrit.misc.examples.**translation_matrix**(*img_size*, *nb_pixels*)

spyrit.misc.load_data

Created on Wed Jan 15 17:06:19 2020

@author: crombez

Functions

Files_names(Path, name_type)	
load_data_Comp_1D_new(Path_files, ...)	
load_data_Comp_1D_old(Path_files, ...)	
load_data_recon_3D(Path_files, list_files, ...)	

spyrit.misc.load_data.Files_names

spyrit.misc.load_data.**Files_names**(*Path*, *name_type*)

spyrit.misc.load_data.load_data_Comp_1D_new

spyrit.misc.load_data.**load_data_Comp_1D_new**(*Path_files*, *list_files*, *Nh*, *Nl*, *Nc*)

spyrit.misc.load_data.load_data_Comp_1D_old

spyrit.misc.load_data.**load_data_Comp_1D_old**(*Path_files*, *list_files*, *Nh*, *Nl*, *Nc*)

spyrit.misc.load_data.load_data_recon_3D

spyrit.misc.load_data.**load_data_recon_3D**(*Path_files*, *list_files*, *Nl*, *Nc*, *Nh*)

spyrit.misc.matrix_tools

Created on Wed Jan 15 16:37:27 2020

@author: crombez

Functions

Permutation_Matrix(mat)	Returns permutation matrix from sampling matrix
Sum_coll(Mat, N_lin, N_coll)	
clean_out(Data, Nl, Nc, Nh[, m])	
compression_1D(H, Nl, Nc, Nh)	
data_conv_hadamard(H, Data, N)	
expend_vect(Vect, N1, N2)	
normalize_by_median_mat_2D(Mat)	
normalize_mat_2D(Mat)	
reject_outliers(data[, m])	
remove_offset_mat_2D(Mat)	
resize(Mat, Nl, Nc, Nh)	
smooth(y, box_pts)	
stack_depth_matrice(Mat, Nl, Nc, Nd)	

spyrit.misc.matrix_tools.Permutation_Matrix

spyrit.misc.matrix_tools.**Permutation_Matrix**(*mat*)

Returns permutation matrix from sampling matrix

Args:

Mat (np.ndarray):
N-by-N sampling matrix, where high values indicate high significance.

Returns:
P (np.ndarray): N^2-by-N^2 permutation matrix (boolean)

Warning: This function is a duplicate of *spyrit.misc.sampling.Permutation_Matrix()* and will be removed in a future release.

Note: Consider using `sort_by_significance()` for increased computational performance if using *Permutation_Matrix()* to reorder a matrix as follows: `y = Permutation_Matrix(Ord) @ Mat`

spyrit.misc.matrix_tools.Sum_coll

spyrit.misc.matrix_tools.**Sum_coll**(*Mat*, *N_lin*, *N_coll*)

spyrit.misc.matrix_tools.clean_out

spyrit.misc.matrix_tools.**clean_out**(*Data*, *Nl*, *Nc*, *Nh*, *m=2*)

spyrit.misc.matrix_tools.compression_1D

spyrit.misc.matrix_tools.**compression_1D**(*H*, *Nl*, *Nc*, *Nh*)

spyrit.misc.matrix_tools.data_conv_hadamard

spyrit.misc.matrix_tools.**data_conv_hadamard**(*H*, *Data*, *N*)

spyrit.misc.matrix_tools.expend_vect

spyrit.misc.matrix_tools.**expend_vect**(*Vect*, *N1*, *N2*)

spyrit.misc.matrix_tools.normalize_by_median_mat_2D

spyrit.misc.matrix_tools.**normalize_by_median_mat_2D**(*Mat*)

spyrit.misc.matrix_tools.normalize_mat_2D

spyrit.misc.matrix_tools.**normalize_mat_2D**(*Mat*)

spyrit.misc.matrix_tools.reject_outliers

spyrit.misc.matrix_tools.**reject_outliers**(*data*, *m=2*)

spyrit.misc.matrix_tools.remove_offset_mat_2D

spyrit.misc.matrix_tools.**remove_offset_mat_2D**(*Mat*)

spyrit.misc.matrix_tools.resize

spyrit.misc.matrix_tools.**resize**(*Mat*, *Nl*, *Nc*, *Nh*)

spyrit.misc.matrix_tools.smooth

spyrit.misc.matrix_tools.**smooth**(*y*, *box_pts*)

spyrit.misc.matrix_tools.stack_depth_matrice

spyrit.misc.matrix_tools.**stack_depth_matrice**(*Mat*, *Nl*, *Nc*, *Nd*)

spyrit.misc.metrics

Functions

batch_psnr(torch_batch, output_batch)	
batch_psnr_vid(input_batch, output_batch)	
batch_ssim(torch_batch, output_batch)	
batch_ssim_vid(input_batch, output_batch)	
compare_nets_unsupervised(net_list, ...)	
compare_video_nets_supervised(net_list, ...)	
dataset_meas(dataloader, model, device)	
dataset_psnr(dataloader, model, device)	
dataset_psnr_ssim(dataloader, model, device)	
dataset_psnr_ssim_fcl(dataloader, model, device)	
dataset_ssim(dataloader, model, device)	
print_mean_std(x[, tag])	
psnr(I1, I2)	Computes the psnr between two images I1 and I2
*psnr_*(img1, img2[, r])	Computes the psnr between two image with values expected in a given range
ssim(I1, I2)	Computes the ssim between two images I1 and I2

spyrit.misc.metrics.batch_psnr

spyrit.misc.metrics.**batch_psnr**(*torch_batch*, *output_batch*)

spyrit.misc.metrics.batch_psnr_vid

spyrit.misc.metrics.**batch_psnr_vid**(*input_batch*, *output_batch*)

spyrit.misc.metrics.batch_ssim

spyrit.misc.metrics.**batch_ssim**(*torch_batch*, *output_batch*)

spyrit.misc.metrics.batch_ssim_vid

spyrit.misc.metrics.**batch_ssim_vid**(*input_batch*, *output_batch*)

spyrit.misc.metrics.compare_nets_unsupervised

spyrit.misc.metrics.**compare_nets_unsupervised**(*net_list*, *testloader*, *device*)

spyrit.misc.metrics.compare_video_nets_supervised

spyrit.misc.metrics.**compare_video_nets_supervised**(*net_list*, *testloader*, *device*)

spyrit.misc.metrics.dataset_meas

spyrit.misc.metrics.**dataset_meas**(*dataloader*, *model*, *device*)

spyrit.misc.metrics.dataset_psnr

spyrit.misc.metrics.**dataset_psnr**(*dataloader*, *model*, *device*)

spyrit.misc.metrics.dataset_psnr_ssim

spyrit.misc.metrics.**dataset_psnr_ssim**(*dataloader*, *model*, *device*)

spyrit.misc.metrics.dataset_psnr_ssim_fcl

spyrit.misc.metrics.**dataset_psnr_ssim_fcl**(*dataloader*, *model*, *device*)

spyrit.misc.metrics.dataset_ssim

spyrit.misc.metrics.**dataset_ssim**(*dataloader*, *model*, *device*)

spyrit.misc.metrics.print_mean_std

spyrit.misc.metrics.**print_mean_std**(*x*, *tag=''*)

spyrit.misc.metrics.psnr

spyrit.misc.metrics.**psnr**(*I1*, *I2*)

Computes the psnr between two images I1 and I2

spyrit.misc.metrics.psnr_

spyrit.misc.metrics.**psnr_**(*img1*, *img2*, *r=2*)

Computes the psnr between two image with values expected in a given range

Args:

img1, img2 (np.ndarray): images r (float): image range

Returns:

Psnr (float): Peak signal-to-noise ratio

spyrit.misc.metrics.ssim

spyrit.misc.metrics.**ssim**(*I1*, *I2*)

Computes the ssim between two images I1 and I2

spyrit.misc.pattern_choice

Functions

Daubechies(def_matrix[, par, lvl])	
Daubechies_opt(M[, par, lvl])	
Fourier(def_matrix[, par, lvl])	
Fourier_opt(M[, par, lvl])	
Haar(def_matrix[, par, lvl])	
Haar_opt(M[, par, lvl])	
Hadamard(def_matrix[, par, lvl])	
Hadamard_opt(M[, par, lvl])	
matrix2conv(Matrix)	Returns Convulution filter che each kernel correponds to a line of Matrix, that has been reshaped
shift(Q, dyn)	
split(Q, dyn)	

spyrit.misc.pattern_choice.Daubechies

spyrit.misc.pattern_choice.**Daubechies**(*def_matrix*, *par=2*, *lvl=1*)

spyrit.misc.pattern_choice.Daubechies_opt

spyrit.misc.pattern_choice.**Daubechies_opt**(*M*, *par=2*, *lvl=1*)

spyrit.misc.pattern_choice.Fourier

spyrit.misc.pattern_choice.**Fourier**(*def_matrix*, *par=0*, *lvl=1*)

spyrit.misc.pattern_choice.Fourier_opt

spyrit.misc.pattern_choice.**Fourier_opt**(*M*, *par=0*, *lvl=1*)

spyrit.misc.pattern_choice.Haar

spyrit.misc.pattern_choice.**Haar**(*def_matrix*, *par=0*, *lvl=1*)

spyrit.misc.pattern_choice.Haar_opt

spyrit.misc.pattern_choice.**Haar_opt**(*M*, *par=0*, *lvl=1*)

spyrit.misc.pattern_choice.Hadamard

spyrit.misc.pattern_choice.**Hadamard**(*def_matrix*, *par=0*, *lvl=1*)

spyrit.misc.pattern_choice.Hadamard_opt

spyrit.misc.pattern_choice.**Hadamard_opt**(*M*, *par=0*, *lvl=1*)

spyrit.misc.pattern_choice.matrix2conv

spyrit.misc.pattern_choice.**matrix2conv**(*Matrix*)

Returns Convulution filter che each kernel correponds to a line of Matrix, that has been reshaped

spyrit.misc.pattern_choice.shift

spyrit.misc.pattern_choice.**shift**(*Q*, *dyn*)

spyrit.misc.pattern_choice.split

spyrit.misc.pattern_choice.**split**(*Q*, *dyn*)

Classes

Basis_patterns(img_size, basis[, method, ...])	
Custom_patterns(img_size, Q[, method, binarized])	
Optimized_patterns(img_size, basis, M[, ...])	
Patterns(img_size[, method, binarized, par, ...])	

spyrit.misc.pattern_choice.Basis_patterns

class spyrit.misc.pattern_choice.**Basis_patterns**(*img_size*, *basis*, *method='split'*, *binarized=False*)

Bases: *Patterns*

Methods

add_desired_pattern(def_matrix)
add_desired_patterns(def_matrix)
get_all_desired_pattern()
get_desired_pattern()
get_measurement_matrix()
save_measurement_matrix(root)
set_desired_pattern(def_matrix)
set_measurement_matrix()

spyrit.misc.pattern_choice.Basis_patterns.add_desired_pattern

Basis_patterns.**add_desired_pattern**(*def_matrix*)

spyrit.misc.pattern_choice.Basis_patterns.add_desired_patterns

abstract Basis_patterns.**add_desired_patterns**(*def_matrix*)

spyrit.misc.pattern_choice.Basis_patterns.get_all_desired_pattern

Basis_patterns.**get_all_desired_pattern**()

spyrit.misc.pattern_choice.Basis_patterns.get_desired_pattern

Basis_patterns.**get_desired_pattern**()

spyrit.misc.pattern_choice.Basis_patterns.get_measurement_matrix

Basis_patterns.**get_measurement_matrix**()

spyrit.misc.pattern_choice.Basis_patterns.save_measurement_matrix

Basis_patterns.**save_measurement_matrix**(*root*)

spyrit.misc.pattern_choice.Basis_patterns.set_desired_pattern

Basis_patterns.**set_desired_pattern**(*def_matrix*)

spyrit.misc.pattern_choice.Basis_patterns.set_measurement_matrix

Basis_patterns.**set_measurement_matrix**()

spyrit.misc.pattern_choice.Custom_patterns

class spyrit.misc.pattern_choice.**Custom_patterns**(*img_size*, *Q*, *method='split'*, *binarized=False*)

Bases: *Patterns*

Methods

add_desired_pattern(Q)
add_desired_patterns(def_matrix)
get_all_desired_pattern()
get_desired_pattern()
get_measurement_matrix()
save_measurement_matrix(root)
set_desired_pattern(Q)
set_measurement_matrix()

spyrit.misc.pattern_choice.Custom_patterns.add_desired_pattern

Custom_patterns.**add_desired_pattern**(*Q*)

spyrit.misc.pattern_choice.Custom_patterns.add_desired_patterns

abstract Custom_patterns.**add_desired_patterns**(*def_matrix*)

spyrit.misc.pattern_choice.Custom_patterns.get_all_desired_pattern

Custom_patterns.**get_all_desired_pattern**()

spyrit.misc.pattern_choice.Custom_patterns.get_desired_pattern

Custom_patterns.**get_desired_pattern**()

spyrit.misc.pattern_choice.Custom_patterns.get_measurement_matrix

Custom_patterns.**get_measurement_matrix**()

spyrit.misc.pattern_choice.Custom_patterns.save_measurement_matrix

Custom_patterns.**save_measurement_matrix**(*root*)

spyrit.misc.pattern_choice.Custom_patterns.set_desired_pattern

Custom_patterns.**set_desired_pattern**(*Q*)

spyrit.misc.pattern_choice.Custom_patterns.set_measurement_matrix

Custom_patterns.**set_measurement_matrix**()

spyrit.misc.pattern_choice.Optimized_patterns

class spyrit.misc.pattern_choice.**Optimized_patterns**(*img_size*, *basis*, *M*, *method='split'*, *binarized=False*)

Bases: *Patterns*

Methods

add_desired_pattern(M_prim)	
add_desired_patterns(def_matrix)	
get_all_desired_pattern()	
get_desired_pattern()	
get_measurement_matrix()	
save_measurement_matrix(root)	
set_desired_pattern(M)	
set_measurement_matrix()	

spyrit.misc.pattern_choice.Optimized_patterns.add_desired_pattern

Optimized_patterns.**add_desired_pattern**(*M_prim*)

spyrit.misc.pattern_choice.Optimized_patterns.add_desired_patterns

abstract Optimized_patterns.**add_desired_patterns**(*def_matrix*)

spyrit.misc.pattern_choice.Optimized_patterns.get_all_desired_pattern

Optimized_patterns.**get_all_desired_pattern**()

spyrit.misc.pattern_choice.Optimized_patterns.get_desired_pattern

Optimized_patterns.**get_desired_pattern**()

spyrit.misc.pattern_choice.Optimized_patterns.get_measurement_matrix

Optimized_patterns.**get_measurement_matrix**()

spyrit.misc.pattern_choice.Optimized_patterns.save_measurement_matrix

Optimized_patterns.**save_measurement_matrix**(*root*)

spyrit.misc.pattern_choice.Optimized_patterns.set_desired_pattern

Optimized_patterns.**set_desired_pattern**(*M*)

spyrit.misc.pattern_choice.Optimized_patterns.set_measurement_matrix

Optimized_patterns.**set_measurement_matrix**()

spyrit.misc.pattern_choice.Patterns

class spyrit.misc.pattern_choice.**Patterns**(*img_size*, *method='split'*, *binarized=False*, *par=2*, *lvl=1*, *dyn=8*)

Bases: ABC

Methods

add_desired_patterns(def_matrix)
get_all_desired_pattern()
get_desired_pattern()
get_measurement_matrix()
save_measurement_matrix(root)
set_desired_pattern(def_matrix)
set_measurement_matrix()

spyrit.misc.pattern_choice.Patterns.add_desired_patterns

abstract Patterns.**add_desired_patterns**(*def_matrix*)

spyrit.misc.pattern_choice.Patterns.get_all_desired_pattern

Patterns.**get_all_desired_pattern**()

spyrit.misc.pattern_choice.Patterns.get_desired_pattern

Patterns.**get_desired_pattern**()

spyrit.misc.pattern_choice.Patterns.get_measurement_matrix

Patterns.**get_measurement_matrix**()

spyrit.misc.pattern_choice.Patterns.save_measurement_matrix

Patterns.**save_measurement_matrix**(*root*)

spyrit.misc.pattern_choice.Patterns.set_desired_pattern

abstract Patterns.**set_desired_pattern**(*def_matrix*)

spyrit.misc.pattern_choice.Patterns.set_measurement_matrix

Patterns.**set_measurement_matrix**()

spyrit.misc.sampling

Functions

Permutation_Matrix(Mat)	Returns permutation matrix from sampling matrix
img2mask(Mat, M)	Returns sampling mask from sampling matrix.
img2meas(Img, Mat)	Return measurement vector from measurement image (not TESTED)
meas2img(meas, Mat)	Return measurement image from a single measurement vector
meas2img2(meas, Mat)	Return multiple measurement images from multiple measurement vectors.
reorder(meas, Perm_acq, Perm_rec)	Reorder measurement vectors
sort_by_indices(arr, indices[, axis, ...])	Returns an array ordered by the given indices along the specified axis.
sort_by_significance(arr, sig[, axis, ...])	Returns an array ordered by decreasing significance along the specified dimension.

spyrit.misc.sampling.Permutation_Matrix

spyrit.misc.sampling.**Permutation_Matrix**(*Mat: ndarray*) → ndarray

Returns permutation matrix from sampling matrix

Args:

Mat (np.ndarray):
N-by-N sampling matrix, where high values indicate high significance.

Returns:
P (np.ndarray): N^2-by-N^2 permutation matrix (boolean)

Note: Consider using *sort_by_significance()* for increased computational performance if using *Permutation_Matrix()* to reorder a matrix as follows: `y = Permutation_Matrix(Ord) @ Mat`

---

spyrit.misc.sampling.img2mask

spyrit.misc.sampling.**img2mask**(*Mat: ndarray*, *M: int*)

Returns sampling mask from sampling matrix.

Args:

Mat (np.ndarray):
N-by-N sampling matrix, where high values indicate high significance.

M (int):
Number of measurements to be kept.

Returns:

Mask (np.ndarray):
N-by-N sampling mask, where 1 indicates the measurements to sample and 0 that to discard.

spyrit.misc.sampling.img2meas

spyrit.misc.sampling.**img2meas**(*Img: ndarray*, *Mat: ndarray*) → ndarray

Return measurement vector from measurement image (not TESTED)

Args:

Img (np.ndarray):
N-by-N measurement image.

Mat (np.ndarray):
N-by-N sampling matrix, where high values indicate high significance.

Returns:

meas (np.ndarray):
Measurement vector of lenth M <= N**2.

spyrit.misc.sampling.meas2img

spyrit.misc.sampling.**meas2img**(*meas: ndarray*, *Mat: ndarray*) → ndarray

Return measurement image from a single measurement vector

Args:

meas
[*np.ndarray* with shape $(M,)$] Measurement vector of length $M \leq N^2$.

Mat
[*np.ndarray* with shape (N, N)] Sampling matrix, where high values indicate high significance.

Returns:

Img
[*np.ndarray* with shape $(N, N,)$] N-by-N measurement image

spyrit.misc.sampling.meas2img2

spyrit.misc.sampling.**meas2img2**(*meas: ndarray*, *Mat: ndarray*) → ndarray

Return multiple measurement images from multiple measurement vectors. It is essentially the same as *meas2img*, but the *meas* argument is two-dimensional.

Args:

meas
[*np.ndarray* with shape (M, B)] Set of B measurement vectors of lenth $M \leq N^2$.

Mat
[*np.ndarray* with shape (N, N)] Sampling matrix, where high values indicate high significance.

Returns:

Img
[*np.ndarray* with shape (N, N, B)] Set of B images of shape (N, N)

spyrit.misc.sampling.reorder

spyrit.misc.sampling.**reorder**(*meas: ndarray*, *Perm_acq: ndarray*, *Perm_rec: ndarray*) → ndarray

Reorder measurement vectors

Args:

meas (np.ndarray):
Measurements with dimensions ($M_{acq} \times K_{rep}$), where M_{acq} is the number of acquired patterns and K_{rep} is the number of acquisition repetitions (e.g., wavelength or image batch).

Perm_acq (np.ndarray):
Permutation matrix used for acquisition ($N_{acq}^2 \times N_{acq}^2$ square matrix).

Perm_rec (np.ndarray):
Permutation matrix used for reconstruction ($N_{rec} \times N_{rec}$ square matrix).

Returns:

(np.ndarray):
Measurements with dimensions ($M_{rec} \times K_{rep}$), where $M_{rec} = N_{rec}^2$.

Note: If $M_{rec} < M_{acq}$, the input measurement vectors are subsampled.

If $M_{rec} > M_{acq}$, the input measurement vectors are filled with zeros.

spyrit.misc.sampling.sort_by_indices

spyrit.misc.sampling.**sort_by_indices**(*arr: ndarray | tensor*, *indices: ndarray | tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*) → ndarray | tensor

Returns an array ordered by the given indices along the specified axis.

The indices are used to reorder the array along the specified axis. The indices give the order in which the elements should be placed. The type of the output is the same as the input array arr.

Args:

arr (np.ndarray or torch.tensor):
Array to be ordered by rows or columns. The output's type is the same as this parameter's type.

indices (np.ndarray or torch.tensor):
Array containing the indices of the elements in the order they should be placed.

axis (str, optional):
Axis along which to order the array. Must be either "rows" or "cols". Defaults to "rows".

inverse_permutation (bool, optional):
If True, the permutation matrix is transposed before being used. Defaults to False.

Raises:

ValueError:
If axis is not "rows" or "cols".

ValueError:
If the number of rows or columns in x is not equal to the length of the indices.

Returns:

np.ndarray or torch.tensor:
Array ordered by the given indices along the specified axis. The type is the same as the input array arr.

Example:

```
>>> arr = [[10, 20, 30], [100, 200, 300]]
>>> indices = [2, 0, 1]
>>> sort_by_indices(arr, indices, axis="cols")
array([[ 30,  10,  20],
       [300, 100, 200]])
```

spyrit.misc.sampling.sort_by_significance

spyrit.misc.sampling.**sort_by_significance**(*arr: ndarray | tensor*, *sig: ndarray | tensor*, *axis: str = 'rows'*, *inverse_permutation: bool = False*, *get_indices: bool = False*) → ndarray | tensor

Returns an array ordered by decreasing significance along the specified dimension.

The significance values are given in the sig array. The type of the output is the same as the input array arr.

This function is equivalent to (but faster) *Permutation_Matrix()* and multiplying the input array by the permutation matrix. More specifically, here are the four possible different calls and their equivalent:

```
h = 64
arr = np.random.randn(h, h)
sig = np.random.randn(h)

# 1
y = sort_by_significance(arr, sig, axis='rows', inverse_permutation=False)
y = Permutation_Matrix(sig) @ arr

# 2
y = sort_by_significance(arr, sig, axis='rows', inverse_permutation=True)
y = Permutation_Matrix(sig).T @ arr

# 3
y = sort_by_significance(arr, sig, axis='cols', inverse_permutation=False)
y = arr @ Permutation_Matrix(sig)

# 4
y = sort_by_significance(arr, sig, axis='cols', inverse_permutation=True)
y = arr @ Permutation_Matrix(sig).T
```

Note: arr must have the same number of rows or columns as there are elements in the flattened sig array.

Args:

arr (np.ndarray or torch.tensor):
Array to be ordered by rows or columns. The output's type is the same as this parameter's type.

sig (np.ndarray or torch.tensor):
Array containing the significance values.

axis (str, optional):
Axis along which to order the array. Must be either 'rows' or 'cols'. Defaults to 'rows'.

inverse_permutation (bool, optional):
If True, the permutation matrix is transposed before being used. This is equivalent to using the inverse permutation matrix. Defaults to False.

get_indices (bool, optional):
If True, the function returns the indices of the significance values in decreasing order. Defaults to False.

Shape:

- arr: $(*, r, c)$ or (c), where $(*)$ is any

number of dimensions, and r and c are the number of rows and columns respectively.

- sig: (r) if axis is 'rows' or (c) if axis is 'cols'

(or any shape that has the same number of elements). Not used if arr is 1D.

- Output: $(*, r, c)$ or (c)

Returns:

Tuple of np.ndarray or torch.tensor:

- **Array arr ordered by decreasing significance sig**
  along its rows or columns.
- **Indices $indices$ of the significance values in decreasing**
  order. This is useful if you want to reorder other arrays in the same way.

spyrit.misc.statistics

Functions

Cov2Var(Cov)	Extracts Variance Matrix from Covariance Matrix
cov_walsh(dataloader, mean, device[, n_loop])	nloop > 1 is relevant for dataloaders with random crops such as that provided by data_loaders_ImageNet
data_loaders_ImageNet(train_root[, ...])	Args:
data_loaders_stl10(data_root[, img_size, ...])	Args:
img2mask(Ord, M)	Returns subsampling mask from order matrix
mea_abs_model(dataloader, device, model, root)	
mean_walsh(dataloader, device[, n_loop])	nloop > 1 is relevant for dataloaders with random crops such as that provided by data_loaders_ImageNet
optim_had(dataloader, root)	Computes image that ranks the hadamard coefficients
stat_fwalsh_S(dataloader, device, root)	
stat_fwalsh_S_stl10([stat_root, data_root, ...])	Fast Walsh S-transform of X in "2D"
stat_mean_coef_from_model(dataloader, ...)	
stat_model(dataloader, device, model, root)	
stat_walsh(dataloader, device, root[, n_loop])	nloop > 1 is relevant for dataloaders with random crops such as that provided by data_loaders_ImageNet
stat_walsh_ImageNet([stat_root, data_root, ...])	Args:
stat_walsh_stl10([stat_root, data_root, ...])	Args:
transform_gray_norm(img_size)	Args:

spyrit.misc.statistics.Cov2Var

spyrit.misc.statistics.**Cov2Var**(*Cov*)

Extracts Variance Matrix from Covariance Matrix

spyrit.misc.statistics.cov_walsh

spyrit.misc.statistics.**cov_walsh**(*dataloader*, *mean*, *device*, *n_loop=1*)

nloop > 1 is relevant for dataloaders with random crops such as that provided by data_loaders_ImageNet

spyrit.misc.statistics.data_loaders_ImageNet

spyrit.misc.statistics.**data_loaders_ImageNet**(*train_root*, *val_root=None*, *img_size=64*, *batch_size=512*, *seed=7*, *shuffle=False*)

Args:

Both 'train_root' and 'val_root' need to have images in a subfolder shuffle=True to shuffle train set only (test set not shuffled)

The output of torchvision datasets are PILImage images in the range [0, 1]. We transform them to Tensors in the range [-1, 1]. Also RGB images are converted into grayscale images.

spyrit.misc.statistics.data_loaders_stl10

spyrit.misc.statistics.**data_loaders_stl10**(*data_root*, *img_size=64*, *batch_size=512*, *seed=7*, *shuffle=False*, *download=True*)

Args:

shuffle=True to shuffle train set only (test set not shuffled)

The output of torchvision datasets are PILImage images in the range [0, 1]. We transform them to Tensors in the range [-1, 1]. Also RGB images are converted into grayscale images.

spyrit.misc.statistics.img2mask

spyrit.misc.statistics.**img2mask**(*Ord*, *M*)

Returns subsampling mask from order matrix

spyrit.misc.statistics.mea_abs_model

spyrit.misc.statistics.**mea_abs_model**(*dataloader*, *device*, *model*, *root*)

spyrit.misc.statistics.mean_walsh

spyrit.misc.statistics.**mean_walsh**(*dataloader*, *device*, *n_loop=1*)

nloop > 1 is relevant for dataloaders with random crops such as that provided by data_loaders_ImageNet

spyrit.misc.statistics.optim_had

spyrit.misc.statistics.**optim_had**(*dataloader*, *root*)

Computes image that ranks the hadamard coefficients

spyrit.misc.statistics.stat_fwalsh_S

spyrit.misc.statistics.**stat_fwalsh_S**(*dataloader*, *device*, *root*)

spyrit.misc.statistics.stat_fwalsh_S_stl10

spyrit.misc.statistics.**stat_fwalsh_S_stl10**(*stat_root=PosixPath('stats')*, *data_root=PosixPath('data')*, *img_size=64*, *batch_size=1024*)

Fast Walsh S-transform of X in “2D”

Args:

`X` (torch.tensor): input image with shape *(*, n, n)*. `n`**2 should be a power of two.

Returns:

torch.tensor: S-transformed signal with shape *(*, n, n)*

Examples:

```
>>> import spyrit.misc.statistics as st
>>> st.stat_fwalsh_S_stl10()
```

spyrit.misc.statistics.stat_mean_coef_from_model

spyrit.misc.statistics.**stat_mean_coef_from_model**(*dataloader*, *device*, *model_exp*)

spyrit.misc.statistics.stat_model

spyrit.misc.statistics.**stat_model**(*dataloader*, *device*, *model*, *root*)

spyrit.misc.statistics.stat_walsh

spyrit.misc.statistics.**stat_walsh**(*dataloader*, *device*, *root*, *n_loop=1*)

nloop > 1 is relevant for dataloaders with random crops such as that provided by data_loaders_ImageNet

spyrit.misc.statistics.stat_walsh_ImageNet

spyrit.misc.statistics.**stat_walsh_ImageNet**(*stat_root=PosixPath('stats')*, *data_root=PosixPath('data/ILSVRC2012_img_test_v10102019')*, *img_size=128*, *batch_size=256*, *n_loop=1*, *device=device(type='cpu')*)

Args:

data_root needs to have all images in a subfolder

Example:

```
>>> from pathlib import Path
>>> from spyrit.misc.statistics import stat_walsh_ImageNet
>>> data_root =  Path('../data/ILSVRC2012_v10102019')
>>> stat_root =  Path('../stat/ILSVRC2012_v10102019')
>>> stat_walsh_ImageNet(stat_root = stat_root, data_root = data_root, img_size␣
↪= 32, batch_size = 1024)
```

spyrit.misc.statistics.stat_walsh_stl10

spyrit.misc.statistics.**stat_walsh_stl10**(*stat_root=PosixPath('stats')*, *data_root=PosixPath('data')*, *img_size=64*, *batch_size=1024*, *device=device(type='cpu')*)

Args:

data_root is expected to contain an 'stl10_binary' subfolder with the test*.bin, train*.bin and unlabeled_X.bin files.

Example:

```
>>> data_root =  Path('../datasets/')
>>> stat_root =  Path('../stat/stl10')
>>> from spyrit.misc.statistics import stat_walsh_stl10
>>> stat_walsh_stl10(stat_root = stat_root, data_root = data_root)
```

spyrit.misc.statistics.transform_gray_norm

spyrit.misc.statistics.**transform_gray_norm**(*img_size*)

Args:

img_size=int, image size

Create torchvision transform for natural images (stl10, imagenet): convert them to grayscale, then to tensor, and normalize between [-1, 1]

Classes

CenterCrop(img_size)	Args:

spyrit.misc.statistics.CenterCrop

class spyrit.misc.statistics.**CenterCrop**(*img_size*)

Bases: object

Args:

img_size=int, image size

Center crop if image not square in order to ensure that all images have same size

spyrit.misc.walsh_hadamard

Walsh-ordered Hadamard tranforms.

Longer description of this module.

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Functions

b2_to_b10(l)	Convert a list of numbers in base 2 to base 10
bit_reverse_traverse(a)	
bit_reversed_list(n)	
bit_reversed_matrix(n)	
fwalsh2_S(X[, ind])	Fast Walsh S-transform of X in "2D"
fwalsh2_S_torch(X[, ind])	Fast Walsh S-transform of X in "2D"
fwalsh_G(x[, ind])	Fast Walsh G-transform of x
fwalsh_G_torch(x[, ind])	Fast Walsh G-transform of x
fwalsh_S(x[, ind])	Fast Walsh S-transform of x
fwalsh_S_torch(x[, ind])	Fast Walsh S-transform of x
fwht(x[, order])	Fast Walsh-Hadamard transform of x
fwht_torch(x[, order])	Fast Walsh-Hadamard transform of x
get_bit_reversed_list(l)	

continues on next page

Table 1 – continued from previous page

gray_code_list(n)	
gray_code_permutation(n)	
ifwalsh2_S(Y[, ind])	Inverse Fast Walsh S-transform of Y in "2D"
ifwalsh_S(s[, ind])	Inverse fast Walsh S-transform of s
iwalsh2(X[, H])	Return 2D inverse Walsh-ordered Hadamard transform of an image
iwalsh2_S(Y[, T])	Inverse Fast Walsh S-transform of Y in "2D"
iwalsh_S(s[, T])	Return the inverse Walsh S-transform of s
iwalsh_S_matrix(n[, H])	Return inverse Walsh S-matrix of order n
perm_matrix_from_ind(l)	
sequency_perm(X[, ind])	Permute the rows of a matrix to get sequency order
sequency_perm_ind(n)	Return permutation indices to get sequency from the natural order
sequency_perm_matrix(n)	Return permutation matrix to get sequency from the natural order
sequency_perm_torch(X[, ind])	Permute the last dimension of a tensor to get sequency order
walsh2(X[, H])	Return 2D Walsh-ordered Hadamard transform of an image $H^\top X H$
walsh2_S(X[, S])	Fast Walsh S-transform of X in "2D"
walsh2_S_fold(x)	Fold a signal to get a "2d" s-transformed representation
walsh2_S_fold_torch(x)	Fold a signal to get a "2d" s-transformed representation
walsh2_S_matrix(n)	Return Walsh S-matrix in "2d"
walsh2_S_unfold(X)	Unfold a signal from a "2d" s-transformed representation
walsh2_S_unfold_torch(X)	Unfold a signal from a "2d" s-transformed representation
walsh2_matrix(n)	Return Walsh-ordered Hadamard matrix in 2D
walsh2_torch(im[, H])	Return 2D Walsh-ordered Hadamard transform of an image
walsh_G(x[, G])	Return the Walsh S-transform of x
walsh_G_matrix(n[, H])	Return Walsh-ordered Hadamard S-matrix of order n
walsh_S(x[, S])	Return the Walsh S-transform of x
walsh_S_matrix(n[, H])	Return Walsh S-matrix of order n
walsh_matrix(n)	Return 1D Walsh-ordered Hadamard transform matrix
walsh_torch(x[, H])	Return 1D Walsh-ordered Hadamard transform of a signal

spyrit.misc.walsh_hadamard.b2_to_b10

spyrit.misc.walsh_hadamard.**b2_to_b10**(*l*)

Convert a list of numbers in base 2 to base 10

Args:

l (list[str]): base2 numbers.

Returns:

list[int]: base10 numbers

spyrit.misc.walsh_hadamard.bit_reverse_traverse

spyrit.misc.walsh_hadamard.**bit_reverse_traverse**(*a*)

spyrit.misc.walsh_hadamard.bit_reversed_list

spyrit.misc.walsh_hadamard.**bit_reversed_list**(*n*)

spyrit.misc.walsh_hadamard.bit_reversed_matrix

spyrit.misc.walsh_hadamard.**bit_reversed_matrix**(*n*)

spyrit.misc.walsh_hadamard.fwalsh2_S

spyrit.misc.walsh_hadamard.**fwalsh2_S**(*X*, *ind=True*)

Fast Walsh S-transform of X in "2D"

Args:

x (np.ndarray): n-by-n signal. n**2 should be a power of two. ind (bool, optional): True for sequency (default) ind (list, optional): permutation indices.

Returns:

np.ndarray: n-by-1 S-transformed signal

Examples:

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import numpy as np
>>> X = np.array([[1, 3, 0, 8],[7, 5, 1, 2],[2, 3, 6, 1],[4, 6, 8, 0]])
>>> wh.fwalsh2_S(X)
```

spyrit.misc.walsh_hadamard.fwalsh2_S_torch

spyrit.misc.walsh_hadamard.**fwalsh2_S_torch**(*X*, *ind=True*)

Fast Walsh S-transform of X in "2D"

Args:

X (torch.tensor): input image with shape *(*, n, n)*. `n`2**
should be a power of two.

ind (bool, optional): True for sequency (default) ind (list, optional): permutation indices.

Returns:

torch.tensor: S-transformed signal with shape *(*, n, n)*

Examples:

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> X = torch.tensor([[1, 3, 0, 8],[7, 5, 1, 2],[2, 3, 6, 1],[4, 6, 8, 0]])
>>> wh.fwalsh2_S_torch(X)
```

Example 2:

Repeating the Walsh-ordered S-transform using input indices is faster

```
>>> import timeit
>>> import torch
>>> import spyrit.misc.walsh_hadamard as wh
>>> X = torch.rand(128, 2**6, 2**6, device=torch.device('cuda:0'))
>>> t = timeit.timeit(lambda: wh.fwalsh2_S_torch(X), number=10)
>>> print(f"No indices as inputs (10x): {t:.3f} seconds")
>>> ind = wh.sequency_perm_ind(X.shape[-1]*X.shape[-2])
>>> t = timeit.timeit(lambda: wh.fwalsh2_S_torch(X,ind), number=10)
>>> print(f"With indices as inputs (10x): {t:.3f} seconds")
```

spyrit.misc.walsh_hadamard.fwalsh_G

spyrit.misc.walsh_hadamard.**fwalsh_G**(*x*, *ind=True*)

Fast Walsh G-transform of x

Args:

`x` (np.ndarray): n-by-1 signal. n+1 should be a power of two.

`ind` (bool, optional): True for sequency (default)

`ind` (list, optional): permutation indices. This is faster than True when repeating the sequency-ordered transform multilple times.

Returns:

np.ndarray: n-by-1 S-transformed signal

Example 1:

Walsh-ordered G-transform of a signal of length 7

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import numpy as np
>>> x = np.array([1, 3, 0, -1, 7, 5, 1])
>>> s = wh.fwalsh_G(x)
>>> print(s)
```

Example 2:

Permuted fast G-transform

```
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.array([1, 3, 0, -1, 7, 5, 1])
>>> ind = [1, 0, 3, 2, 7, 4, 5, 6]
>>> y = wh.fwalsh_G(x, ind)
>>> print(y)
```

Example 3:

Repeating the Walsh-ordered G-transform using input indices is faster

```
>>> import timeit
>>> x = np.random.rand(2**12-1,1)
>>> t = timeit.timeit(lambda: wh.fwalsh_G(x), number=10)
>>> print(f"No indices as inputs (10x): {t:.3f} seconds")
```

(continues on next page)

(continued from previous page)

```
>>> ind = wh.sequency_perm_ind(len(x)+1)
>>> t = timeit.timeit(lambda: wh.fwalsh_G(x,ind), number=10)
>>> print(f"With indices as inputs (10x): {t:.3f} seconds")
```

Example 4:

Comparison with G-transform via matrix-vector product

```
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.array([3, 0, -1, 7, 5, 1, -2])
>>> y1 = wh.fwalsh_G(x)
>>> y2 = wh.walsh_G(x)
>>> print(f"Diff = {np.linalg.norm(y1-y2)}")
```

spyrit.misc.walsh_hadamard.fwalsh_G_torch

spyrit.misc.walsh_hadamard.**fwalsh_G_torch**(*x*, *ind=True*)

Fast Walsh G-transform of x

Args:

x (torch.tensor): input signal with shape $(*, n)$. :math:`n`+1 should be a power of two.

ind (bool, optional): True for sequency (default)

ind (list, optional): permutation indices. This is faster than True when repeating the sequency-ordered transform multilple times.

Returns:

torch.tensor: S-transformed signal with shape $(*, n)$.

Example 1:

Walsh-ordered G-transform of a signal of length 7

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import torch
>>> x = torch.tensor([1, 3, 0, -1, 7, 5, 1])
>>> s = wh.fwalsh_G_torch(x)
>>> print(s)
```

Example 2:

Permuted fast G-transform

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import torch
>>> x = torch.tensor([1, 3, 0, -1, 7, 5, 1])
>>> ind = [0, 1, 3, 2, 7, 4, 5, 6]
>>> y = wh.fwalsh_G_torch(x, ind)
>>> print(y)
```

Example 3:

Comparison with the numpy transform

```
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.array([1, 3, 0, -1, 7, 5, 1])
>>> y_np = wh.fwalsh_G(x)
>>> x_torch = torch.from_numpy(x).to(torch.device('cuda:0'))
>>> y_torch = wh.fwalsh_G_torch(x_torch)
>>> print(y_np)
>>> print(y_torch)
```

Example 3:
Repeating the Walsh-ordered G-transform using input indices is faster

```
>>> import timeit
>>> x = torch.rand(512,2**12-1, device=torch.device('cuda:0'))
>>> t = timeit.timeit(lambda: wh.fwalsh_G_torch(x), number=10)
>>> print(f"No indices as inputs (10x): {t:.3f} seconds")
>>> ind = wh.sequency_perm_ind(x.shape[-1]+1)
>>> t = timeit.timeit(lambda: wh.fwalsh_G_torch(x,ind), number=10)
>>> print(f"With indices as inputs (10x): {t:.3f} seconds")
```

spyrit.misc.walsh_hadamard.fwalsh_S

spyrit.misc.walsh_hadamard.**fwalsh_S**(*x*, *ind=True*)

Fast Walsh S-transform of x

Args:
`x` (np.ndarray): n-by-1 signal. n+1 should be a power of two.

`ind` (bool, optional): True for sequency (default)

`ind` (list, optional): permutation indices. This is faster than True
when repeating the sequency-ordered transform multilple times.

Returns:
np.ndarray: n-by-1 S-transformed signal

Example 1:
Walsh-ordered S-transform of a signal of length 7

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import numpy as np
>>> x = np.array([1, 3, 0, -1, 7, 5, 1])
>>> s = wh.fwalsh_S(x)
>>> print(s)
```

Example 2:
Permuted fast S-transform

```
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.array([1, 3, 0, -1, 7, 5, 1])
>>> ind = [1, 0, 3, 2, 7, 4, 5, 6]
>>> y = wh.fwalsh_S(x, ind)
>>> print(y)
```

Example 3:

Repeating the Walsh-ordered S-transform using input indices is faster

```
>>> import timeit
>>> x = np.random.rand(2**12-1,1)
>>> t = timeit.timeit(lambda: wh.fwalsh_S(x), number=10)
>>> print(f"No indices as inputs (10x): {t:.3f} seconds")
>>> ind = wh.sequency_perm_ind(len(x)+1)
>>> t = timeit.timeit(lambda: wh.fwalsh_S(x,ind), number=10)
>>> print(f"With indices as inputs (10x): {t:.3f} seconds")
```

Example 4:

Comparison with S-transform via matrix-vector product

```
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.array([3, 0, -1, 7, 5, 1, -2])
>>> y1 = wh.fwalsh_S(x)
>>> y2 = wh.walsh_S(x)
>>> print(f"Diff = {np.linalg.norm(y1-y2)}")
```

Example 5:

Computation times for a signal of length 2**12-1

```
>>> import timeit
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.random.rand(2**14-1,1)
>>> t = timeit.timeit(lambda: wh.fwalsh_S(x), number=10)
>>> print(f"Fast transform, no ind (10x): {t:.3f} seconds")
>>> ind = wh.sequency_perm_ind(len(x)+1)
>>> t = timeit.timeit(lambda: wh.fwalsh_S(x,ind), number=10)
>>> print(f"Fast transform, with ind (10x): {t:.3f} seconds")
>>> S = wh.walsh_S_matrix(len(x))
>>> t = timeit.timeit(lambda: wh.walsh_S(x,S), number=10)
>>> print(f"Naive transform (10x): {t:.3f} seconds")
```

spyrit.misc.walsh_hadamard.fwalsh_S_torch

spyrit.misc.walsh_hadamard.**fwalsh_S_torch**(*x*, *ind=True*)

Fast Walsh S-transform of x

Args:

x (torch.tensor): input signal with shape *(*, n)*. `n`+1
should be a power of two.

ind (bool, optional): True for sequency (default)

ind (list, optional): permutation indices. This is faster than True
when repeating the sequency-ordered transform multilple times.

Returns:

torch.tensor: -by-n S-transformed signal

Example 1:

Walsh-ordered S-transform of a signal of length 7

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import torch
>>> x = torch.tensor([1, 3, 0, -1, 7, 5, 1])
>>> s = wh.fwalsh_S_torch(x)
>>> print(s)
```

Example 2:

Repeating the Walsh-ordered S-transform using input indices is faster

```
>>> import timeit
>>> import torch
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = torch.rand(512, 2**14-1, device=torch.device('cuda:0'))
>>> t = timeit.timeit(lambda: wh.fwalsh_S_torch(x), number=10)
>>> print(f"No indices as inputs (10x): {t:.3f} seconds")
>>> ind = wh.sequency_perm_ind(x.shape[-1]+1)
>>> t = timeit.timeit(lambda: wh.fwalsh_S_torch(x,ind), number=10)
>>> print(f"With indices as inputs (10x): {t:.3f} seconds")
```

spyrit.misc.walsh_hadamard.fwht

spyrit.misc.walsh_hadamard.**fwht**(*x*, *order=True*)

Fast Walsh-Hadamard transform of x

Args:

x (np.ndarray): n-by-1 input signal, where n is a power of two. order (bool, optional): True for sequency (default), False for natural. order (list, optional): permutation indices.

Returns:

np.ndarray: n-by-1 transformed signal

Example 1:

Fast sequency-ordered (i.e., Walsh) Hadamard transform

```
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.array([1, 3, 0, -1, 7, 5, 1, -2])
>>> y = wh.fwht(x)
>>> print(y)
```

Example 2:

Fast Hadamard transform

```
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.array([1, 3, 0, -1, 7, 5, 1, -2])
>>> y = wh.fwht(x, False)
>>> print(y)
```

Example 3:

Permuted fast Hadamard transform

```
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.array([1, 3, 0, -1, 7, 5, 1, -2])
>>> ind = [1, 0, 3, 2, 7, 4, 5, 6]
>>> y = wh.fwht(x, ind)
>>> print(y)
```

Example 4:

Comparison with Walsh-Hadamard transform via matrix-vector product

```
>>> from spyrit.misc.walsh_hadamard import fwht, walsh_matrix
>>> x = np.array([1, 3, 0, -1, 7, 5, 1, -2])
>>> y1 = fwht(x)
>>> H = walsh_matrix(8)
>>> y2 = H @ x
>>> print(f"Diff = {np.linalg.norm(y1-y2)}")
```

Example 5:

Comparison with the fast Walsh-Hadamard transform from sympy >>> import spyrit.misc.walsh_hadamard as wh >>> import sympy as sp >>> x = np.array([1, 3, 0, -1, 7, 5, 1, -2]) >>> y1 = wh.fwht(x) >>> y2 = sp.fwht(x) >>> y3 = wh.sequency_perm(np.array(y2)) >>> print(f"Diff = {np.linalg.norm(y1-y3)}")

Example 6:

Computation times for a signal of length 2**12

```
>>> import timeit
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.random.rand(2**12,1)
>>> t = timeit.timeit(lambda: wh.fwht(x), number=10)
>>> print(f"Fast Walsh transform, no ind (10x): {t:.3f} seconds")
>>> ind = wh.sequency_perm_ind(len(x))
>>> t = timeit.timeit(lambda: wh.fwht(x,ind), number=10)
>>> print(f"Fast Walsh transform, with ind (10x): {t:.3f} seconds")
>>> t = timeit.timeit(lambda: wh.fwht(x,False), number=10)
>>> print(f"Fast Hadamard transform (10x): {t:.3f} seconds")
>>> import sympy as sp
>>> t = timeit.timeit(lambda: sp.fwht(x), number=10)
>>> print(f"Fast Hadamard transform from sympy (10x): {t:.3f} seconds")
```

spyrit.misc.walsh_hadamard.fwht_torch

spyrit.misc.walsh_hadamard.**fwht_torch**(*x*, *order=True*)

Fast Walsh-Hadamard transform of x

Args:

x (np.ndarray): -by-n input signal, where n is a power of two. order (bool, optional): True for sequency (default), False for natural. order (list, optional): permutation indices.

Returns:

np.ndarray: n-by-1 transformed signal

Example 1:

Fast sequency-ordered (i.e., Walsh) Hadamard transform

```
>>> import torch
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = torch.tensor([1, 3, 0, -1, 7, 5, 1, -2])
>>> x = x[None,:]
>>> y = wh.fwht_torch(x)
>>> print(y)
```

Example 2:

Fast Hadamard transform

```
>>> import torch
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = torch.tensor([1, 3, 0, -1, 7, 5, 1, -2])
>>> x = x[None,:]
>>> y = wh.fwht_torch(x, False)
>>> print(y)
```

Example 3:

Permuted fast Hadamard transform

```
>>> import numpy as np
>>> import torch
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = torch.tensor([1, 3, 0, -1, 7, 5, 1, -2])
>>> ind = [1, 0, 3, 2, 7, 4, 5, 6]
>>> y = wh.fwht_torch(x, ind)
>>> print(y)
```

Example 4:

Comparison with the numpy transform

```
>>> import numpy as np
>>> import torch
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.array([1, 3, 0, -1, 7, 5, 1, -2])
>>> y_np = wh.fwht(x)
>>> x_torch = torch.from_numpy(x).to(torch.device('cuda:0'))
>>> y_torch = wh.fwht_torch(x_torch)
>>> print(y_np)
>>> print(y_torch)
```

Example 5:

Computation times for a signal of length 2**12

```
>>> import timeit
>>> import torch
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = np.random.rand(2**12,1)
>>> t = timeit.timeit(lambda: wh.fwht(x,False), number=200)
>>> print(f"Fast Hadamard transform numpy CPU (200x): {t:.4f} seconds")
>>> x_torch = torch.from_numpy(x)
>>> t = timeit.timeit(lambda: wh.fwht_torch(x_torch,False), number=200)
>>> print(f"Fast Hadamard transform pytorch CPU (200x): {t:.4f} seconds")
>>> x_torch = torch.from_numpy(x).to(torch.device('cuda:0'))
>>> t = timeit.timeit(lambda: wh.fwht_torch(x_torch,False), number=200)
>>> print(f"Fast Hadamard transform pytorch GPU (200x): {t:.4f} seconds")
```

Example 6:

CPU vs GPU: Computation times for 512 signals of length 2**12

```
>>> import timeit
>>> import torch
>>> import spyrit.misc.walsh_hadamard as wh
>>> x_cpu = torch.rand(512,2**12)
>>> t = timeit.timeit(lambda: wh.fwht_torch(x_cpu,False), number=10)
>>> print(f"Fast Hadamard transform pytorch CPU (10x): {t:.4f} seconds")
>>> x_gpu = x_cpu.to(torch.device('cuda:0'))
>>> t = timeit.timeit(lambda: wh.fwht_torch(x_gpu,False), number=10)
>>> print(f"Fast Hadamard transform pytorch GPU (10x): {t:.4f} seconds")
```

Example 7:

Repeating the Walsh-ordered transform using input indices is faster

```
>>> import timeit
>>> import torch
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = torch.rand(256,2**12).to(torch.device('cuda:0'))
>>> t = timeit.timeit(lambda: wh.fwht_torch(x), number=100)
>>> print(f"No indices as inputs (100x): {t:.3f} seconds")
>>> ind = wh.sequency_perm_ind(x.shape[-1])
>>> t = timeit.timeit(lambda: wh.fwht_torch(x,ind), number=100)
>>> print(f"With indices as inputs (100x): {t:.3f} seconds")
```

spyrit.misc.walsh_hadamard.get_bit_reversed_list

spyrit.misc.walsh_hadamard.**get_bit_reversed_list**(*l*)

spyrit.misc.walsh_hadamard.gray_code_list

spyrit.misc.walsh_hadamard.**gray_code_list**(*n*)

spyrit.misc.walsh_hadamard.gray_code_permutation

spyrit.misc.walsh_hadamard.**gray_code_permutation**(*n*)

spyrit.misc.walsh_hadamard.ifwalsh2_S

spyrit.misc.walsh_hadamard.**ifwalsh2_S**(*Y*, *ind=True*)

Inverse Fast Walsh S-transform of Y in "2D"

Args:

Y (np.ndarray): n-by-n signal. n**2 should be a power of two.

ind (bool, optional): True for sequency (default)

ind (list, optional): permutation indices.

Returns:

np.ndarray: n-by-1 S-transformed signal

Examples:

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> X = np.array([[1, 3, 0, 8],[7, 5, 1, 2],[2, 3, 6, 1],[4, 6, 8, 0]])
>>> print(f"image:\n {X}")
>>> Y = wh.fwalsh2_S(X)
>>> print(f"s-transform:\n {Y}")
>>> Z = wh.ifwalsh2_S(Y)
>>> print(f"inverse s-transform:\n {Z}")
```

Note that the first pixel is not meaningful and arbitrily set to 0.

spyrit.misc.walsh_hadamard.ifwalsh_S

spyrit.misc.walsh_hadamard.**ifwalsh_S**(*s*, *ind=True*)

Inverse fast Walsh S-transform of s

Args:

x (np.ndarray): n-by-1 signal. n+1 should be a power of two.

ind (bool, optional): True for sequency (default).

ind (list, optional): permutation indices. This is faster than True
when repeating the sequency-ordered transform multilple times.

Returns:
np.ndarray: n-by-1 inverse transformed signal

Examples:
Inverse S-transform of a 15-by-1 signal

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import numpy as np
>>> x = np.array([1, 3, 0, -1, 7, 5, 1])
>>> print(f"signal: {x}")
>>> s = wh.fwalsh_S(x)
>>> print(f"s-transform: {s}")
>>> y = wh.ifwalsh_S(s)
>>> print(f"inverse s-transform: {y}")
```

spyrit.misc.walsh_hadamard.iwalsh2

spyrit.misc.walsh_hadamard.**iwalsh2**(*X*, *H=None*)

Return 2D inverse Walsh-ordered Hadamard transform of an image

Args:
X (np.ndarray): Image as a 2D array. The image is square and its size is a power of two. H (np.ndarray, optional): 1D inverse Walsh-ordered Hadamard transformation matrix

Returns:
np.ndarray: Inverse Hadamard transformed image as a 2D array.

spyrit.misc.walsh_hadamard.iwalsh2_S

spyrit.misc.walsh_hadamard.**iwalsh2_S**(*Y*, *T=None*)

Inverse Fast Walsh S-transform of Y in “2D”

Args:
Y (np.ndarray): n-by-n signal. n**2 should be a power of two.

T (np.ndarray): Inverse S-matrix

Returns:
np.ndarray: n-by-1 S-transformed signal

Examples:

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import numpy as np
>>> X = np.array([[1, 3, 0, 8],[7, 5, 1, 2],[2, 3, 6, 1],[4, 6, 8, 0]])
>>> print(f"image:\n {X}")
>>> Y = wh.walsh2_S(X)
>>> print(f"s-transform:\n {Y}")
>>> Z = wh.iwalsh2_S(Y)
>>> print(f"inverse s-transform:\n {Z}")
```

Note that the first pixel is not meaningful and arbitrily set to 0.

spyrit.misc.walsh_hadamard.iwalsh_S

spyrit.misc.walsh_hadamard.**iwalsh_S**(*s*, *T=None*)

Return the inverse Walsh S-transform of s

Args:
x (np.ndarray): n-by-1 signal. n+1 should be a power of two.

Returns:
np.ndarray: n-by-1 inverse transformed signal

Examples:
Inverse S-transform of a 4095-by-1 signal

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import numpy as np
>>> x = np.random.rand(4095,1)
>>> s = wh.walsh_S(x)
>>> y = wh.iwalsh_S(s)
>>> err = np.linalg.norm(1-y/x)
>>> print(f'Error: {err}')
```

spyrit.misc.walsh_hadamard.iwalsh_S_matrix

spyrit.misc.walsh_hadamard.**iwalsh_S_matrix**(*n*, *H=None*)

Return inverse Walsh S-matrix of order n

Args:
n (int): Matrix order. n+1 should be a power of two.

Returns:
np.ndarray: n-by-n array

Example 1:
Inverse of the Walsh S-matrix of order 7

```
>>> print(iwalsh_S_matrix(7))
```

Example 2:
Check the inverse of the Walsh S-matrix of order 7 >>> print(iwalsh_S_matrix(7) @ walsh_S_matrix(7))

spyrit.misc.walsh_hadamard.perm_matrix_from_ind

spyrit.misc.walsh_hadamard.**perm_matrix_from_ind**(*l*)

spyrit.misc.walsh_hadamard.sequency_perm

spyrit.misc.walsh_hadamard.**sequency_perm**(*X*, *ind=None*)

Permute the rows of a matrix to get sequency order

Args:

X (np.ndarray): n-by-m input matrix

ind : index list of length n

Returns:

np.ndarray: n-by-m input matrix

spyrit.misc.walsh_hadamard.sequency_perm_ind

spyrit.misc.walsh_hadamard.**sequency_perm_ind**(*n*)

Return permutation indices to get sequency from the natural order

Args:

n (int): Order of the matrix, a power of two.

Returns:

list:

Examples:

Permutation indices to get a Walsh matrix of order 8

```
>>> print(sequency_perm_ind(8))
```

spyrit.misc.walsh_hadamard.sequency_perm_matrix

spyrit.misc.walsh_hadamard.**sequency_perm_matrix**(*n*)

Return permutation matrix to get sequency from the natural order

Args:

n (int): Order of the matrix, a power of two.

Returns:

np.ndarray: A n-by-n permutation matrix

Examples:

Permutation matrix of order 8

```
>>> print(sequency_perm_matrix(8))
```

spyrit.misc.walsh_hadamard.sequency_perm_torch

spyrit.misc.walsh_hadamard.**sequency_perm_torch**(*X*, *ind=None*)

Permute the last dimension of a tensor to get sequency order

Args:

`X` (torch.tensor): -by-n input matrix

`ind` : index list of length n

Returns:

torch.tensor: -by-n input matrix

Example :

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = torch.tensor([1, 3, 0, -1, 7, 5, 1, -2])
>>> x = x[None, None, :]
>>> x = wh.sequency_perm_torch(x)
>>> print(x)
```

spyrit.misc.walsh_hadamard.walsh2

spyrit.misc.walsh_hadamard.**walsh2**(*X*, *H=None*)

Return 2D Walsh-ordered Hadamard transform of an image $H^\top X H$

Args:

X (np.ndarray): image as a 2d array. The size is a power of two. H (np.ndarray, optional): 1D Walsh-ordered Hadamard transformation matrix

Returns:

np.ndarray: Hadamard transformed image as a 2D array.

spyrit.misc.walsh_hadamard.walsh2_S

spyrit.misc.walsh_hadamard.**walsh2_S**(*X*, *S=None*)

Fast Walsh S-transform of X in “2D”

Args:

x (np.ndarray): n-by-n signal. n**2 should be a power of two. ind (bool, optional): True for sequency (default) ind (list, optional): permutation indices.

Returns:

np.ndarray: n-by-1 S-transformed signal

Examples:

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import numpy as np
>>> X = np.array([[1, 3, 0, 8],[7, 5, 1, 2],[2, 3, 6, 1],[4, 6, 8, 0]])
>>> wh.walsh2_S(X)
```

spyrit.misc.walsh_hadamard.walsh2_S_fold

spyrit.misc.walsh_hadamard.**walsh2_S_fold**(*x*)

Fold a signal to get a "2d" s-transformed representation

Note: the top left (first) pixel is arbitrarily set to zero

Args:

x (np.ndarray): N-by- vector. N is such that N+1 = n*n, where n is a power of two. N = 2**(2b) - 1, where b is an integer

Returns:

X (np.ndarray): n-by-n matrix

Example 1:

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> S = wh.walsh2_S_matrix(4)
>>> X = wh.walsh2_S_fold(S[2,:])
>>> print(X)
```

spyrit.misc.walsh_hadamard.walsh2_S_fold_torch

spyrit.misc.walsh_hadamard.**walsh2_S_fold_torch**(*x*)

Fold a signal to get a "2d" s-transformed representation

Note: the top left (first) pixel is arbitrarily set to zero

Args:

x (torch.tensor): input signal with shape *(*, n)*. n is such that n+1 = N*N, where N is a power of two. n = 2**(2b) - 1, where b is an integer.

Returns:

torch.tensor: output matrix with shape *(*, N, N)*

Example 1:

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import torch
>>> S = wh.walsh2_S_matrix(4)
>>> X = torch.from_numpy(S[2:4,:])
>>> Y = wh.walsh2_S_fold_torch(X)
>>> print(Y)
```

spyrit.misc.walsh_hadamard.walsh2_S_matrix

spyrit.misc.walsh_hadamard.**walsh2_S_matrix**(*n*)

Return Walsh S-matrix in "2d"

Args:

n (int): Order of the matrix. n must be a power of two.

Returns:

S (np.ndarray): (n*n-1)-by-(n*n-1) matrix

Example 1:

```
>>> S = walsh2_S_matrix(4)
```

spyrit.misc.walsh_hadamard.walsh2_S_unfold

spyrit.misc.walsh_hadamard.**walsh2_S_unfold**(*X*)

Unfold a signal from a "2d" s-transformed representation

Note: the top left (first) pixel is arbitrarily set to zero

Args:
X (np.ndarray): n-by-m image.

Returns:
X (np.ndarray): (n*n-1)-by-1 signal

Example 1:

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> X = np.array([[1, 3, 0, 8],[7, 5, 1, 2]])
>>> wh.walsh2_S_unfold(X)
```

spyrit.misc.walsh_hadamard.walsh2_S_unfold_torch

spyrit.misc.walsh_hadamard.**walsh2_S_unfold_torch**(*X*)

Unfold a signal from a "2d" s-transformed representation

Note: Return a view of X

Args:
X (torch.tensor): input image with shape *(*, n,n)*.

Returns:
output signal with shape *(*, n*n-1)*

Example 1:

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import torch
>>> X = torch.tensor([[1, 3, 0, 8],[7, 5, 1, 2]])
>>> x = wh.walsh2_S_unfold_torch(X)
>>> print(X)
>>> print(x)
```

Example 2:

```
>>> import spyrit.misc.walsh_hadamard as wh
>>> import torch
>>> X = torch.randint(10,(3,4,4))
>>> x = wh.walsh2_S_unfold_torch(X)
>>> print(X)
>>> print(x)
```

spyrit.misc.walsh_hadamard.walsh2_matrix

spyrit.misc.walsh_hadamard.**walsh2_matrix**(*n*)

Return Walsh-ordered Hadamard matrix in 2D

Args:

n (int): Order of the matrix, which must be a power of two.

Returns:

H (np.ndarray): A n*n-by-n*n matrix

spyrit.misc.walsh_hadamard.walsh2_torch

spyrit.misc.walsh_hadamard.**walsh2_torch**(*im*, *H=None*)

Return 2D Walsh-ordered Hadamard transform of an image

Args:

im (torch.tensor): Image, typically a B-by-C-by-W-by-H Tensor H (torch.tensor, optional): 1D Walsh-ordered Hadamard transformation matrix. A 2-D tensor of size W-by-H.

Returns:

torch.tensor: Hadamard transformed image. Same size as im

Examples:

```
>>> im = torch.randn(256, 1, 64, 64)
>>> had = walsh2_torch(im)
```

spyrit.misc.walsh_hadamard.walsh_G

spyrit.misc.walsh_hadamard.**walsh_G**(*x*, *G=None*)

Return the Walsh S-transform of x

Args:

x (np.ndarray): n-by-1 signal. n+1 should be a power of two.

Returns:

np.ndarray: n-by-1 S-transformed signal

Examples:

Walsh-ordered S-transform of a 15-by-1 signal

```
>>> x = np.random.rand(15,1)
>>> s = walsh_S(x)
```

spyrit.misc.walsh_hadamard.walsh_G_matrix

spyrit.misc.walsh_hadamard.**walsh_G_matrix**(*n*, *H=None*)

Return Walsh-ordered Hadamard S-matrix of order n

Args:
n (int): Matrix order. n+1 should be a power of two. H (np.ndarray, optional):

Returns:
np.ndarray: n-by-n array

Examples:
Walsh-ordered Hadamard G-matrix of order 7

```
>>> print(walsh_G_matrix(7))
```

spyrit.misc.walsh_hadamard.walsh_S

spyrit.misc.walsh_hadamard.**walsh_S**(*x*, *S=None*)

Return the Walsh S-transform of x

Args:
x (np.ndarray): n-by-1 signal. n+1 should be a power of two.

Returns:
np.ndarray: n-by-1 S-transformed signal

Examples:
Walsh-ordered S-transform of a 15-by-1 signal

```
>>> x = np.random.rand(15,1)
>>> s = walsh_S(x)
```

spyrit.misc.walsh_hadamard.walsh_S_matrix

spyrit.misc.walsh_hadamard.**walsh_S_matrix**(*n*, *H=None*)

Return Walsh S-matrix of order n

Args:
n (int): Matrix order. n+1 should be a power of two.

Returns:
np.ndarray: n-by-n array

Examples:
Walsh-ordered Hadamard S-matrix of order 7

```
>>> print(walsh_S_matrix(7))
```

spyrit.misc.walsh_hadamard.walsh_matrix

spyrit.misc.walsh_hadamard.**walsh_matrix**(*n*)

Return 1D Walsh-ordered Hadamard transform matrix

Args:

n (int): Order of the matrix, a power of two.

Returns:

np.ndarray: A n-by-n array

Examples:

Walsh-ordered Hadamard matrix of order 8

```
>>> print(walsh_matrix(8))
```

spyrit.misc.walsh_hadamard.walsh_torch

spyrit.misc.walsh_hadamard.**walsh_torch**(*x*, *H=None*)

Return 1D Walsh-ordered Hadamard transform of a signal

Args:

`x` (`torch.tensor`): Input signals with shape *(*, n)*.

`H` (`torch.tensor`, optional): 1D Walsh-ordered Hadamard matrix with shape *(*, m)*.

Returns:

`torch.tensor`: Hadamard transformed signals with shape *(*, m)*.

Note:

Providing the input argument `H` leads to much faster computation when multiple Hadamard transforms are repeated (see Example 2).

Example 1:

Sequency-ordered (i.e., Walsh) Hadamard transform

```
>>> import torch
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = torch.tensor([1.0, 3.0, 0.0, -1.0, 7.0, 5.0, 1.0, -2.0])
>>> y = wh.fwht_torch(x)
>>> print(y)
>>> y = wh.walsh_torch(x)
>>> print(y)
```

Example 2:

Fast vs regular: Computation times for 5 batches of 512 signals of length 2**10

```
>>> import timeit
>>> import torch
>>> import numpy as np
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = torch.rand(5, 1, 512, 2**10)
>>> t = timeit.timeit(lambda: wh.fwht_torch(x), number=200)
>>> print(f"Fast Hadamard transform (200x): {t:.4f} seconds")
>>> t = timeit.timeit(lambda: torch.from_numpy(walsh_matrix(x.shape[-1]).astype(
```

(continues on next page)

(continued from previous page)

```
↪'float32')), number=200)
>>> print(f"Construction of Hadamard matrix (200x): {t:.4f} seconds")
>>> H = torch.from_numpy(walsh_matrix(x.shape[-1]).astype('float32'))
>>> t = timeit.timeit(lambda: wh.walsh_torch(x, H), number=200)
>>> print(f"Matrix-vector products (200x): {t:.4f} seconds")
```

Example 3:
CPU vs GPU: Computation times for 5 batches of 512 signals of length 2**10

```
>>> import timeit
>>> import torch
>>> import spyrit.misc.walsh_hadamard as wh
>>> x = torch.rand(5, 1, 512, 2**10)
>>> H = torch.tensor(walsh_matrix(x.shape[-1]), dtype=torch.float32)
>>> t = timeit.timeit(lambda: wh.walsh_torch(x, H), number=200)
>>> print(f"Fast Hadamard transform pytorch CPU (200x): {t:.4f} seconds")
>>> x = x.to(torch.device('cuda:0'))
>>> H = H.to(torch.device('cuda:0'))
>>> t = timeit.timeit(lambda: wh.walsh_torch(x, H), number=200)
>>> print(f"Fast Hadamard transform pytorch GPU (200x): {t:.4f} seconds")
```

2.4.3 Tutorials

Here you can find a series of Tutorials that will guide you through the use of Spyrit. It is recommended to follow them in order.

For each tutorial, please download the corresponding Python Script (*.py*) or Jupyter notebook (*.ipynb*) file at the bottom of the page. The images used in these tutorials can be found on this page of the Spyrit GitHub.

Below is a diagram of the entire image processing pipeline. Each tutorial focuseson a specific part of the pipeline.

- *Tutorial 1* focuses

on the measurement operators, with or without noise

- *Tutorial 2* explains

the pseudo-inverse reconstruction process from the (possibly noisy) measurements

- *Tutorial 3* uses

a CNN to denoise the image if necessary

- *Tutorial 4*

is used to train the CNN introduced in Tutorial 3

- *Tutorial 5*

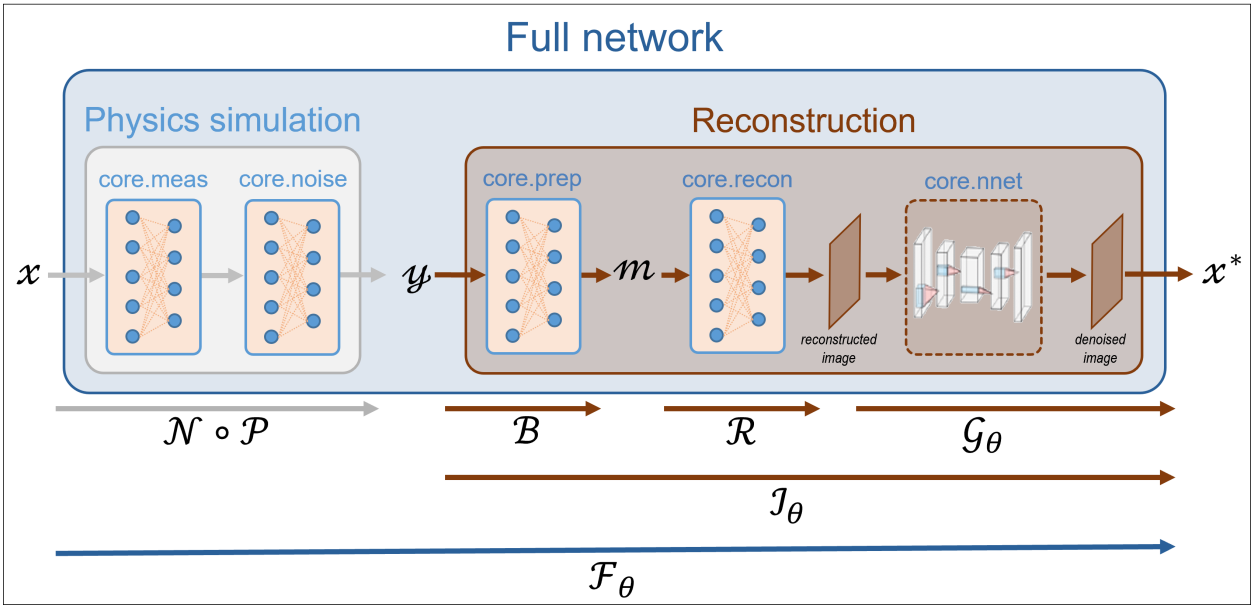
introduces a new type of measurement operator ('split') that simulates positive and negative measurements

- *Tutorial 6* uses

a Denoised Completion Network with a trainable image denoiser to improve the results obtained in Tutorial 5

- Explore *Bonus Tutorial*

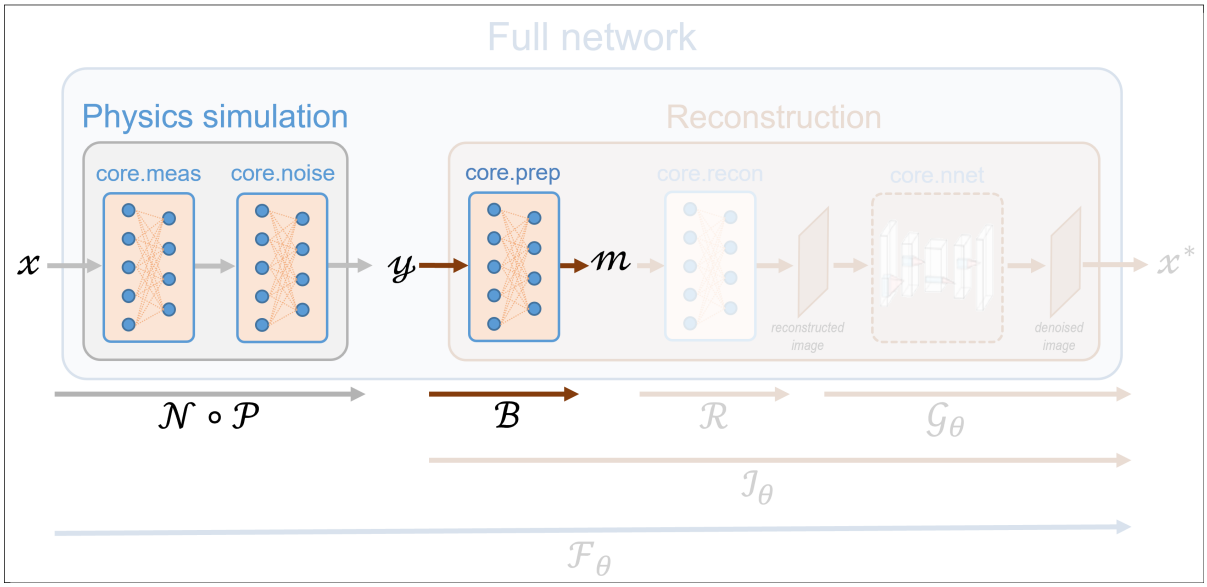
if you want to go deeper into Spyrit's capabilities



01. Acquisition operators

This tutorial shows how to simulate measurements using the `spyrit.core` submodule. The simulation is based on three modules:

1. **Measurement operators** compute linear measurements $y = Hx$ from images x, where H is a linear operator (matrix) and x is a vectorized image (see `spyrit.core.meas`)
2. **Noise operator** corrupts measurements y with noise (see `spyrit.core.noise`)
3. **Preprocessing operators** are typically used to process the noisy measurements prior to reconstruction (see `spyrit.core.prep`)



These tutorials load image samples from */images/*.

Load a batch of images

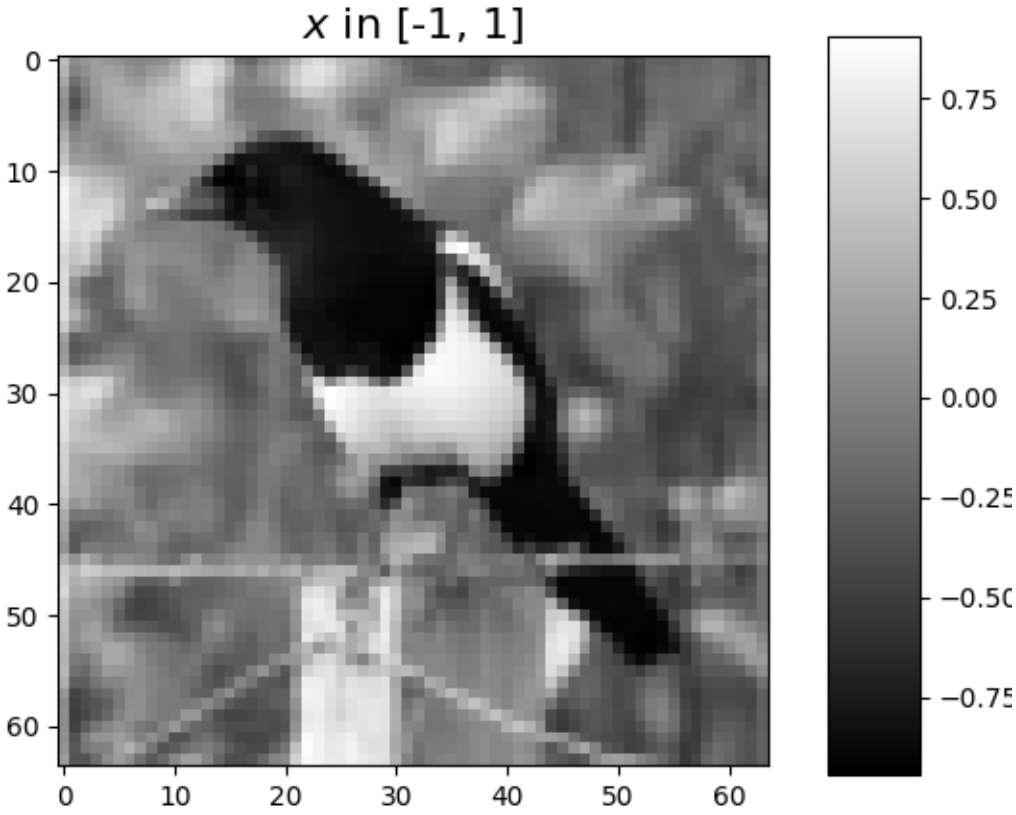
Images x for training neural networks expect values in [-1,1]. The images are normalized using the `transform_gray_norm()` function.

```
import os

import torch
import torchvision
import matplotlib.pyplot as plt

from spyrit.misc.disp import imagesc
from spyrit.misc.statistics import transform_gray_norm

# sphinx_gallery_thumbnail_path = 'fig/tuto1.png'

h = 64  # image size hxh
i = 1  # Image index (modify to change the image)
spyritPath = os.getcwd()
imgs_path = os.path.join(spyritPath, "images/")


# Create a transform for natural images to normalized grayscale image tensors
transform = transform_gray_norm(img_size=h)

# Create dataset and loader (expects class folder 'images/test/')
dataset = torchvision.datasets.ImageFolder(root=imgs_path, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=7)

x, _ = next(iter(dataloader))
print(f"Shape of input images: {x.shape}")
```

(continues on next page)

(continued from previous page)

```
# Select image
x = x[i : i + 1, :, :, :]
x = x.detach().clone()
b, c, h, w = x.shape

# plot
x_plot = x.view(-1, h, h).cpu().numpy()
imagesc(x_plot[0, :, :], r"$x$ in [-1, 1]")
```




```
Shape of input images: torch.Size([7, 1, 64, 64])
```

The measurement and noise operators

Noise operators are defined in the *noise* module. A noise operator computes the following three steps sequentially:

1. Normalization of the image x with values in [-1,1] to get an image $\tilde{x} = \frac{x+1}{2}$ in [0,1], as it is required for measurement simulation

2. Application of the measurement model, i.e., computation of $H\tilde{x}$

3. Application of the noise model

$$y \sim \texttt{Noise}(H\tilde{x}) = \texttt{Noise}\left(\frac{H(x+1)}{2}\right),$$

The normalization is usefull when considering distributions such as the Poisson distribution that are defined on positive values.

Note: The noise operator is constructed from a measurement operator (see the *meas* submodule) in order to compute the measurements $H\tilde{x}$, as given by step #2.

A simple example: identity measurement matrix and no noise

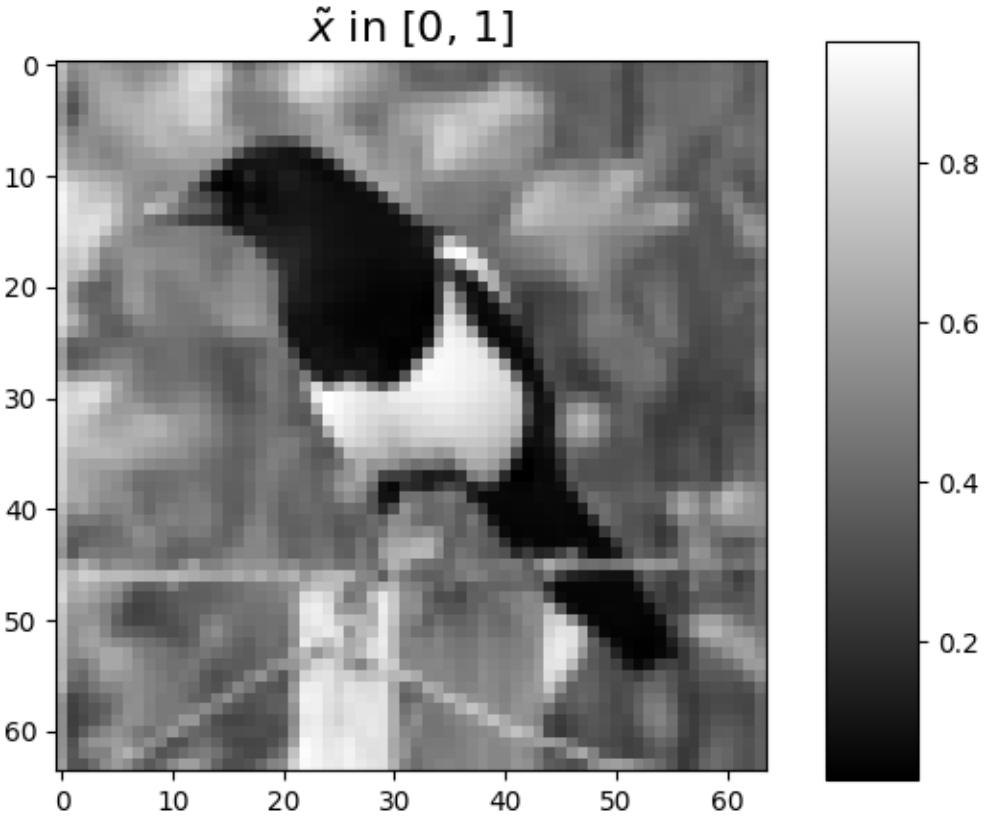
$$y = \tilde{x}$$

We start with a simple example where the measurement matrix H is the identity, which can be handled by the more general *spyrit.core.meas.Linear* class. We consider the noiseless case handled by the *spyrit.core.noise.NoNoise* class.

```
from spyrit.core.meas import Linear
from spyrit.core.noise import NoNoise

meas_op = Linear(torch.eye(h * h))
noise_op = NoNoise(meas_op)
```

We simulate the measurement vector y that we visualise as an image. Remember that the input image x is handled as a vector.

```
x = x.view(b * c, h * w)  # vectorized image
print(f"Shape of vectorized image: {x.shape}")
y_eye = noise_op(x)  # noisy measurement vector
print(f"Shape of simulated measurements y: {y_eye.shape}")

# plot
x_plot = y_eye.view(-1, h, h).cpu().numpy()
imagesc(x_plot[0, :, :], r"$\tilde{x}$ in [0, 1]")
```




```
Shape of vectorized image: torch.Size([1, 4096])
Shape of simulated measurements y: torch.Size([1, 4096])
```

Note: Note that the image identical to the original one, except it has been normalized in [0,1].

Same example with Poisson noise

We now consider Poisson noise, i.e., a noisy measurement vector given by

$$y \sim \mathcal{P}(\alpha H \tilde{x}),$$

where α is a scalar value that represents the maximum image intensity (in photons). The larger α, the higher the signal-to-noise ratio.

We consider the *spyrit.core.noise.Poisson* class and set α to 100 photons.

```
from spyrit.core.noise import Poisson
from spyrit.misc.disp import add_colorbar, noaxis

alpha = 100  # number of photons
noise_op = Poisson(meas_op, alpha)
```

We simulate two noisy measurement vectors

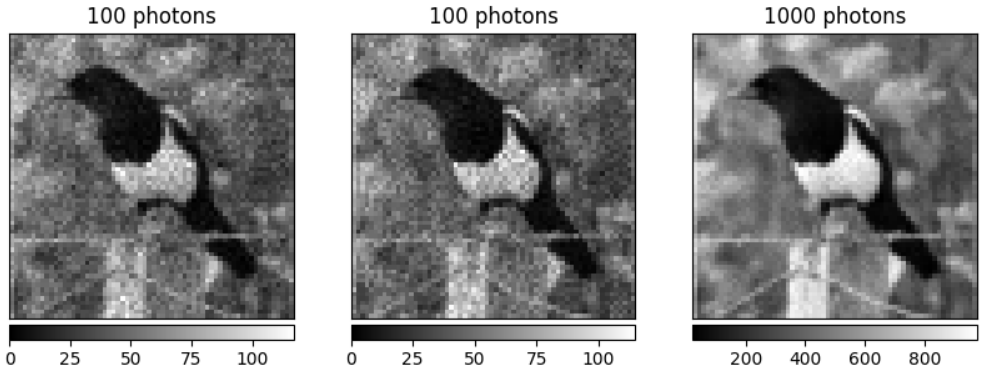
```
y1 = noise_op(x)  # a noisy measurement vector
y2 = noise_op(x)  # another noisy measurement vector
```

We now consider the case $\alpha = 1000$ photons.

```
noise_op.alpha = 1000
y3 = noise_op(x)  # noisy measurement vector
```

We finally plot the measurement vectors as images

```
# plot
y1_plot = y1.view(b, h, h).detach().numpy()
y2_plot = y2.view(b, h, h).detach().numpy()
y3_plot = y3.view(b, h, h).detach().numpy()

f, axs = plt.subplots(1, 3, figsize=(10, 5))
axs[0].set_title("100 photons")
im = axs[0].imshow(y1_plot[0, :, :], cmap="gray")
add_colorbar(im, "bottom")

axs[1].set_title("100 photons")
im = axs[1].imshow(y2_plot[0, :, :], cmap="gray")
add_colorbar(im, "bottom")

axs[2].set_title("1000 photons")
im = axs[2].imshow(y3_plot[0, :, :], cmap="gray")
add_colorbar(im, "bottom")

noaxis(axs)
```



As expected the signal-to-noise ratio of the measurement vector is higher for 1,000 photons than for 100 photons

Note: Not only the signal-to-noise, but also the scale of the measurements depends on α, which motivate the introduction of the preprocessing operator.

The preprocessing operator

Preprocessing operators are defined in the *spyrit.core.prep* module. A preprocessing operator applies to the noisy measurements

$$m = \texttt{Prep}(y),$$

For instance, a preprocessing operator can be used to compensate for the scaling factors that appear in the measurement or noise operators. In this case, a preprocessing operator is closely linked to its measurement and/or noise operator counterpart. While scaling factors are required to simulate realistic measurements, they are not required for reconstruction.

Preprocessing measurements corrupted by Poisson noise

We consider the *spyrit.core.prep.DirectPoisson* class that intends to "undo" the *spyrit.core.noise.Poisson* class by compensating for:

- the scaling that appears when computing Poisson-corrupted measurements
- the affine transformation to get images in [0,1] from images in [-1,1]

For this, it computes

$$m = \frac{2}{\alpha} y - H1$$

We consider the *spyrit.core.prep.DirectPoisson* class and set α to 100 photons.

```
from spyrit.core.prep import DirectPoisson

alpha = 100  # number of photons
prep_op = DirectPoisson(alpha, meas_op)
```

We preprocess the first two noisy measurement vectors

```
m1 = prep_op(y1)
m2 = prep_op(y2)
```

We now consider the case $\alpha = 1000$ photons to preprocess the third measurement vector

```
prep_op.alpha = 1000
m3 = prep_op(y3)
```

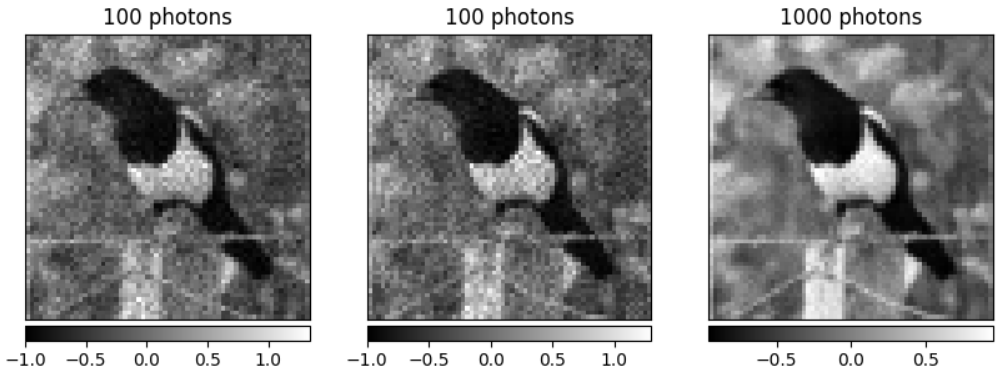
We finally plot the preprocessed measurement vectors as images

```
# plot
m1 = m1.view(b, h, h).detach().numpy()
m2 = m2.view(b, h, h).detach().numpy()
m3 = m3.view(b, h, h).detach().numpy()

```

(continues on next page)

(continued from previous page)

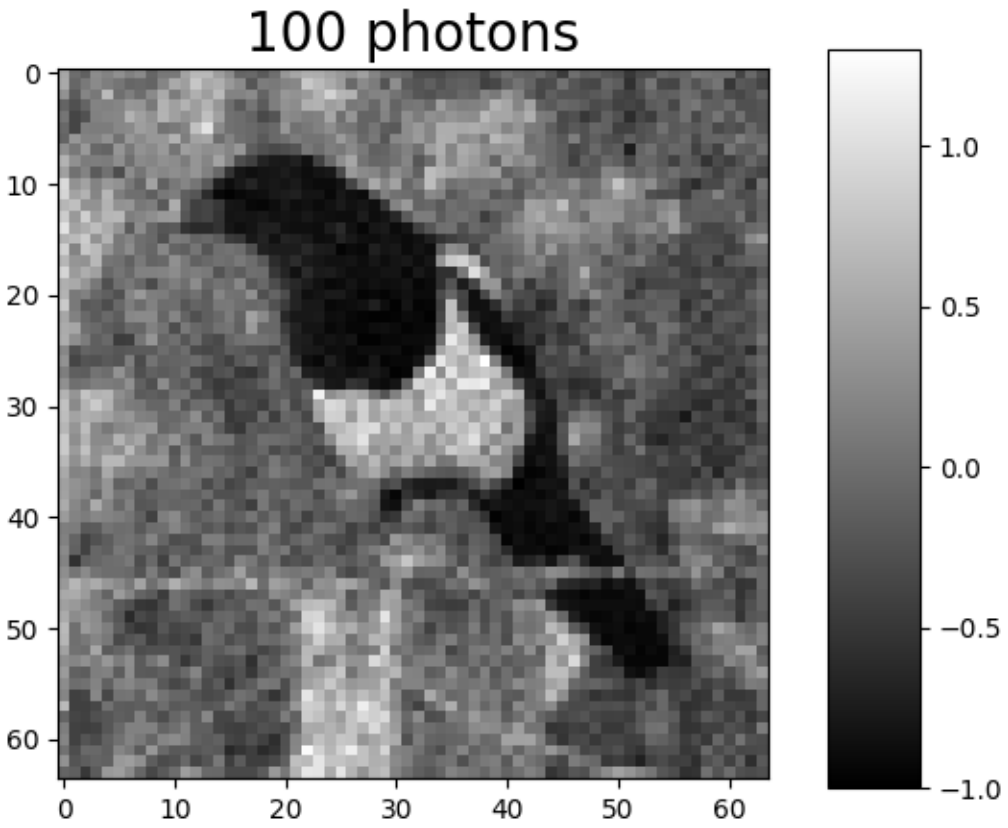
```
f, axs = plt.subplots(1, 3, figsize=(10, 5))
axs[0].set_title("100 photons")
im = axs[0].imshow(m1[0, :, :], cmap="gray")
add_colorbar(im, "bottom")

axs[1].set_title("100 photons")
im = axs[1].imshow(m2[0, :, :], cmap="gray")
add_colorbar(im, "bottom")

axs[2].set_title("1000 photons")
im = axs[2].imshow(m3[0, :, :], cmap="gray")
add_colorbar(im, "bottom")

noaxis(axs)
```



Note: The preprocessed measurements still have different the signal-to-noise ratios depending on α; however, they (approximately) all lie within the same range (here, [-1, 1]).

We show again one of the preprocessed measurement vectors (tutorial thumbnail purpose)

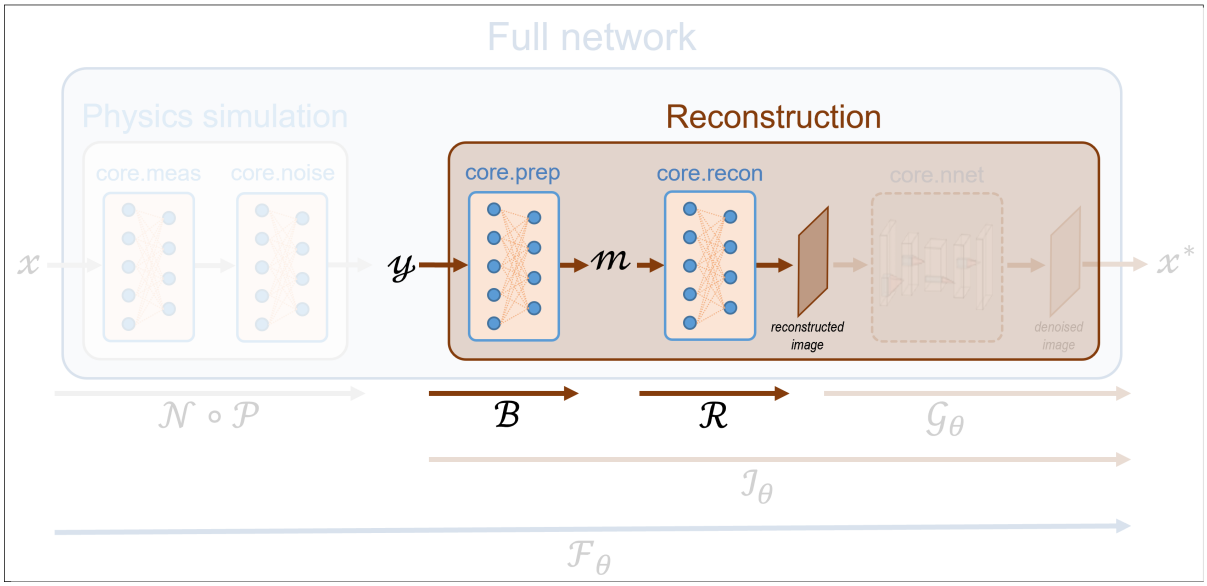
```
# Plot
imagesc(m2[0, :, :], "100 photons", title_fontsize=20)
```



Total running time of the script: (0 minutes 1.217 seconds)

02. Pseudoinverse solution from linear measurements

This tutorial shows how to simulate measurements and perform image reconstruction. The measurement operator is chosen as a Hadamard matrix with positive coefficients. Note that this matrix can be replaced by any desired matrix.



These tutorials load image samples from */images/*.

Load a batch of images

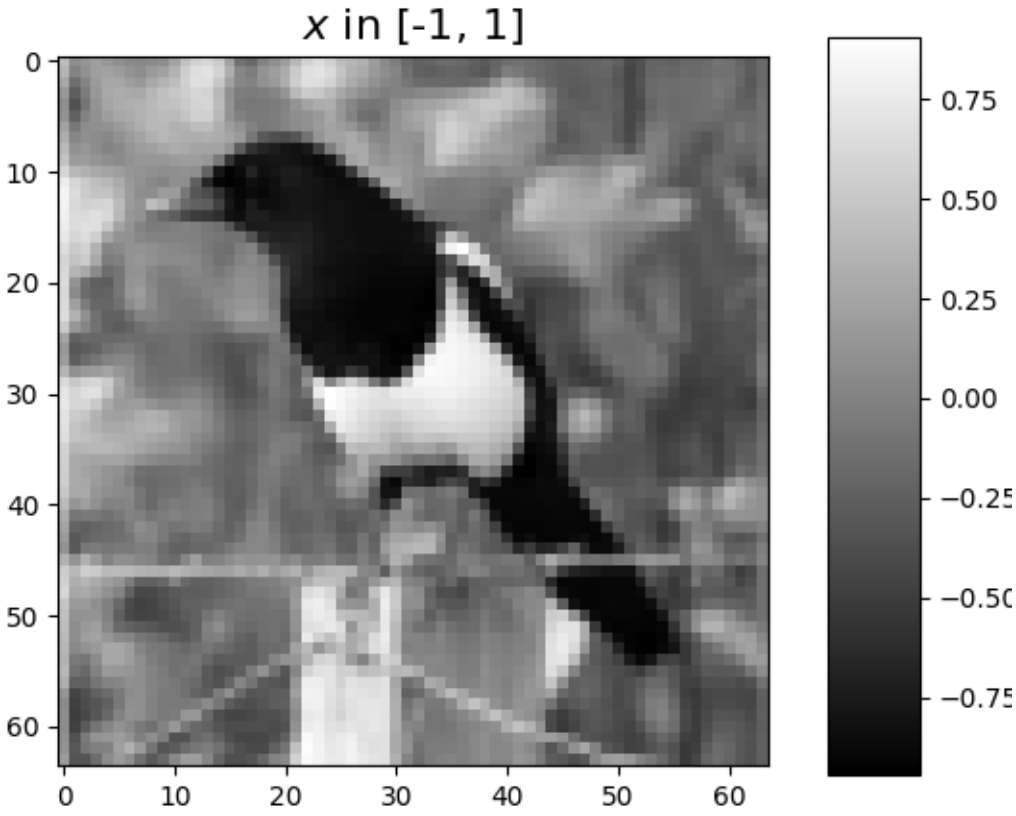
Images x for training expect values in [-1,1]. The images are normalized using the `transform_gray_norm()` function.

```
import os

import torch
import torchvision
import numpy as np

from spyrit.misc.disp import imagesc
from spyrit.misc.statistics import transform_gray_norm

# sphinx_gallery_thumbnail_path = 'fig/tuto2.png'

h = 64  # image size hxh
i = 1  # Image index (modify to change the image)
spyritPath = os.getcwd()
imgs_path = os.path.join(spyritPath, "images/")


# Create a transform for natural images to normalized grayscale image tensors
transform = transform_gray_norm(img_size=h)

# Create dataset and loader (expects class folder 'images/test/')
dataset = torchvision.datasets.ImageFolder(root=imgs_path, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=7)

x, _ = next(iter(dataloader))
print(f"Shape of input images: {x.shape}")

# Select image
```

(continues on next page)

(continued from previous page)

```
x = x[i : i + 1, :, :, :]
x = x.detach().clone()
b, c, h, w = x.shape

# plot
x_plot = x.view(-1, h, h).cpu().numpy()
imagesc(x_plot[0, :, :], r"$x$ in [-1, 1]")
```




```
Shape of input images: torch.Size([7, 1, 64, 64])
```

Define a measurement operator

We consider the case where the measurement matrix is the positive component of a Hadamard matrix, which is often used in single-pixel imaging. First, we compute a full Hadamard matrix that computes the 2D transform of an image of size `h` and takes its positive part.

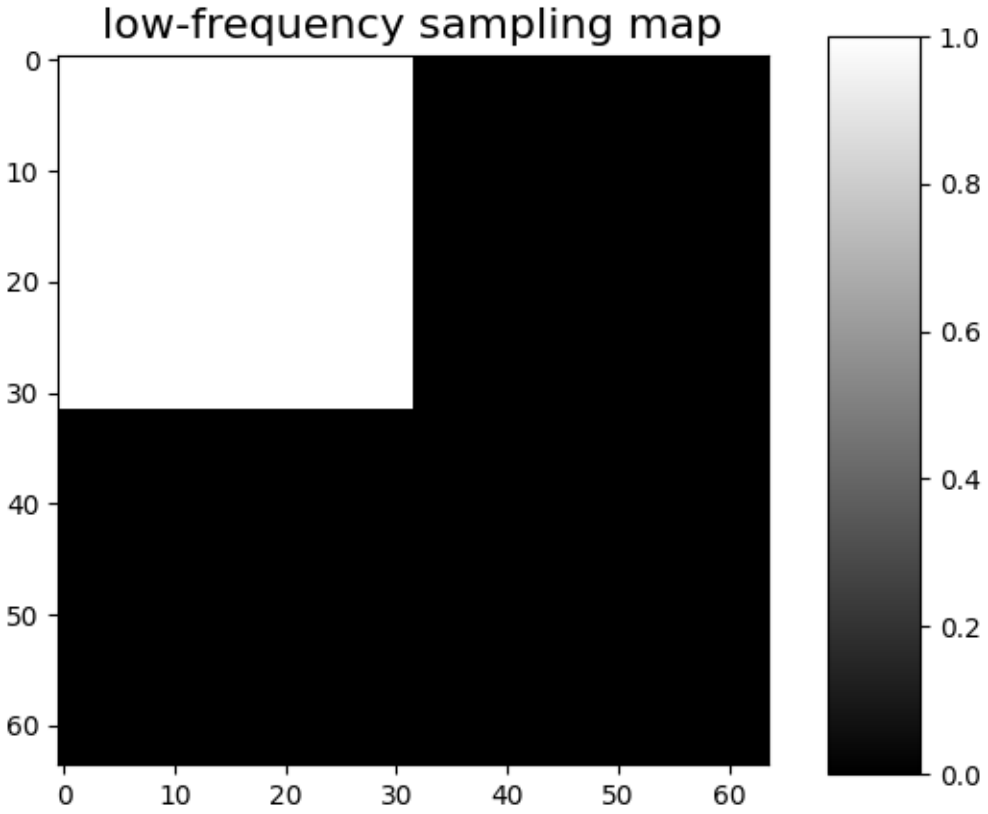
```
from spyrit.misc.walsh_hadamard import walsh2_matrix

F = walsh2_matrix(h)
F = np.where(F > 0, F, 0)
```

Next, we subsample the rows of the measurement matrix to simulate an accelerated acquisition. For this, we use the

spyrit.misc.sampling.sort_by_significance() function that returns an input matrix whose rows are ordered in increasing order of significance according to a given array. The array is a sampling map that indicates the location of the most significant coefficients in the transformed domain.

To keep the low-frequency Hadamard coefficients, we choose a sampling map with ones in the top left corner and zeros elsewhere.

```
import math

und = 4  # undersampling factor
M = h**2 // und  # number of measurements (undersampling factor = 4)

Sampling_map = np.ones((h, h))
M_xy = math.ceil(M**0.5)
Sampling_map[:, M_xy:] = 0
Sampling_map[M_xy:, :] = 0

imagesc(Sampling_map, "low-frequency sampling map")
```



After permutation of the full Hadamard matrix, we keep only its first `M` rows

```
from spyrit.misc.sampling import sort_by_significance

F = sort_by_significance(F, Sampling_map, "rows", False)
H = F[:M, :]
```

(continues on next page)

(continued from previous page)

```
print(f"Shape of the measurement matrix: {H.shape}")
```

```
Shape of the measurement matrix: (1024, 4096)
```

Then, we instantiate a *spyrit.core.meas.Linear* measurement operator

```
from spyrit.core.meas import Linear

meas_op = Linear(torch.from_numpy(H), pinv=True)
```

Noiseless case

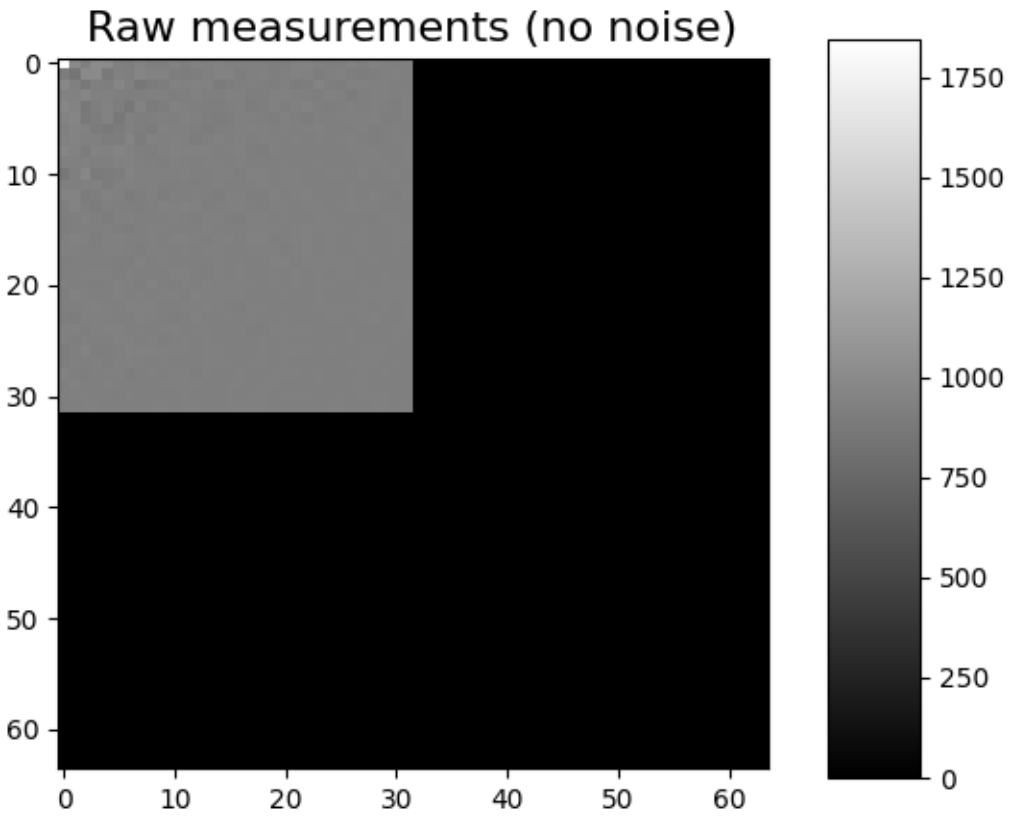
In the noiseless case, we consider the *spyrit.core.noise.NoNoise* noise operator

```
from spyrit.core.noise import NoNoise

noise = NoNoise(meas_op)

# Simulate measurements
y = noise(x.view(b * c, h * w))
print(f"Shape of raw measurements: {y.shape}")
```

```
Shape of raw measurements: torch.Size([1, 1024])
```

To display the subsampled measurement vector as an image in the transformed domain, we use the *spyrit.misc.sampling.meas2img()* function

```
# plot
from spyrit.misc.sampling import meas2img

y_plot = y.detach().numpy().squeeze()
y_plot = meas2img(y_plot, Sampling_map)
print(f"Shape of the raw measurement image: {y_plot.shape}")

imagesc(y_plot, "Raw measurements (no noise)")
```

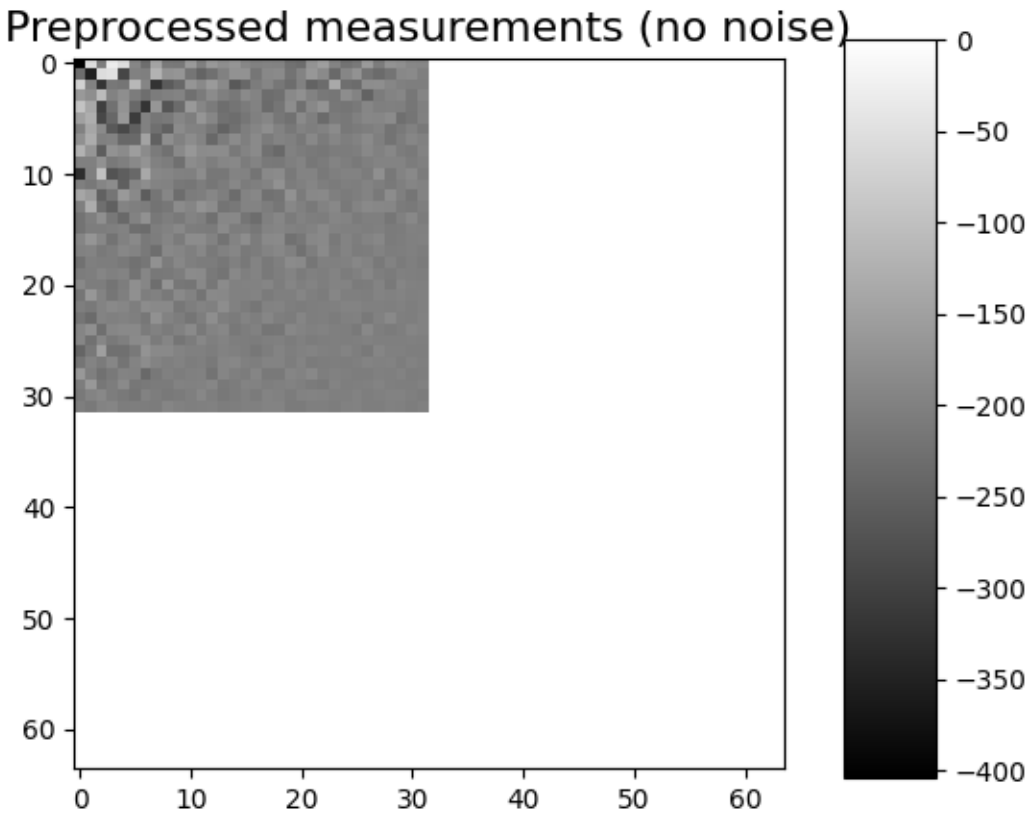



```
Shape of the raw measurement image: (64, 64)
```

We now compute and plot the preprocessed measurements corresponding to an image in [-1,1]. For details in the preprocessing, see *Tutorial 1*.

Note: Using *spyrit.core.prep.DirectPoisson* with $\alpha = 1$ allows to compensate for the image normalisation achieved by *spyrit.core.noise.NoNoise*.

```
from spyrit.core.prep import DirectPoisson

prep = DirectPoisson(1.0, meas_op)  # "Undo" the NoNoise operator

m = prep(y)
print(f"Shape of the preprocessed measurements: {m.shape}")

# plot
m_plot = m.detach().numpy().squeeze()
m_plot = meas2img(m_plot, Sampling_map)
print(f"Shape of the preprocessed measurement image: {m_plot.shape}")

imagesc(m_plot, "Preprocessed measurements (no noise)")
```

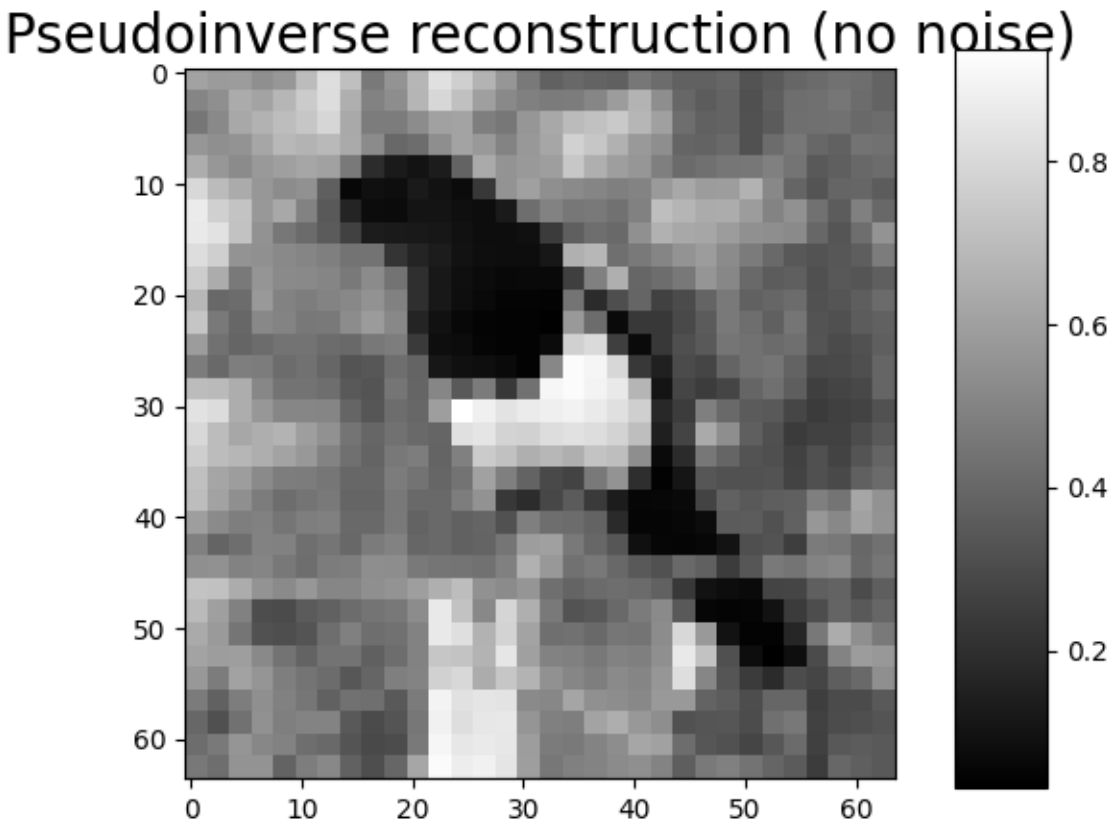



```
Shape of the preprocessed measurements: torch.Size([1, 1024])
Shape of the preprocessed measurement image: (64, 64)
```

Pseudo inverse

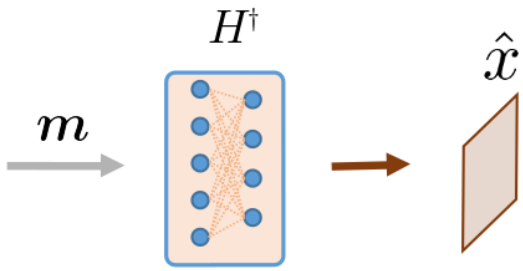
We can use the *spyrit.core.recon.PseudoInverse* class to perform the pseudo inverse reconstruction from the measurements y

```
from spyrit.core.recon import PseudoInverse

# Pseudo-inverse reconstruction operator
recon_op = PseudoInverse()

# Reconstruction
x_rec = recon_op(y, meas_op)

# plot
x_plot = x_rec.squeeze().view(h, h).cpu().numpy()
imagesc(x_plot, "Pseudoinverse reconstruction (no noise)", title_fontsize=20)
```



PinvNet Network

Alternatively, we can consider the *spyrit.core.recon.PinvNet* class that reconstructs an image by computing the pseudoinverse solution, which is fed to a neural networker denoiser. To compute the pseudoinverse solution only, the denoiser can be set to the identity operator




```
from spyrit.core.recon import PinvNet
```

(continues on next page)

(continued from previous page)

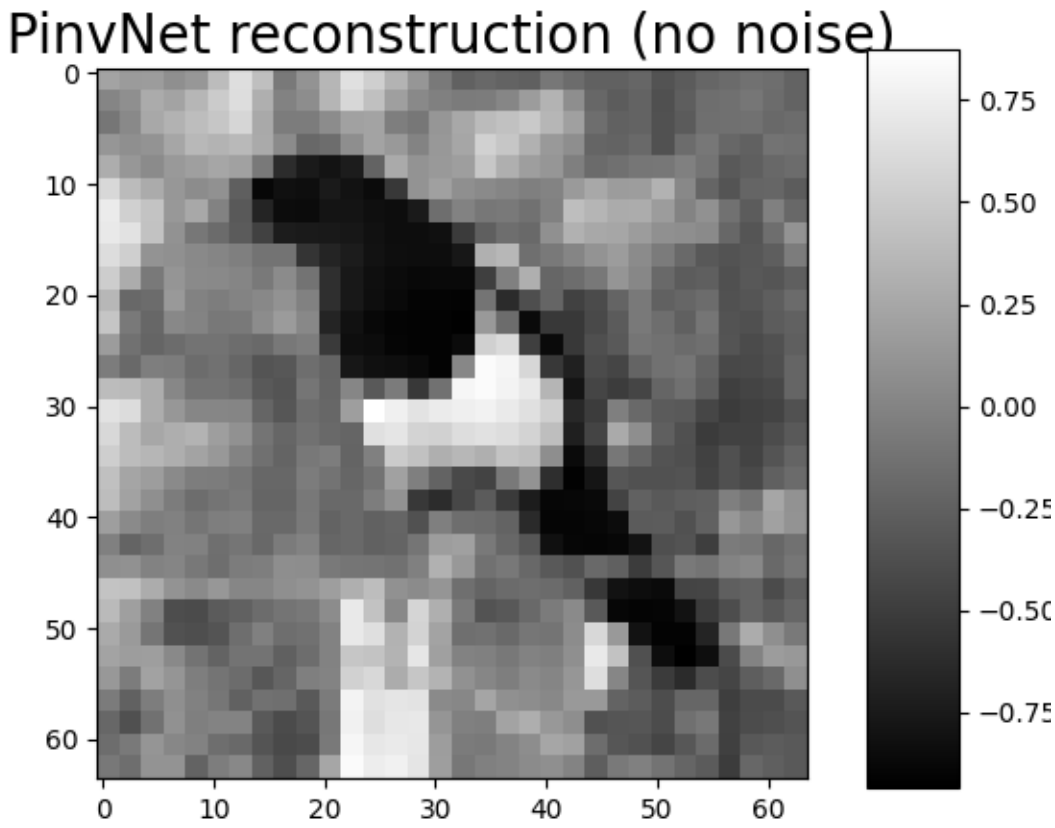
```
pinv_net = PinvNet(noise, prep, denoi=torch.nn.Identity())
```

or equivalently

```
pinv_net = PinvNet(noise, prep)
```

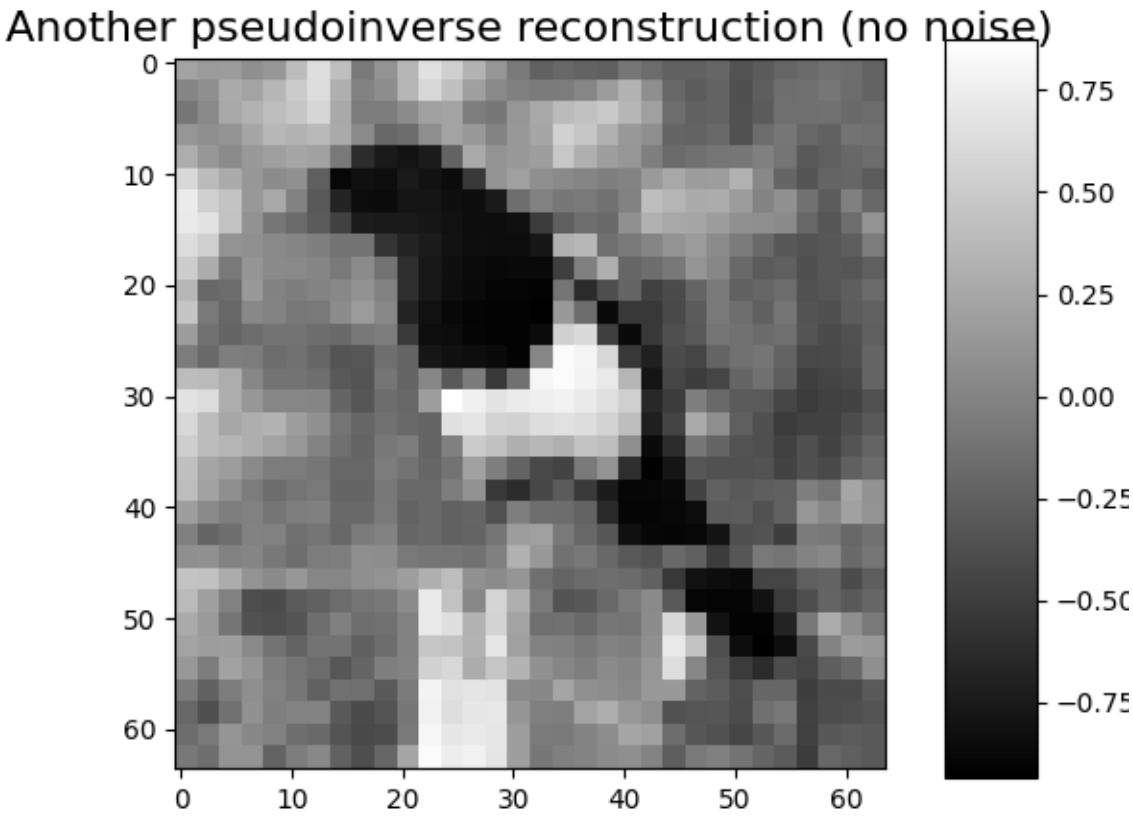
Then, we reconstruct the image from the measurement vector `y` using the *reconstruct()* method

```
x_rec = pinv_net.reconstruct(y)

# plot
x_plot = x_rec.squeeze().cpu().numpy()
imagesc(x_plot, "PinvNet reconstruction (no noise)", title_fontsize=20)
```



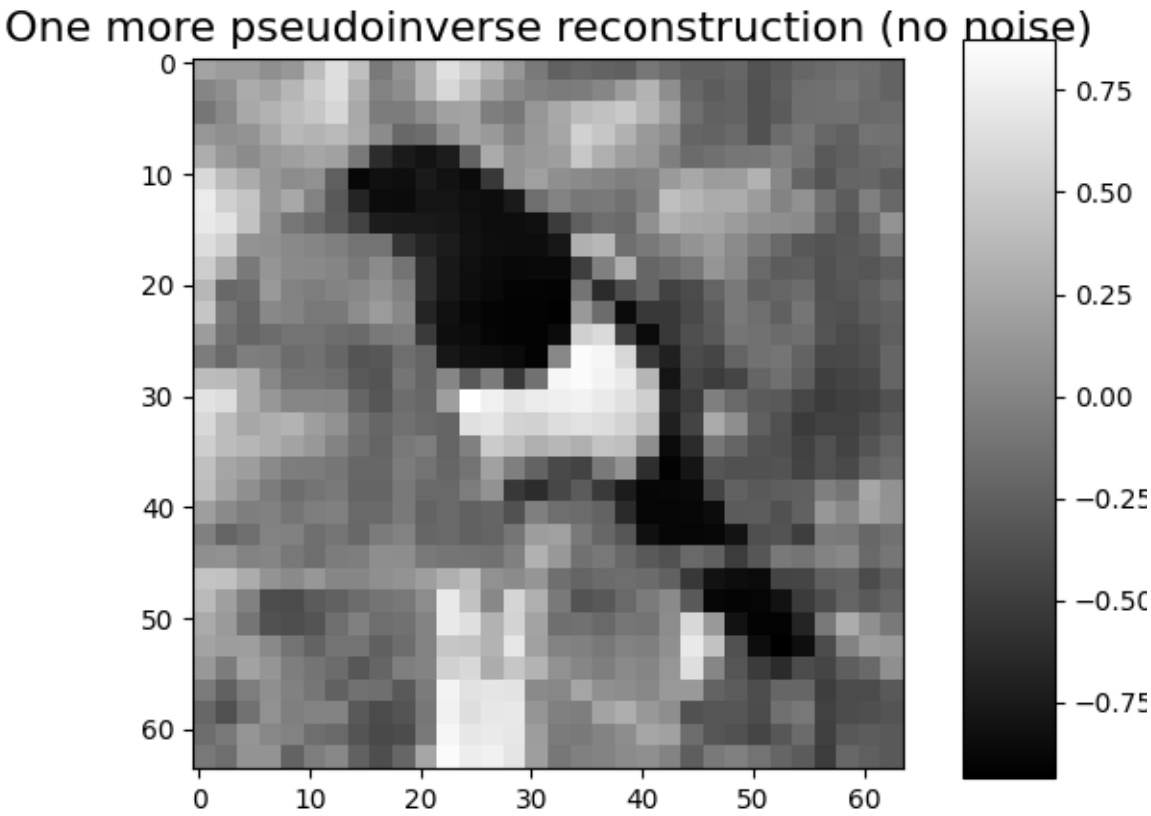
Alternatively, the measurement vector can be simulated using the *acquire()* method

```
y = pinv_net.acquire(x)
x_rec = pinv_net.reconstruct(y)

# plot
x_plot = x_rec.squeeze().cpu().numpy()
imagesc(x_plot, "Another pseudoinverse reconstruction (no noise)")
```



Note that the full module `pinv_net` both simulates noisy measurements and reconstruct them

```
x_rec = pinv_net(x)
print(f"Ground-truth image x: {x.shape}")
print(f"Reconstructed x_rec: {x_rec.shape}")

# plot
x_plot = x_rec.squeeze().cpu().numpy()
imagesc(x_plot, "One more pseudoinverse reconstruction (no noise)")
```




```
Ground-truth image x: torch.Size([1, 1, 64, 64])
Reconstructed x_rec: torch.Size([1, 1, 64, 64])
```

Poisson-corrupted measurement

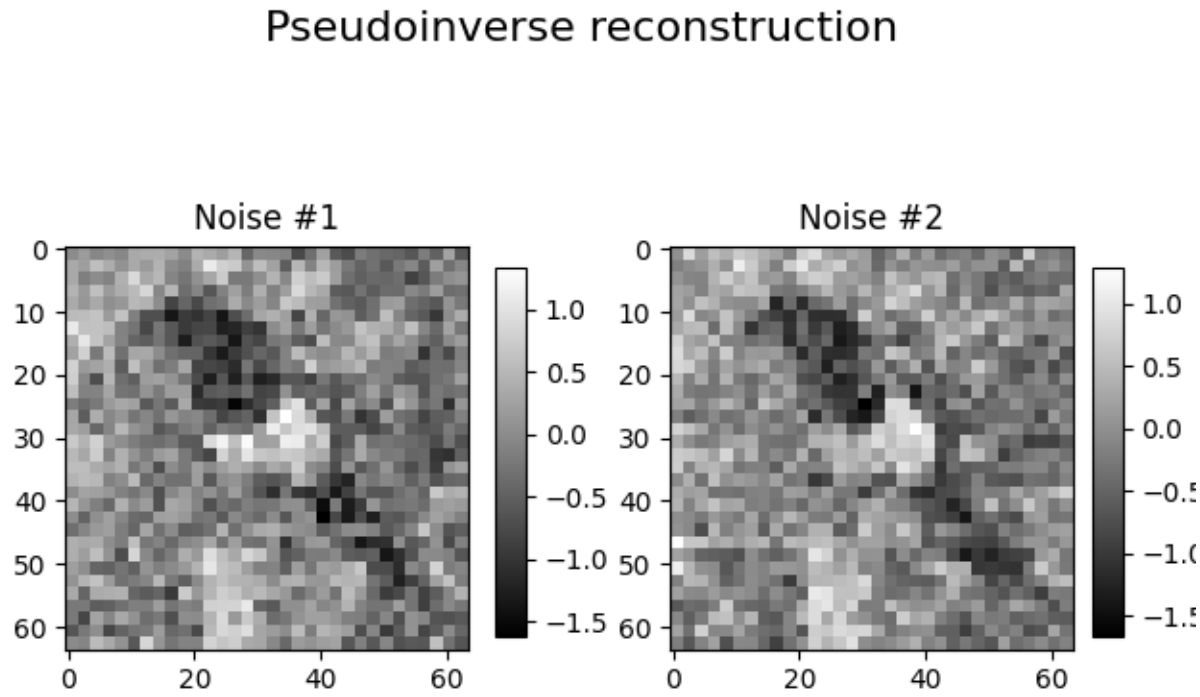
Here, we consider the *spyrit.core.noise.Poisson* class together with a *spyrit.core.prep.DirectPoisson* preprocessing operator (see *Tutorial 1*).

```
alpha = 10  # maximum number of photons in the image

from spyrit.core.noise import Poisson
from spyrit.misc.disp import imagecomp

noise = Poisson(meas_op, alpha)
prep = DirectPoisson(alpha, meas_op)  # To undo the "Poisson" operator
pinv_net = PinvNet(noise, prep)

x_rec_1 = pinv_net(x)
x_rec_2 = pinv_net(x)
print(f"Ground-truth image x: {x.shape}")
print(f"Reconstructed x_rec: {x_rec.shape}")
```

(continues on next page)

(continued from previous page)

```
# plot
x_plot_1 = x_rec_1.squeeze().cpu().numpy()
x_plot_1[:2, :2] = 0.0  # hide the top left "crazy pixel" that collects noise
x_plot_2 = x_rec_2.squeeze().cpu().numpy()
x_plot_2[:2, :2] = 0.0  # hide the top left "crazy pixel" that collects noise
imagecomp(x_plot_1, x_plot_2, "Pseudoinverse reconstruction", "Noise #1", "Noise #2")
```




```
Ground-truth image x: torch.Size([1, 1, 64, 64])
Reconstructed x_rec: torch.Size([1, 1, 64, 64])
```

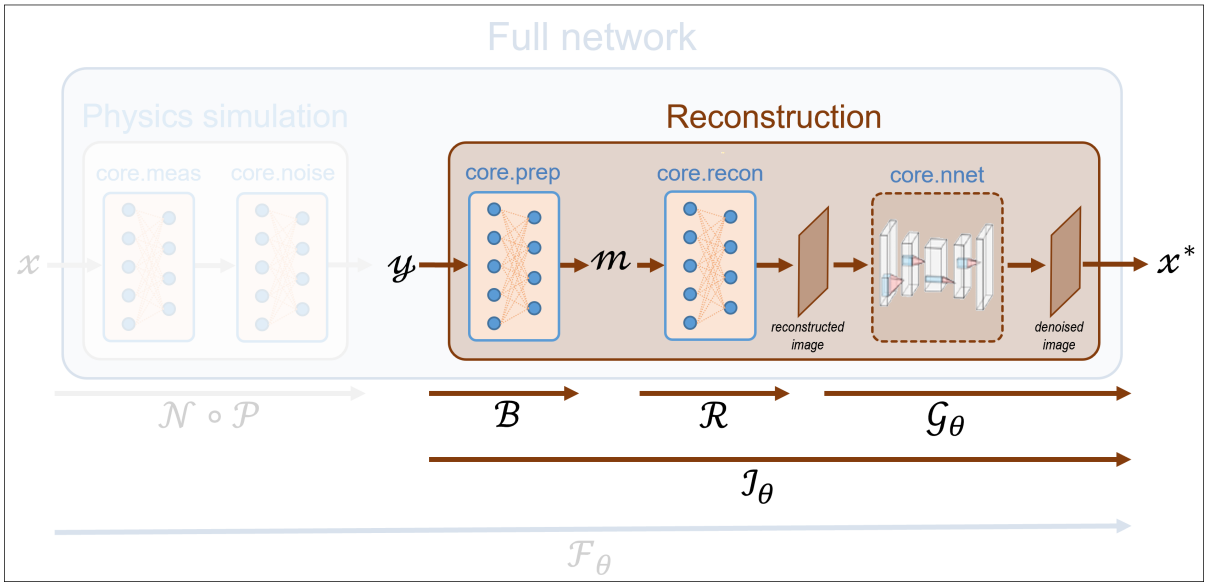
As shown in the next tutorial, a denoising neural network can be trained to postprocess the pseudo inverse solution.

Total running time of the script: (0 minutes 3.627 seconds)

03. Pseudoinverse solution + CNN denoising

This tutorial shows how to simulate measurements and perform image reconstruction using PinvNet (pseudoinverse linear network) with CNN denoising as a last layer. This tutorial is a continuation of the *Pseudoinverse solution tutorial* but uses a CNN denoiser instead of the identity operator in order to remove artefacts.

The measurement operator is chosen as a Hadamard matrix with positive coefficients, which can be replaced by any matrix.



These tutorials load image samples from */images/*.

Load a batch of images

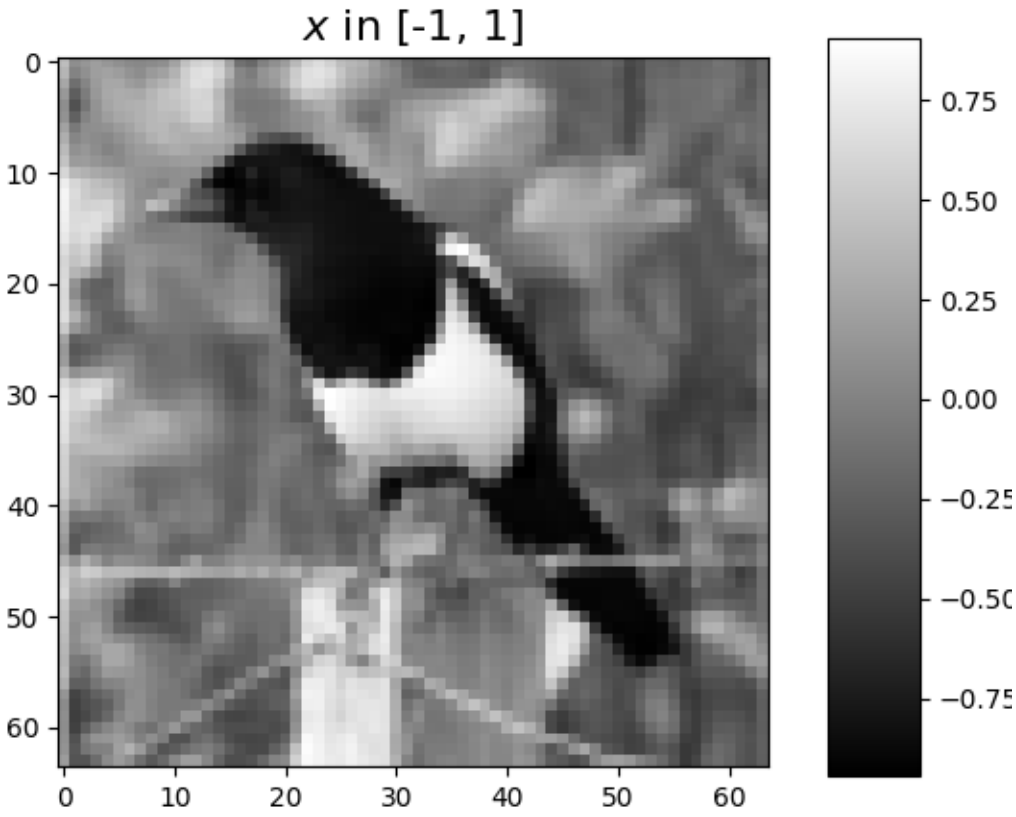
Images x for training expect values in [-1,1]. The images are normalized using the `transform_gray_norm()` function.

```
import os

import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt

from spyrit.misc.disp import imagesc
from spyrit.misc.statistics import transform_gray_norm

# sphinx_gallery_thumbnail_path = 'fig/tuto3.png'

h = 64  # image size hxh
i = 1  # Image index (modify to change the image)
spyritPath = os.getcwd()
imgs_path = os.path.join(spyritPath, "images/")

# Create a transform for natural images to normalized grayscale image tensors
transform = transform_gray_norm(img_size=h)
```

(continues on next page)

(continued from previous page)

```
# Create dataset and loader (expects class folder 'images/test/')
dataset = torchvision.datasets.ImageFolder(root=imgs_path, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=7)

x, _ = next(iter(dataloader))
print(f"Shape of input images: {x.shape}")

# Select image
x = x[i : i + 1, :, :, :]
x = x.detach().clone()
b, c, h, w = x.shape

# plot
x_plot = x.view(-1, h, h).cpu().numpy()
imagesc(x_plot[0, :, :], r"$x$ in [-1, 1]")
```

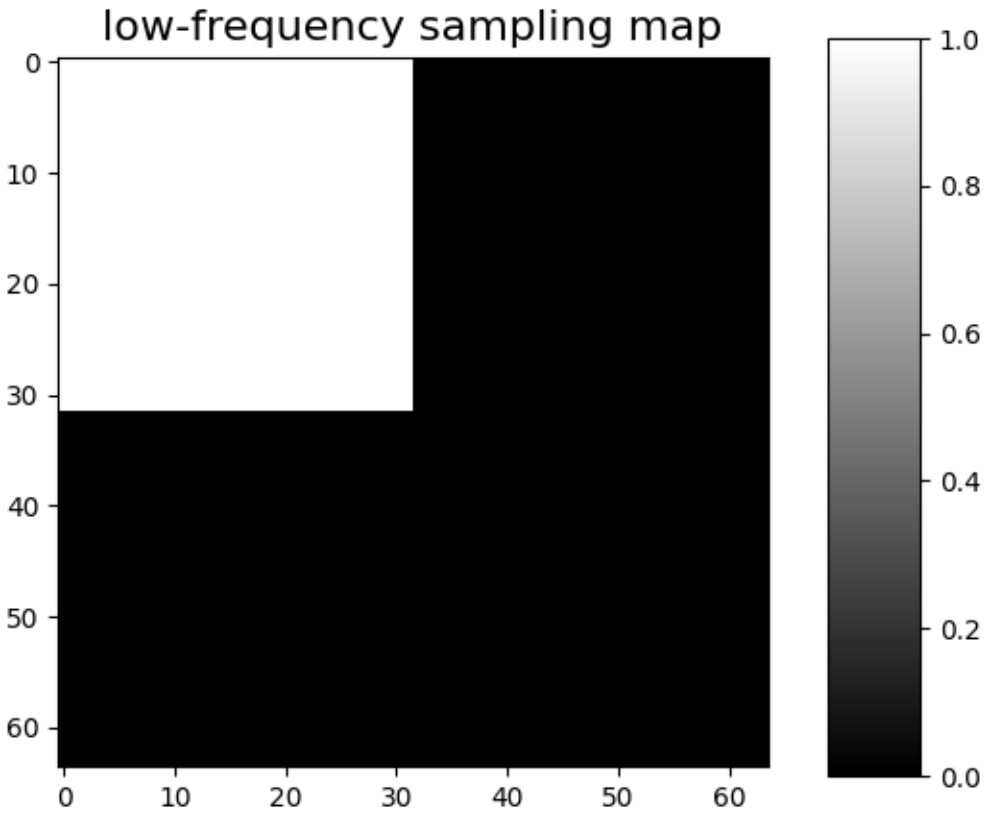



```
Shape of input images: torch.Size([7, 1, 64, 64])
```

Define a measurement operator

We consider the case where the measurement matrix is the positive component of a Hadamard matrix and the sampling operator preserves only the first `M` low-frequency coefficients (see *Positive Hadamard matrix* for full explantion).

```
import math
from spyrit.misc.sampling import sort_by_significance
from spyrit.misc.walsh_hadamard import walsh2_matrix

F = walsh2_matrix(h)
F = np.where(F > 0, F, 0)
und = 4  # undersampling factor
M = h**2 // und  # number of measurements (undersampling factor = 4)

Sampling_map = np.ones((h, h))
M_xy = math.ceil(M**0.5)
Sampling_map[:, M_xy:] = 0
Sampling_map[M_xy:, :] = 0

F = sort_by_significance(F, Sampling_map, "rows", False)
H = F[:M, :]
print(f"Shape of the measurement matrix: {H.shape}")

imagesc(Sampling_map, "low-frequency sampling map")
```



```
Shape of the measurement matrix: (1024, 4096)
```

Then, we instantiate a *spyrit.core.meas.Linear* measurement operator

```
from spyrit.core.meas import Linear

meas_op = Linear(torch.from_numpy(H), pinv=True)
```

Noiseless case

In the noiseless case, we consider the *spyrit.core.noise.NoNoise* noise operator

```
from spyrit.core.noise import NoNoise

N0 = 1.0  # Noise level (noiseless)
noise = NoNoise(meas_op)

# Simulate measurements
y = noise(x.view(b * c, h * w))
print(f"Shape of raw measurements: {y.shape}")
```

```
Shape of raw measurements: torch.Size([1, 1024])
```

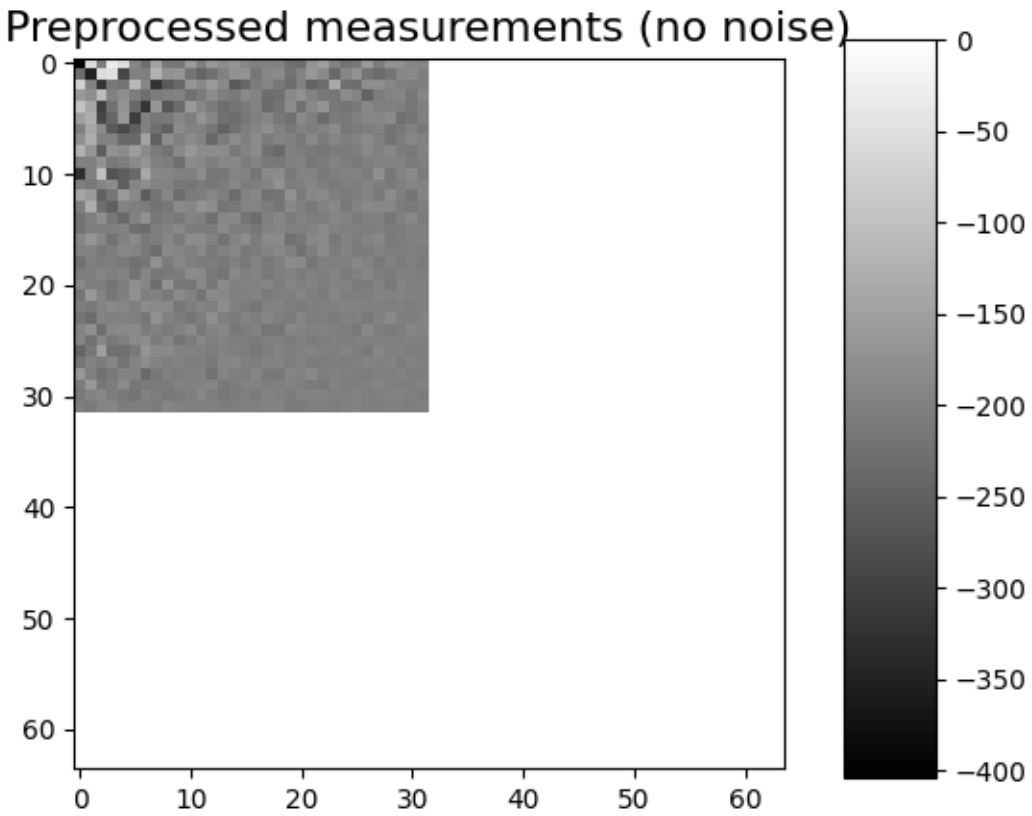
We now compute and plot the preprocessed measurements corresponding to an image in [-1,1]

```
from spyrit.core.prep import DirectPoisson

prep = DirectPoisson(N0, meas_op)  # "Undo" the NoNoise operator

m = prep(y)
print(f"Shape of the preprocessed measurements: {m.shape}")
```

```
Shape of the preprocessed measurements: torch.Size([1, 1024])
```

To display the subsampled measurement vector as an image in the transformed domain, we use the *spyrit.misc.sampling.meas2img()* function

```
# plot
from spyrit.misc.sampling import meas2img

m_plot = m.detach().numpy().squeeze()
m_plot = meas2img(m_plot, Sampling_map)
print(f"Shape of the preprocessed measurement image: {m_plot.shape}")

imagesc(m_plot, "Preprocessed measurements (no noise)")
```




```
Shape of the preprocessed measurement image: (64, 64)
```

PinvNet Network

We consider the *spyrit.core.recon.PinvNet* class that reconstructs an image by computing the pseudoinverse solution, which is fed to a neural network denoiser. To compute the pseudoinverse solution only, the denoiser can be set to the identity operator

```
from spyrit.core.recon import PinvNet

pinv_net = PinvNet(noise, prep, denoi=torch.nn.Identity())
```

or equivalently

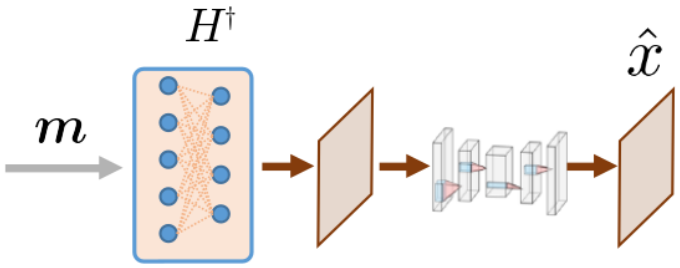
```
pinv_net = PinvNet(noise, prep)
```

Then, we reconstruct the image from the measurement vector `y` using the *reconstruct()* method

```
x_rec = pinv_net.reconstruct(y)
```

Removing artefacts with a CNN

Artefacts can be removed by selecting a neural network denoiser (last layer of PinvNet). We select a simple CNN using the *spyrit.core.nnet.ConvNet* class, but this can be replaced by any neural network (eg. UNet from *spyrit.core.nnet.Unet*).




```
from spyrit.core.nnet import ConvNet, Unet
from spyrit.core.train import load_net

# Define PInvNet with ConvNet denoising layer
denoi = ConvNet()
pinv_net_cnn = PinvNet(noise, prep, denoi)

# Send to GPU if available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
pinv_net_cnn = pinv_net_cnn.to(device)
```

As an example, we use a simple ConvNet that has been pretrained using STL-10 dataset. We download the pretrained weights and load them into the network.

```
# Load pretrained model
model_path = "./model"
num_epochs = 1

pretrained_model_num = 3
if pretrained_model_num == 1:
    # 1 epoch
    url_cnn = "https://drive.google.com/file/d/1iGjxOk06nlB5hSm3caIfx0vy2byQd-ZC/view?
↪usp=drive_link"
    name_cnn = "pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_1_lr_0.001_sss_10_sdr_0.5_bs_512_
↪reg_1e-07.pth"
    num_epochs = 1
elif pretrained_model_num == 2:
    # 5 epochs
    url_cnn = "https://drive.google.com/file/d/1tzZg1lU3AxOi8-EVXFgnxdtqQCJPjQ9f/view?
↪usp=drive_link"
    name_cnn = (
        "pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_5_lr_0.001_sss_10_sdr_0.5_bs_512.pth"
    )
```

(continues on next page)

(continued from previous page)

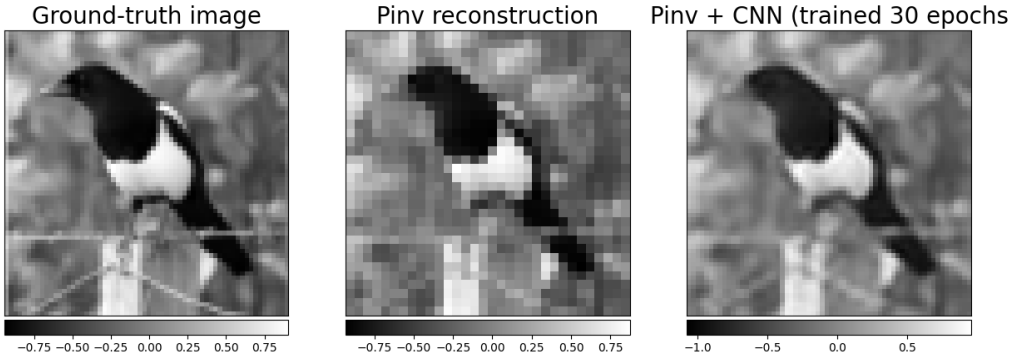
```
    num_epochs = 5
elif pretrained_model_num == 3:
    # 30 epochs
    url_cnn = "https://drive.google.com/file/d/1IZYff1xQxJ3ckAnObqAWyOure6Bjkj4k/view?
↪usp=drive_link"
    name_cnn = "pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_30_lr_0.001_sss_10_sdr_0.5_bs_
↪512_reg_1e-07.pth"
    num_epochs = 30

# Create model folder
if os.path.exists(model_path) is False:
    os.mkdir(model_path)
    print(f"Created {model_path}")

# Download model weights
model_cnn_path = os.path.join(model_path, name_cnn)
print(model_cnn_path)
if os.path.exists(model_cnn_path) is False:
    try:
        import gdown

        gdown.download(url_cnn, f"{model_cnn_path}.pth", quiet=False, fuzzy=True)
    except:
        print(f"Model {model_cnn_path} not downloaded!")

# Load model weights
load_net(model_cnn_path, pinv_net_cnn, device, False)
print(f"Model {model_cnn_path} loaded.")
```

```
Created ./model
./model/pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_30_lr_0.001_sss_10_sdr_0.5_bs_512_reg_1e-
↪07.pth
Downloading...
From: https://drive.google.com/uc?id=1IZYff1xQxJ3ckAnObqAWyOure6Bjkj4k
To: /home/docs/checkouts/readthedocs.org/user_builds/spyrit/checkouts/master/tutorial/
↪model/pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_30_lr_0.001_sss_10_sdr_0.5_bs_512_reg_1e-
↪07.pth.pth

  0%|          | 0.00/50.4M [00:00<?, ?B/s]
  2%|         | 1.05M/50.4M [00:00<00:04, 10.4MB/s]
 18%|        | 8.91M/50.4M [00:00<00:00, 48.3MB/s]
 34%|      | 17.3M/50.4M [00:00<00:00, 50.4MB/s]
 53%|    | 26.7M/50.4M [00:00<00:00, 65.0MB/s]
 68%|   | 34.1M/50.4M [00:00<00:00, 66.2MB/s]
 86%| | 43.5M/50.4M [00:00<00:00, 74.8MB/s]
100%|| 50.4M/50.4M [00:00<00:00, 67.2MB/s]
Model Loaded: ./model/pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_30_lr_0.001_sss_10_sdr_0.5_
↪bs_512_reg_1e-07.pth
Model ./model/pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_30_lr_0.001_sss_10_sdr_0.5_bs_512_
↪reg_1e-07.pth loaded.
```

We now reconstruct the image using PinvNet with pretrained CNN denoising and plot results side by side with the

PinvNet without denoising

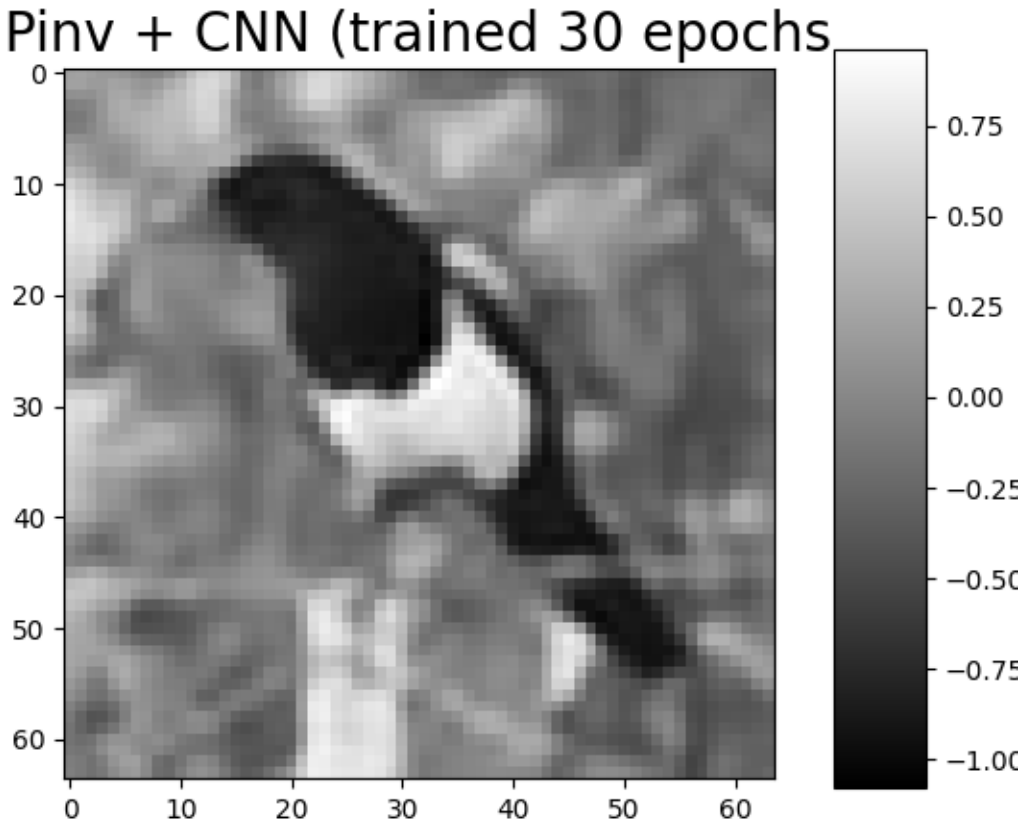
```
with torch.no_grad():
    x_rec_cnn = pinv_net_cnn.reconstruct(y.to(device))
    x_rec_cnn = pinv_net_cnn(x.to(device))

# plot
x_plot = x.squeeze().cpu().numpy()
x_plot2 = x_rec.squeeze().cpu().numpy()
x_plot3 = x_rec_cnn.squeeze().cpu().numpy()

from spyrit.misc.disp import add_colorbar, noaxis

f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))
im1 = ax1.imshow(x_plot, cmap="gray")
ax1.set_title("Ground-truth image", fontsize=20)
noaxis(ax1)
add_colorbar(im1, "bottom", size="20%")

im2 = ax2.imshow(x_plot2, cmap="gray")
ax2.set_title("Pinv reconstruction", fontsize=20)
noaxis(ax2)
add_colorbar(im2, "bottom", size="20%")

im3 = ax3.imshow(x_plot3, cmap="gray")
ax3.set_title(f"Pinv + CNN (trained {num_epochs} epochs", fontsize=20)
noaxis(ax3)
add_colorbar(im3, "bottom", size="20%")
```




```
<matplotlib.colorbar.Colorbar object at 0x7f32cca68710>
```

We show the best result again (tutorial thumbnail purpose)

```
# Plot
imagesc(x_plot3, f"Pinv + CNN (trained {num_epochs} epochs", title_fontsize=20)

plt.show()
```



In the next tutorial, we will show how to train PinvNet + CNN denoiser.

Total running time of the script: (0 minutes 4.474 seconds)

04. Train pseudoinverse solution + CNN denoising

This tutorial shows how to train PinvNet with a CNN denoiser for reconstruction of linear measurements (results shown in the *previous tutorial*). As an example, we use a small CNN, which can be replaced by any other network, for example Unet. Training is performed on the STL-10 dataset.

You can use Tensorboard for Pytorch for experiment tracking and for visualizing the training process: losses, network weights, and intermediate results (reconstructed images at different epochs).

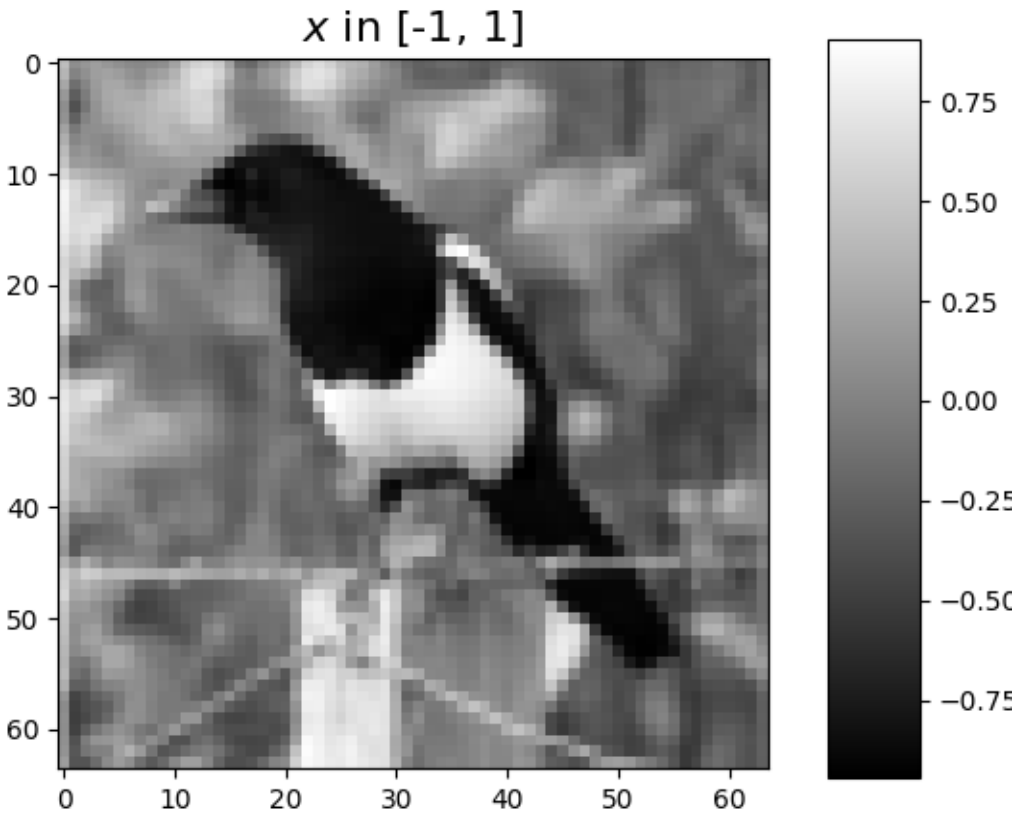
The linear measurement operator is chosen as the positive part of a Hadamard matrix, but this matrix can be replaced by any desired matrix.

These tutorials load image samples from */images/*.

Load a batch of images

First, we load an image x and normalized it to [-1,1], as in previous examples.

```
import os

import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt

from spyrit.misc.disp import imagesc
from spyrit.misc.statistics import transform_gray_norm

h = 64  # image size hxh
i = 1  # Image index (modify to change the image)
spyritPath = os.getcwd()
imgs_path = os.path.join(spyritPath, "images/")


# Create a transform for natural images to normalized grayscale image tensors
transform = transform_gray_norm(img_size=h)

# Create dataset and loader (expects class folder 'images/test/')
dataset = torchvision.datasets.ImageFolder(root=imgs_path, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=7)

x, _ = next(iter(dataloader))
print(f"Shape of input images: {x.shape}")

# Select image
x = x[i : i + 1, :, :, :]
x = x.detach().clone()
b, c, h, w = x.shape

# plot
x_plot = x.view(-1, h, h).cpu().numpy()
imagesc(x_plot[0, :, :], r"$x$ in [-1, 1]")
```




```
Shape of input images: torch.Size([7, 1, 64, 64])
```

Define a dataloader

We define a dataloader for STL-10 dataset using *spyrit.misc.statistics.data_loaders_stl10()*. This will download the dataset to the provided path if it is not already downloaded. It is based on pytorch pre-loaded dataset `torchvision.datasets.STL10` and `torch.utils.data.DataLoader`, which creates a generator that iterates through the dataset, returning a batch of images and labels at each iteration.

Set `mode_run` to True in the script below to download the dataset and for training; otherwise, pretrained weights and results will be download for display.

```
from spyrit.misc.statistics import data_loaders_stl10
from pathlib import Path

# Parameters
h = 64  # image size hxh
data_root = Path("./data")  # path to data folder (where the dataset is stored)
batch_size = 512

# Dataloader for STL-10 dataset
mode_run = False
if mode_run:
```

(continues on next page)

(continued from previous page)

```
    dataloaders = data_loaders_stl10(
        data_root,
        img_size=h,
        batch_size=batch_size,
        seed=7,
        shuffle=True,
        download=True,
    )
```

Define a measurement operator

We consider the case where the measurement matrix is the positive component of a Hadamard matrix, which is often used in single-pixel imaging (see *Hadamard matrix*). Then, we simulate an accelerated acquisition by keeping only the first `M` low-frequency coefficients (see *low frequency sampling*).

```
import math
from spyrit.misc.walsh_hadamard import walsh2_matrix
from spyrit.misc.sampling import sort_by_significance

und = 4  # undersampling factor
M = h**2 // und  # number of measurements (undersampling factor = 4)

F = walsh2_matrix(h)
F = np.where(F > 0, F, 0)

Sampling_map = np.ones((h, h))
M_xy = math.ceil(M**0.5)
Sampling_map[:, M_xy:] = 0
Sampling_map[M_xy:, :] = 0

# imagesc(Sampling_map, 'low-frequency sampling map')

F = sort_by_significance(F, Sampling_map, "rows", False)
H = F[:M, :]

print(f"Shape of the measurement matrix: {H.shape}")
```

```
Shape of the measurement matrix: (1024, 4096)
```

Then, we instantiate a *spyrit.core.meas.Linear* measurement operator, a *spyrit.core.noise.NoNoise* noise operator for noiseless case, and a preprocessing measurements operator *spyrit.core.prep.DirectPoisson*.

```
from spyrit.core.meas import Linear
from spyrit.core.noise import NoNoise
from spyrit.core.prep import DirectPoisson

meas_op = Linear(torch.from_numpy(H), pinv=True)
noise = NoNoise(meas_op)
N0 = 1.0  # Mean maximum total number of photons
prep = DirectPoisson(N0, meas_op)  # "Undo" the NoNoise operator
```

PinvNet Network

We consider the *spyrit.core.recon.PinvNet* class that reconstructs an image by computing the pseudoinverse solution and applies a nonlinear network denoiser. First, we must define the denoiser. As an example, we choose a small CNN using the *spyrit.core.nnet.ConvNet* class. Then, we define the PinvNet network by passing the noise and preprocessing operators and the denoiser.

```
from spyrit.core.nnet import ConvNet
from spyrit.core.recon import PinvNet

denoiser = ConvNet()
model = PinvNet(noise, prep, denoi=denoiser)

# Send to GPU if available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Use multiple GPUs if available
if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
    model = nn.DataParallel(model)

model = model.to(device)
```

Note: In the example provided, we choose a small CNN using the *spyrit.core.nnet.ConvNet* class. This can be replaced by any denoiser, for example the *spyrit.core.nnet.Unet* class or a custom denoiser.

Define a Loss function optimizer and scheduler

In order to train the network, we need to define a loss function, an optimizer and a scheduler. We use the Mean Square Error (MSE) loss function, weigh decay loss and the Adam optimizer. The scheduler decreases the learning rate by a factor of `gamma` every `step_size` epochs.

```
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
from spyrit.core.train import save_net, Weight_Decay_Loss

# Parameters
lr = 1e-3
step_size = 10
gamma = 0.5

loss = nn.MSELoss()
criterion = Weight_Decay_Loss(loss)
optimizer = optim.Adam(model.parameters(), lr=lr)
scheduler = lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=gamma)
```

Train the network

To train the network, we use the *train_model()* function, which handles the training process. It iterates through the dataloader, feeds the inputs to the network and optimizes the solution (by computing the loss and its gradients and updating the network weights at each iteration). In addition, it computes the loss and desired metrics on the training and validation sets at each iteration. The training process can be monitored using Tensorboard.

Note: To launch Tensorboard type in a new console:

tensorboard –logdir runs

and open the provided link in a browser. The training process can be monitored in real time in the "Scalars" tab. The "Images" tab allows to visualize the reconstructed images at different iterations `tb_freq`.

In order to train, you must set `mode_run` to True for training. It is set to False by default to download the pretrained weights and results for display, as training takes around 40 min for 30 epochs.

```
# We train for one epoch only to check that everything works fine.

from spyrit.core.train import train_model
from datetime import datetime

# Parameters
model_root = Path("./model")  # path to model saving files
num_epochs = 5  # number of training epochs (num_epochs = 30)
checkpoint_interval = 2  # interval between saving model checkpoints
tb_freq = (
    50  # interval between logging to Tensorboard (iterations through the dataloader)
)

# Path for Tensorboard experiment tracking logs
name_run = "stdl10_hadampos"
now = datetime.now().strftime("%Y-%m-%d_%H-%M")
tb_path = f"runs/runs_{name_run}_n{int(N0)}_m{M}/{now}"

# Train the network
if mode_run:
    model, train_info = train_model(
        model,
        criterion,
        optimizer,
        scheduler,
        dataloaders,
        device,
        model_root,
        num_epochs=num_epochs,
        disp=True,
        do_checkpoint=checkpoint_interval,
        tb_path=tb_path,
        tb_freq=tb_freq,
    )
else:
    train_info = {}
```

Save the network and training history

We save the model so that it can later be utilized. We save the network's architecture, the training parameters and the training history.

```
from spyrit.core.train import save_net

# Training parameters
train_type = "N0_{:g}".format(N0)
arch = "pinv-net"
denoi = "cnn"
data = "stl10"
reg = 1e-7  # Default value
suffix = "N_{}_M_{}_epo_{}_lr_{}_sss_{}_sdr_{}_bs_{}".format(
    h, M, num_epochs, lr, step_size, gamma, batch_size
)
title = model_root / f"{arch}_{denoi}_{data}_{train_type}_{suffix}"
print(title)

Path(model_root).mkdir(parents=True, exist_ok=True)

if checkpoint_interval:
    Path(title).mkdir(parents=True, exist_ok=True)

save_net(title, model)

# Save training history
import pickle

if mode_run:
    from spyrit.core.train import Train_par

    params = Train_par(batch_size, lr, h, reg=reg)
    params.set_loss(train_info)

    train_path = model_root / f"TRAIN_{arch}_{denoi}_{data}_{train_type}_{suffix}.pkl"

    with open(train_path, "wb") as param_file:
        pickle.dump(params, param_file)
    torch.cuda.empty_cache()
else:
    # Download training history
    import gdown

    train_path = os.path.join(
        model_root,
        "TRAIN_pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_30_lr_0.001_sss_10_sdr_0.5_bs_512_
↪reg_1e-07.pkl",
    )
    url_train = "https://drive.google.com/file/d/13KIbSEigHBZ8ub_JxMUqwRDMHklnFz8A/view?
↪usp=drive_link"
    gdown.download(url_train, train_path, quiet=False, fuzzy=True)
```

(continues on next page)

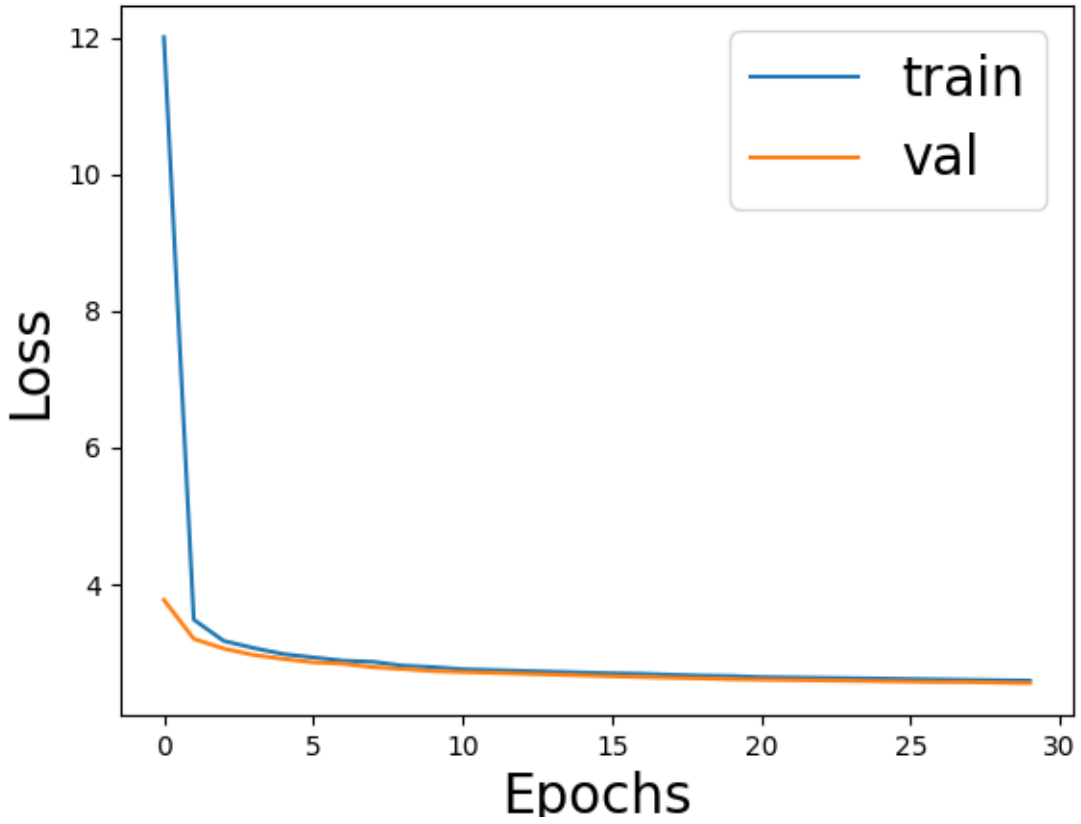
(continued from previous page)

```
    with open(train_path, "rb") as param_file:
        params = pickle.load(param_file)
    train_info["train"] = params.train_loss
    train_info["val"] = params.val_loss
```

```
model/pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_5_lr_0.001_sss_10_sdr_0.5_bs_512
model/pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_5_lr_0.001_sss_10_sdr_0.5_bs_512.pth
Model Saved
Downloading...
From: https://drive.google.com/uc?id=13KIbSEigHBZ8ub_JxMUqwRDMHklnFz8A
To: /home/docs/checkouts/readthedocs.org/user_builds/spyrit/checkouts/master/tutorial/
↪model/TRAIN_pinv-net_cnn_stl10_N0_1_N_64_M_1024_epo_30_lr_0.001_sss_10_sdr_0.5_bs_512_
↪reg_1e-07.pkl

  0%|          | 0.00/714 [00:00<?, ?B/s]
100%|| 714/714 [00:00<00:00, 3.09MB/s]
```

We plot the training loss and validation loss

```
# Plot
# sphinx_gallery_thumbnail_number = 2

fig = plt.figure()
plt.plot(train_info["train"], label="train")
plt.plot(train_info["val"], label="val")
plt.xlabel("Epochs", fontsize=20)
plt.ylabel("Loss", fontsize=20)
plt.legend(fontsize=20)
```




```
<matplotlib.legend.Legend object at 0x7f32d35295d0>
```

Note: See the googlecolab notebook spyrit-examples/tutorial/tuto_train_lin_meas_colab.ipynb for training a reconstruction network on GPU. It shows how to train using different architectures, denoisers and other hyperparameters from *train_model()* function.

Total running time of the script: (0 minutes 13.703 seconds)

05. Acquisition operators (advanced) - Split measurements and subsampling

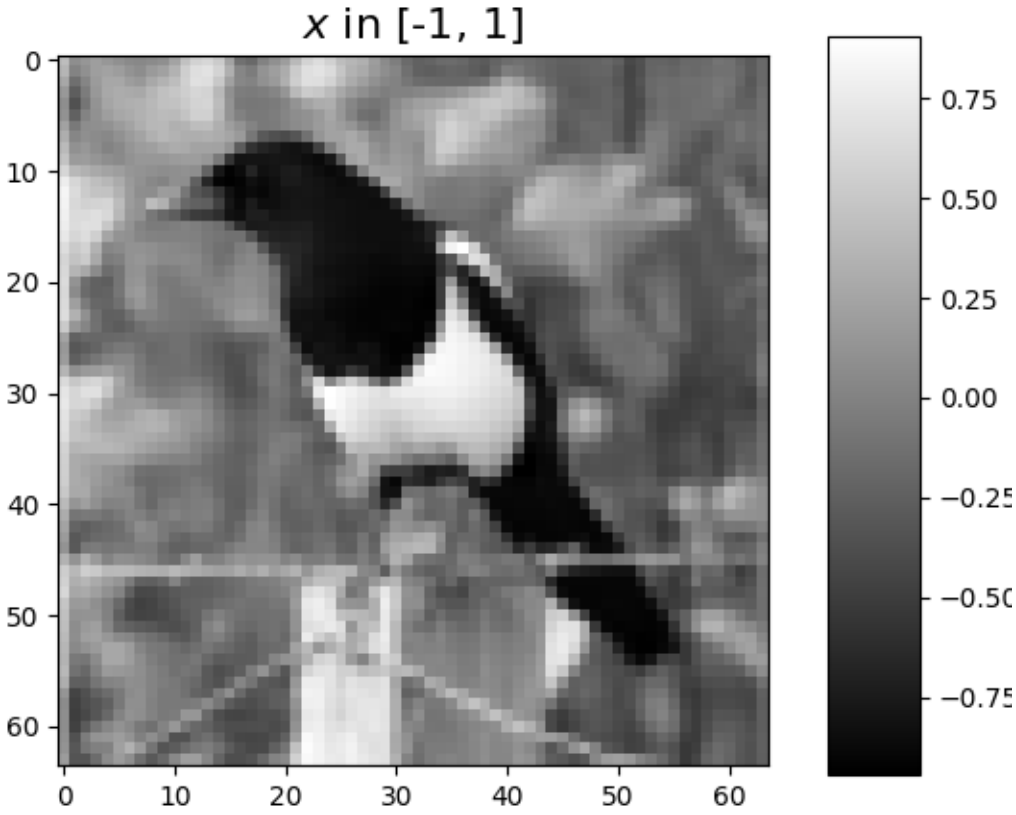
This tutorial is a continuation of the *Acquisition operators tutorial* for single-pixel imaging, which showed how to simulate linear measurements using the *spyrit.core* submodule (based on three classes *spyrit.core.meas*, *spyrit.core.noise*, and *spyrit.core.prep*). This tutorial extends the previous case: i) by introducing split measurements that can handle a Hadamard measurement matrix, and ii) by discussing the choice of the subsampling pattern for accelerated acquisitions.

These tutorials load image samples from */images/*.

Load a batch of images

Images x for training neural networks expect values in [-1,1]. The images are normalized using the `transform_gray_norm()` function.

```
import os

import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt

from spyrit.misc.disp import imagesc
from spyrit.misc.statistics import transform_gray_norm


h = 64  # image size hxh
i = 1  # Image index (modify to change the image)
spyritPath = os.getcwd()
imgs_path = os.path.join(spyritPath, "images/")


# Create a transform for natural images to normalized grayscale image tensors
transform = transform_gray_norm(img_size=h)

# Create dataset and loader (expects class folder 'images/test/')
dataset = torchvision.datasets.ImageFolder(root=imgs_path, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=7)

x, _ = next(iter(dataloader))
print(f"Shape of input images: {x.shape}")

# Select image
x = x[i : i + 1, :, :, :]
x = x.detach().clone()
b, c, h, w = x.shape

# plot
x_plot = x.view(-1, h, h).cpu().numpy()
imagesc(x_plot[0, :, :], r"$x$ in [-1, 1]")
```




```
Shape of input images: torch.Size([7, 1, 64, 64])
```

The measurement and noise operators

Noise operators are defined in the *noise* module. A noise operator computes the following three steps sequentially:

1. Normalization of the image x with values in [-1,1] to get an image $\tilde{x} = \frac{x+1}{2}$ in [0,1], as it is required for measurement simulation
2. Application of the measurement model, i.e., computation of $P\tilde{x}$
3. Application of the noise model

$$y \sim \texttt{Noise}(P\tilde{x}) = \texttt{Noise}\left(\frac{P(x+1)}{2}\right).$$

The normalization is useful when considering distributions such as the Poisson distribution that are defined on positive values.

Split measurement operator and no noise

$$y = P\tilde{x} = \begin{bmatrix} H_+ \\ H_- \end{bmatrix} \tilde{x}.$$

Hadamard split measurement operator is defined in the *spyrit.core.meas.HadamSplit* class. It computes linear measurements from incoming images, where P is a linear operator (matrix) with positive entries and $\tilde{x}$ is a vectorized image. The class relies on a matrix H with shape (M, N) where N represents the number of pixels in the image and $M \leq N$ the number of measurements. The matrix P is obtained by splitting the matrix H as $H = H_+ - H_-$ where $H_+ = \max(0, H)$ and $H_- = \max(0, -H)$.

Subsampling

We simulate an accelerated acquisition by subsampling the measurement matrix. We consider two subsampling strategies:

- "Naive subsampling" by retaining only the first M rows of the measurement matrix.
- "Variance subsampling" by retaining only the first M rows of a permuted measurement matrix where the first rows corresponds to the coefficients with largest variance and the last ones to the coefficients that are close to constant. The motivation is that almost constant coefficients are less informative than the others. This can be supported by principal component analysis, which states that preserving the components with largest variance leads to the best linear predictor.

Subsampling is done by retaining only the first M rows of a permuted Hadamard matrix Perm$H$, where Perm is a permutation matrix with shape with shape (M, N) and H is a "full" Hadamard matrix with shape (N, N) (see Hadamard matrix in *tutorial on pseudoinverse solution*). The permutation matrix Perm is obtained from the ordering matrix Ord with shape (h, h). This is all handled internally by the *spyrit.core.meas.HadamSplit* class.

First, we download the covariance matrix from our warehouse and load it. The covariance matrix has been computed from ImageNet 2012 dataset.

```
import girder_client

# api Rest url of the warehouse
url = "https://pilot-warehouse.creatis.insa-lyon.fr/api/v1"

# Generate the warehouse client
gc = girder_client.GirderClient(apiUrl=url)

# Download the covariance matrix and mean image
data_folder = "./stat/"
dataId_list = [
    "63935b624d15dd536f0484a5",  # for reconstruction (imageNet, 64)
    "63935a224d15dd536f048496",  # for reconstruction (imageNet, 64)
]
cov_name = "./stat/Cov_64x64.npy"

try:
    for dataId in dataId_list:
        myfile = gc.getFile(dataId)
        gc.downloadFile(dataId, data_folder + myfile["name"])
```

(continues on next page)

(continued from previous page)

```
    print(f"Created {data_folder}")

    # Load covariance matrix for "variance subsampling"
    Cov = np.load(cov_name)
    print(f"Cov matrix {cov_name} loaded")
except:
    # Set to the identity if not found for "naive subsampling"
    Cov = np.eye(h * h)
    print(f"Cov matrix {cov_name} not found! Set to the identity")
```

```
/home/docs/checkouts/readthedocs.org/user_builds/spyrit/envs/master/lib/python3.11/site-
↪packages/girder_client/__init__.py:1: DeprecationWarning: pkg_resources is deprecated␣
↪as an API. See https://setuptools.pypa.io/en/latest/pkg_resources.html
  from pkg_resources import DistributionNotFound, get_distribution
/home/docs/checkouts/readthedocs.org/user_builds/spyrit/envs/master/lib/python3.11/site-
↪packages/pkg_resources/__init__.py:2832: DeprecationWarning: Deprecated call to `pkg_
↪resources.declare_namespace('sphinxcontrib')`.
Implementing implicit namespace packages (as specified in PEP 420) is preferred to `pkg_
↪resources.declare_namespace`. See https://setuptools.pypa.io/en/latest/references/
↪keywords.html#keyword-namespace-packages
  declare_namespace(pkg)
Created ./stat/
Cov matrix ./stat/Cov_64x64.npy loaded
```

We compute the order matrix Ord for the two sampling strategies, from the covariance matrix for the "variance subsampling", and from the identity matrix for the "naive subsampling". In the latter case, the order matrix is constant, as all coefficients are considered equally informative, and they are retained in the increasing 'naive' order. We also define the number of measurements M that will be used later.

```
from spyrit.misc.statistics import Cov2Var
from spyrit.misc.disp import add_colorbar, noaxis

# number of measurements (here, 1/4 of the pixels)
M = 64 * 64 // 4

# Compute the order matrix
# "Naive subsampling"
Cov_eye = np.eye(h * h)
Ord_nai = Cov2Var(Cov_eye)

# "Variance subsampling"
Ord_var = Cov2Var(Cov)
```

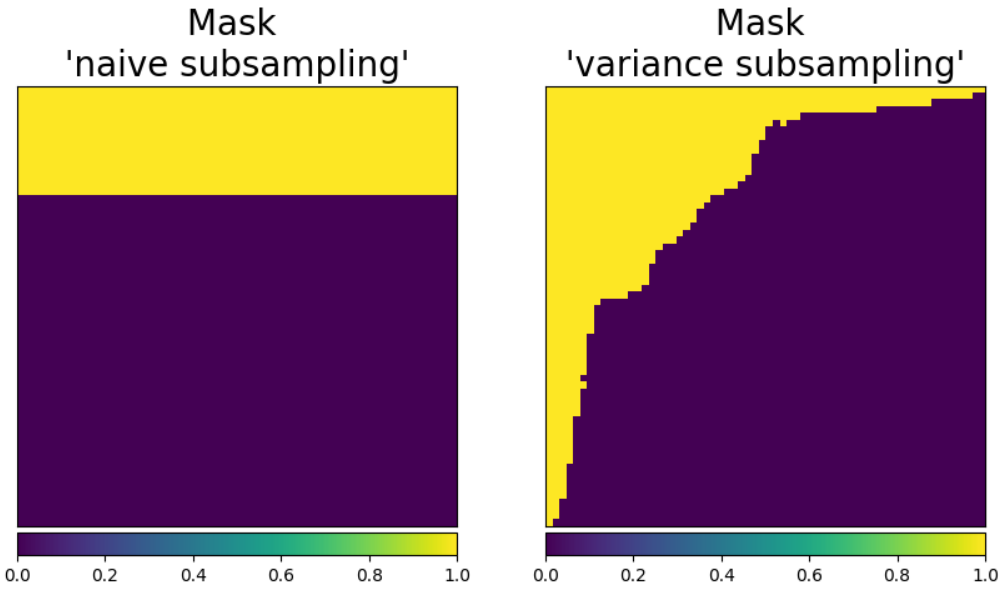
To provide further insight on the subsampling strategies, we can plot an approximation of the masks that are used to subsample the measurement matrices.

```
# sphinx_gallery_thumbnail_number = 2

# Mask for "naive subsampling"
mask_nai = np.zeros((h, h))
mask_nai[0 : int(M / h), :] = 1
```

(continues on next page)

(continued from previous page)

```
# Mas for "variance subsampling"
idx = np.argsort(Ord_var.ravel(), axis=None)[::-1]
mask_var = np.zeros_like(Ord_var)
mask_var.flat[idx[0:M]] = 1

# Plot the masks
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
im1 = ax1.imshow(mask_nai, vmin=0, vmax=1)
ax1.set_title("Mask \n'naive subsampling'", fontsize=20)
noaxis(ax1)
add_colorbar(im1, "bottom", size="20%")

im2 = ax2.imshow(mask_var, vmin=0, vmax=1)
ax2.set_title("Mask \n'variance subsampling'", fontsize=20)
noaxis(ax2)
add_colorbar(im2, "bottom", size="20%")
```




```
<matplotlib.colorbar.Colorbar object at 0x7f32cc8ba6d0>
```

Note: Note that in this tutorial the covariance matrix is used only for chosing the subsampling strategy. Although the covariance matrix can be also exploited to improve the reconstruction, this will be considered in a future tutorial.

Measurement and noise operators

We compute the measurement and noise operators and then simulate the measurement vector y.

We consider Poisson noise, i.e., a noisy measurement vector given by

$$y \sim \mathcal{P}(\alpha P \tilde{x}),$$

where α is a scalar value that represents the maximum image intensity (in photons). The larger α, the higher the signal-to-noise ratio.

We use the *spyrit.core.noise.Poisson* class, set α to 100 photons, and simulate a noisy measurement vector for the two sampling strategies.

```
from spyrit.core.noise import Poisson
from spyrit.core.meas import HadamSplit
from spyrit.core.noise import Poisson

alpha = 100.0  # number of photons

# "Naive subsampling"
# Measurement and noise operators
meas_nai_op = HadamSplit(M, h, torch.from_numpy(Ord_nai))
noise_nai_op = Poisson(meas_nai_op, alpha)

# Measurement operator
x = x.view(b * c, h * w)  # vectorized image
y_nai = noise_nai_op(x)  # a noisy measurement vector

# "Variance subsampling"
meas_var_op = HadamSplit(M, h, torch.from_numpy(Ord_var))
noise_var_op = Poisson(meas_var_op, alpha)
y_var = noise_var_op(x)  # a noisy measurement vector

x = x.view(b * c, h * w)  # vectorized image
print(f"Shape of vectorized image: {x.shape}")
print(f"Shape of simulated measurements y: {y_var.shape}")
```

```
Shape of vectorized image: torch.Size([1, 4096])
Shape of simulated measurements y: torch.Size([1, 2048])
```

The preprocessing operator measurements for split measurements

We compute the preprocessing operators for the three cases considered above, using the *spyrit.core.prep* module. As previously introduced, a preprocessing operator applies to the noisy measurements in order to compensate for the scaling factors that appear in the measurement or noise operators:

$$m = \texttt{Prep}(y),$$

We consider the *spyrit.core.prep.SplitPoisson* class that intends to "undo" the *spyrit.core.noise.Poisson* class, for split measurements, by compensating for

- the scaling that appears when computing Poisson-corrupted measurements
- the affine transformation to get images in [0,1] from images in [-1,1]

For this, it computes

$$m = \frac{2(y_+ - y_-)}{\alpha} - P\mathbb{1},$$

where $y_+ = H_+\tilde{x}$ and $y_- = H_-\tilde{x}$. This is handled internally by the *spyrit.core.prep.SplitPoisson* class.

We compute the preprocessing operator and the measurements vectors for the two sampling strategies.

```
from spyrit.core.prep import SplitPoisson

# "Naive subsampling"
#
# Preprocessing operator
prep_nai_op = SplitPoisson(alpha, meas_nai_op)

# Preprocessed measurements
m_nai = prep_nai_op(y_nai)

# "Variance subsampling"
prep_var_op = SplitPoisson(alpha, meas_var_op)
m_var = prep_var_op(y_var)
```

Noiseless measurements

We consider now noiseless measurements for the "naive subsampling" strategy. We compute the required operators and the noiseless measurement vector. For this we use the *spyrit.core.noise.NoNoise* class, which normalizes the input vector to get an image in [0,1], as explained in *acquisition operators tutorial*. For the preprocessing operator, we assign the number of photons equal to one.

```
from spyrit.core.noise import NoNoise

nonoise_nai_op = NoNoise(meas_nai_op)
y_nai_nonoise = nonoise_nai_op(x)  # a noisy measurement vector

prep_nonoise_op = SplitPoisson(1.0, meas_nai_op)
m_nai_nonoise = prep_nonoise_op(y_nai_nonoise)
```

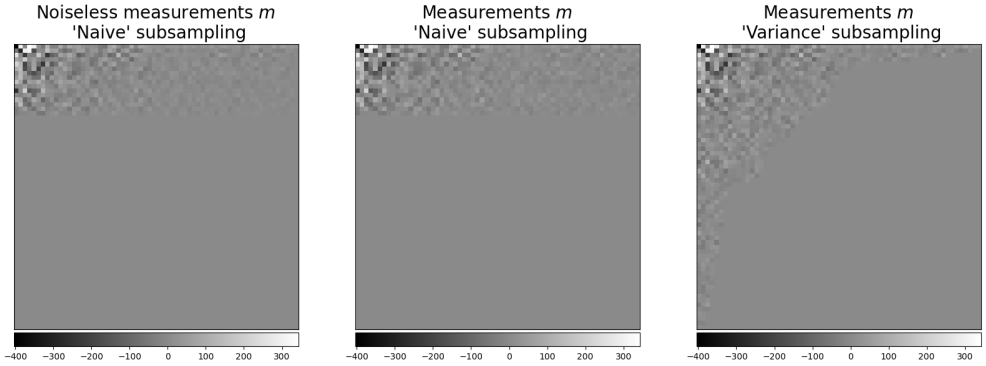
We can now plot the three measurement vectors

```
from spyrit.misc.sampling import meas2img2

# Plot the three measurement vectors
m_plot = m_nai_nonoise.numpy()
m_plot = meas2img2(m_plot.T, Ord_nai)
m_plot = np.moveaxis(m_plot, -1, 0)
m_plot_max = np.max(m_plot[0, :, :])
m_plot_min = np.min(m_plot[0, :, :])

m_plot2 = m_nai.numpy()
m_plot2 = meas2img2(m_plot2.T, Ord_nai)
m_plot2 = np.moveaxis(m_plot2, -1, 0)

m_plot3 = m_var.numpy()
```

(continues on next page)

(continued from previous page)

```
m_plot3 = meas2img2(m_plot3.T, Ord_var)
m_plot3 = np.moveaxis(m_plot3, -1, 0)

f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 7))
im1 = ax1.imshow(m_plot[0, :, :], cmap="gray")
ax1.set_title("Noiseless measurements $m$ \n 'Naive' subsampling", fontsize=20)
noaxis(ax1)
add_colorbar(im1, "bottom", size="20%")

im2 = ax2.imshow(m_plot2[0, :, :], cmap="gray", vmin=m_plot_min, vmax=m_plot_max)
ax2.set_title("Measurements $m$ \n 'Naive' subsampling", fontsize=20)
noaxis(ax2)
add_colorbar(im2, "bottom", size="20%")

im3 = ax3.imshow(m_plot3[0, :, :], cmap="gray", vmin=m_plot_min, vmax=m_plot_max)
ax3.set_title("Measurements $m$ \n 'Variance' subsampling", fontsize=20)
noaxis(ax3)
add_colorbar(im3, "bottom", size="20%")
```




```
<matplotlib.colorbar.Colorbar object at 0x7f32ccb021d0>
```

PinvNet network

We use the *spyrit.core.recon.PinvNet* class where the pseudo inverse reconstruction is performed by a neural network

```
from spyrit.core.recon import PinvNet

# PinvNet(meas_op, prep_op, denoi=torch.nn.Identity())
pinvnet_nai_nonoise = PinvNet(nonoise_nai_op, prep_nonoise_op)
pinvnet_nai = PinvNet(noise_nai_op, prep_nai_op)
pinvnet_var = PinvNet(noise_var_op, prep_var_op)

# Reconstruction
z_nai_nonoise = pinvnet_nai_nonoise.reconstruct(y_nai_nonoise)
z_nai = pinvnet_nai.reconstruct(y_nai)
```

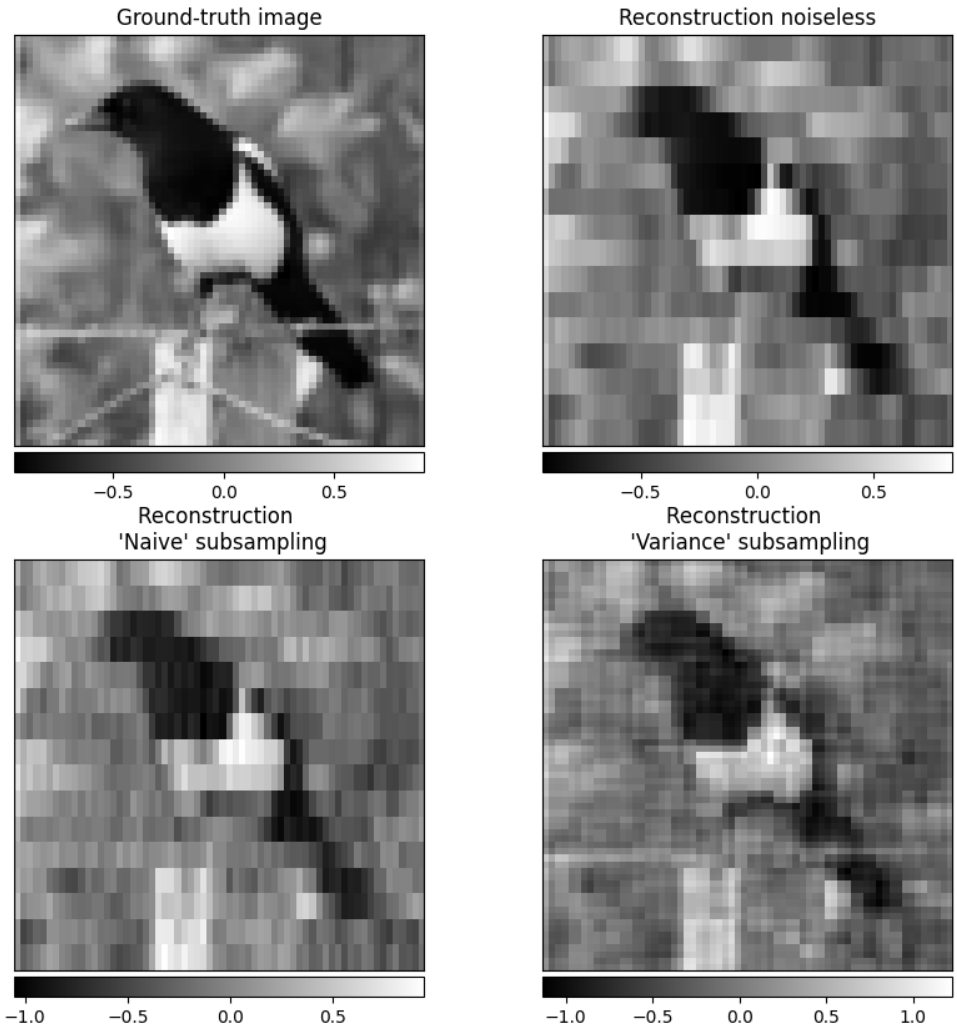
(continues on next page)

(continued from previous page)

```
z_var = pinvnet_var.reconstruct(y_var)
```

We can now plot the three reconstructed images

```
from spyrit.misc.disp import add_colorbar, noaxis

# Plot
x_plot = x.view(-1, h, h).numpy()
z_plot_nai_nonoise = z_nai_nonoise.view(-1, h, h).numpy()
z_plot_nai = z_nai.view(-1, h, h).numpy()
z_plot_var = z_var.view(-1, h, h).numpy()

f, axs = plt.subplots(2, 2, figsize=(10, 10))
im1 = axs[0, 0].imshow(x_plot[0, :, :], cmap="gray")
axs[0, 0].set_title("Ground-truth image")
noaxis(axs[0, 0])
add_colorbar(im1, "bottom")

im2 = axs[0, 1].imshow(z_plot_nai_nonoise[0, :, :], cmap="gray")
axs[0, 1].set_title("Reconstruction noiseless")
noaxis(axs[0, 1])
add_colorbar(im2, "bottom")

im3 = axs[1, 0].imshow(z_plot_nai[0, :, :], cmap="gray")
axs[1, 0].set_title("Reconstruction \n 'Naive' subsampling")
noaxis(axs[1, 0])
add_colorbar(im3, "bottom")

im4 = axs[1, 1].imshow(z_plot_var[0, :, :], cmap="gray")
axs[1, 1].set_title("Reconstruction \n 'Variance' subsampling")
noaxis(axs[1, 1])
add_colorbar(im4, "bottom")

plt.show()
```



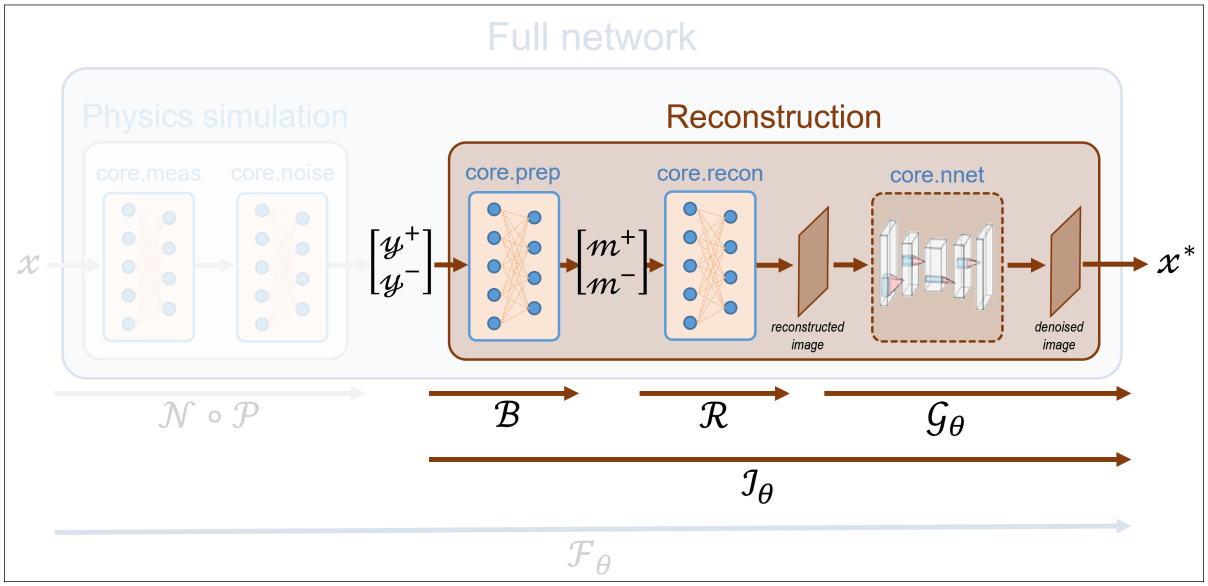
Note: Note that reconstructed images are pixelized when using the “naive subsampling”, while they are smoother and more similar to the ground-truth image when using the “variance subsampling”.

Another way to further improve results is to include a nonlinear post-processing step, which we will consider in a future tutorial.

Total running time of the script: (0 minutes 26.324 seconds)

06. DCNet solution for split measurements

This tutorial shows how to perform image reconstruction using DCNet (denoised completion network) with and without a trainable image denoiser. In the previous tutorial *Acquisition - split measurements* we showed how to handle split measurements for a Hadamard operator and how to perform a pseudo-inverse reconstruction with PinvNet.



These tutorials load image samples from */images/*.

Load a batch of images

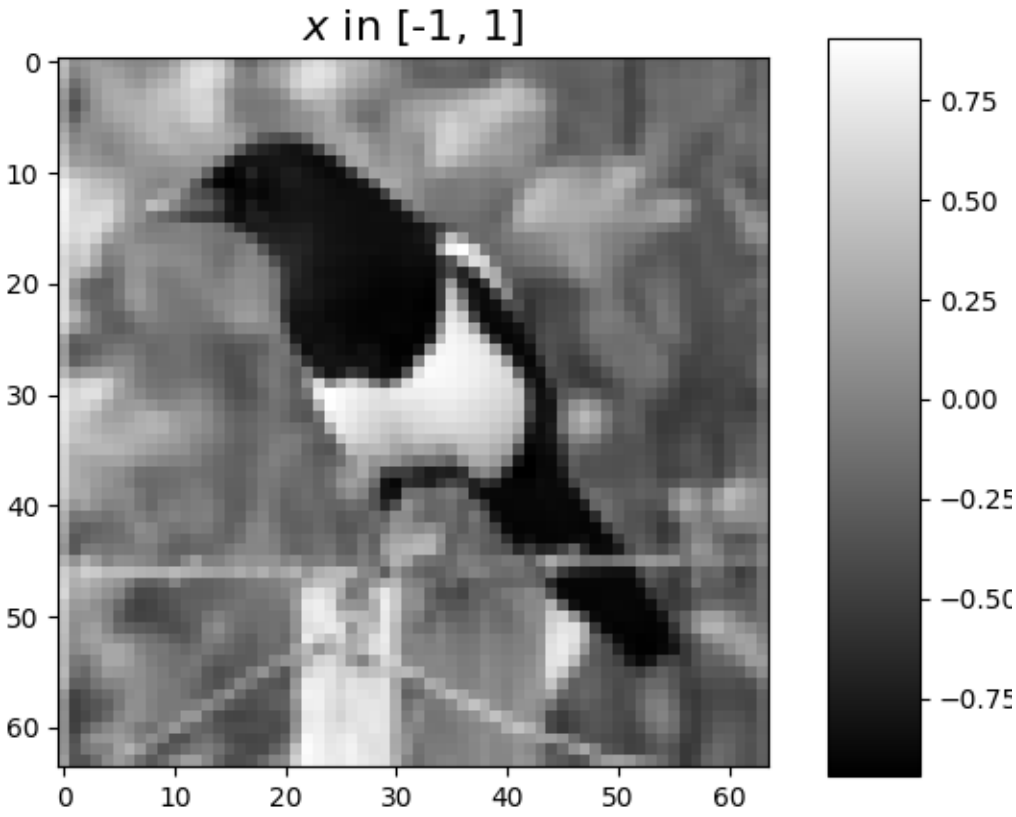
Images x for training neural networks expect values in [-1,1]. The images are normalized using the `transform_gray_norm()` function.

```
import os

import torch
import torchvision
import numpy as np
import matplotlib.pyplot as plt

from spyrit.misc.disp import imagesc
from spyrit.misc.statistics import transform_gray_norm

# sphinx_gallery_thumbnail_path = 'fig/tuto6.png'

h = 64  # image size hxh
i = 1  # Image index (modify to change the image)
spyritPath = os.getcwd()
imgs_path = os.path.join(spyritPath, "images/")


# Create a transform for natural images to normalized grayscale image tensors
transform = transform_gray_norm(img_size=h)
```

(continues on next page)

(continued from previous page)

```
# Create dataset and loader (expects class folder 'images/test/')
dataset = torchvision.datasets.ImageFolder(root=imgs_path, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=7)

x, _ = next(iter(dataloader))
print(f"Shape of input images: {x.shape}")

# Select image
x = x[i : i + 1, :, :, :]
x = x.detach().clone()
b, c, h, w = x.shape

# plot
x_plot = x.view(-1, h, h).cpu().numpy()
imagesc(x_plot[0, :, :], r"$x$ in [-1, 1]")
```




```
Shape of input images: torch.Size([7, 1, 64, 64])
```

Forward operators for split measurements

We consider noisy split measurements for a Hadamard operator and a "variance subsampling" strategy that preserves the coefficients with the largest variance, obtained from a previously estimated covariance matrix (for more details, refer to *Acquisition - split measurements*).

First, we download the covariance matrix and load it.

```
import girder_client

# api Rest url of the warehouse
url = "https://pilot-warehouse.creatis.insa-lyon.fr/api/v1"

# Generate the warehouse client
gc = girder_client.GirderClient(apiUrl=url)

# Download the covariance matrix and mean image
data_folder = "./stat/"
dataId_list = [
    "63935b624d15dd536f0484a5",  # for reconstruction (imageNet, 64)
    "63935a224d15dd536f048496",  # for reconstruction (imageNet, 64)
]
cov_name = "./stat/Cov_64x64.npy"

try:
    Cov = np.load(cov_name)
    print(f"Cov matrix {cov_name} loaded")
except FileNotFoundError:
    for dataId in dataId_list:
        myfile = gc.getFile(dataId)
        gc.downloadFile(dataId, data_folder + myfile["name"])

    print(f"Created {data_folder}")

    Cov = np.load(cov_name)
    print(f"Cov matrix {cov_name} loaded")
except:
    Cov = np.eye(h * h)
    print(f"Cov matrix {cov_name} not found! Set to the identity")
```

```
Cov matrix ./stat/Cov_64x64.npy loaded
```

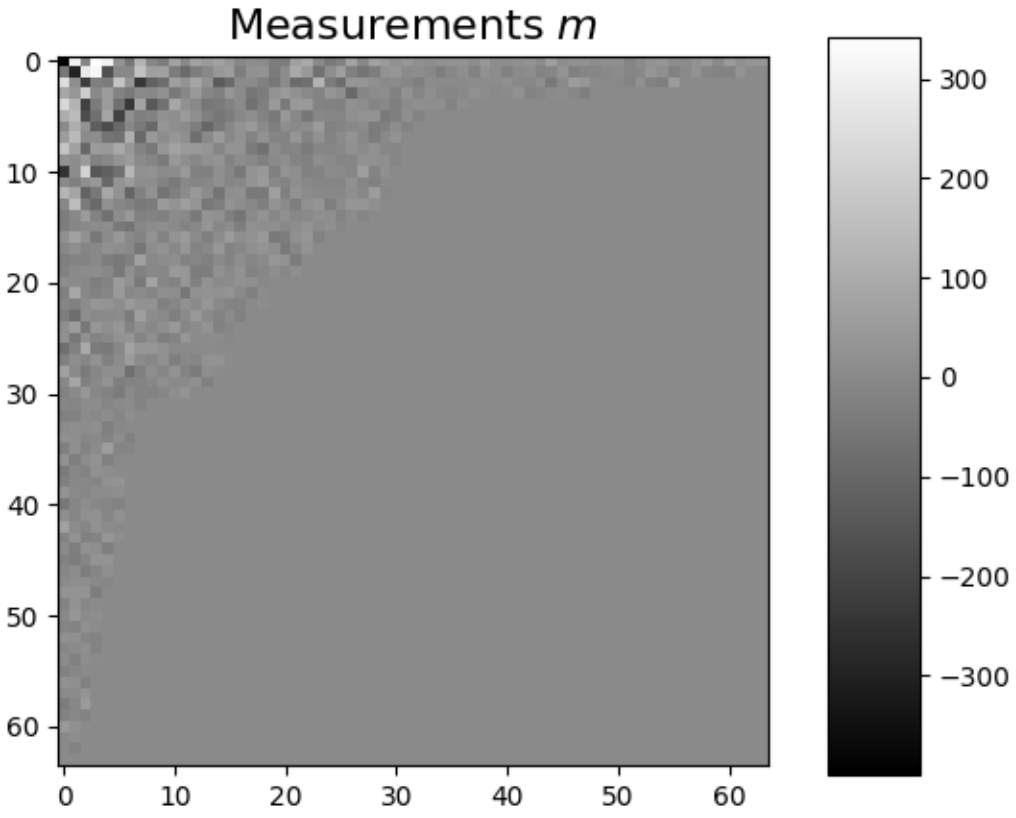
We define the measurement, noise and preprocessing operators and then simulate a noiseless measurement vector y. As in the previous tutorial, we simulate an accelerated acquisition by subsampling the measurement matrix by retaining only the first M rows of a Hadamard matrix Perm$H$.

```
from spyrit.core.meas import HadamSplit
from spyrit.core.noise import Poisson
from spyrit.misc.sampling import meas2img2
from spyrit.misc.statistics import Cov2Var
from spyrit.core.prep import SplitPoisson

# Measurement parameters
M = 64 * 64 // 4  # Number of measurements (here, 1/4 of the pixels)
```

(continues on next page)

(continued from previous page)

```
alpha = 100.0  # number of photons

# Ordering matrix
Ord = Cov2Var(Cov)

# Measurement and noise operators
meas_op = HadamSplit(M, h, torch.from_numpy(Ord))
noise_op = Poisson(meas_op, alpha)
prep_op = SplitPoisson(alpha, meas_op)

# Vectorize image
x = x.view(b * c, h * w)
print(f"Shape of vectorized image: {x.shape}")

# Measurements
y = noise_op(x)  # a noisy measurement vector
m = prep_op(y)  # preprocessed measurement vector

m_plot = m.detach().numpy()
m_plot = meas2img2(m_plot.T, Ord)
imagesc(m_plot, r"Measurements $m$")
```



```
Shape of vectorized image: torch.Size([1, 4096])
```

PinvNet network

We reconstruct with the pseudo inverse using *spyrit.core.recon.PinvNet* class as in the previous tutorial. For this, we define the neural network and then perform the reconstruction.

```
from spyrit.core.recon import PinvNet
from spyrit.misc.disp import add_colorbar, noaxis

# Reconstruction with for Core module (linear net)
pinvnet = PinvNet(noise_op, prep_op)

# use GPU, if available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Pseudo-inverse net
pinvnet = pinvnet.to(device)

# Reconstruction
with torch.no_grad():
    z_invnet = pinvnet.reconstruct(y.to(device))  # reconstruct from raw measurements
```

DCNet network

We can improve PinvNet results by using the *denoised* completion network DCNet with the *spyrit.core.recon.DCNet* class. It has four sequential steps:

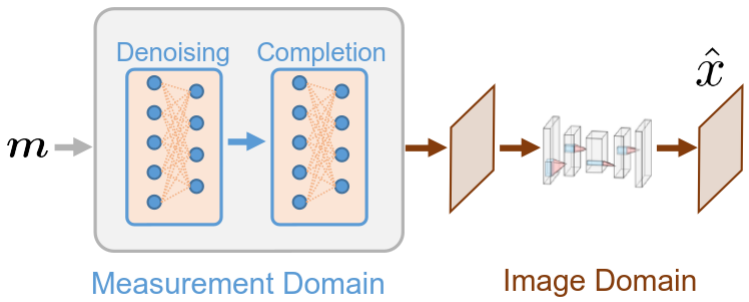
i) denoising of the acquired measurements,

ii) estimation of the missing measurements from the denoised ones,

iii) mapping them to the image domain, and

iv) denoising in the image-domain.

Only the last step involves learnable parameters.



For the denoiser, we compare the default unit matrix (no denoising) with the UNet denoiser with the *spyrit.core.nnet.Unet* class. For the latter, we load the pretrained model weights.

Without *learnable image-domain* denoising

```
from spyrit.core.recon import DCNet
from spyrit.core.nnet import Unet
from torch import nn

# Reconstruction with for DCNet (linear net)
dcnet = DCNet(noise_op, prep_op, torch.from_numpy(Cov), denoi=nn.Identity())
dcnet = dcnet.to(device)

# Reconstruction
with torch.no_grad():
    z_dcnet = dcnet.reconstruct(y.to(device))  # reconstruct from raw measurements
```

With a UNet denoising layer, we define the denoising network and then load the pretrained weights.

```
from spyrit.core.train import load_net
import matplotlib.pyplot as plt
from spyrit.misc.disp import add_colorbar, noaxis

# Define UNet denoiser
denoi = Unet()

# Define DCNet (with UNet denoising)
dcnet_unet = DCNet(noise_op, prep_op, torch.from_numpy(Cov), denoi)
dcnet_unet = dcnet_unet.to(device)

# Load previously trained model
# Download weights
url_unet = "https://drive.google.com/file/d/15PRRZj5OxKpn1iJw78lGwUUBtTbFco1l/view?
↪usp=drive_link"
model_path = "./model"
if os.path.exists(model_path) is False:
    os.mkdir(model_path)
    print(f"Created {model_path}")
model_unet_path = os.path.join(
    model_path,
    "dc-net_unet_stl10_N0_100_N_64_M_1024_epo_30_lr_0.001_sss_10_sdr_0.5_bs_512_reg_1e-
↪07.pth",
)

load_unet = True
if os.path.exists(model_unet_path) is False:
    try:
        import gdown

        gdown.download(url_unet, f"{model_unet_path}.pth", quiet=False, fuzzy=True)
    except:
        print(f"Model {model_unet_path} not found!")
        load_unet = False

if load_unet:
    # Load pretrained model
    load_net(model_unet_path, dcnet_unet, device, False)
    # print(f"Model {model_unet_path} loaded.")
```

(continues on next page)

(continued from previous page)

```
# Reconstruction
with torch.no_grad():
    z_dcnet_unet = dcnet_unet.reconstruct(
        y.to(device)
    )  # reconstruct from raw measurements
```

```
Downloading...
From: https://drive.google.com/uc?id=15PRRZj5OxKpn1iJw78lGwUUBtTbFco1l
To: /home/docs/checkouts/readthedocs.org/user_builds/spyrit/checkouts/master/tutorial/
↪model/dc-net_unet_stl10_N0_100_N_64_M_1024_epo_30_lr_0.001_sss_10_sdr_0.5_bs_512_reg_
↪1e-07.pth.pth

  0%|          | 0.00/149M [00:00<?, ?B/s]
  1%|          | 1.57M/149M [00:00<00:09, 14.7MB/s]
  4%|         | 5.77M/149M [00:00<00:04, 30.2MB/s]
  6%|         | 8.91M/149M [00:00<00:08, 17.4MB/s]
 12%|        | 17.3M/149M [00:00<00:05, 25.7MB/s]
 17%|        | 25.7M/149M [00:00<00:04, 29.3MB/s]
 23%|       | 34.1M/149M [00:01<00:05, 22.7MB/s]
 29%|       | 42.5M/149M [00:01<00:03, 28.2MB/s]
 34%|      | 50.9M/149M [00:01<00:02, 34.8MB/s]
 40%|      | 59.2M/149M [00:01<00:02, 39.0MB/s]
 45%|     | 67.6M/149M [00:02<00:01, 42.7MB/s]
 51%|     | 76.0M/149M [00:02<00:01, 46.8MB/s]
 57%|    | 84.4M/149M [00:02<00:01, 49.8MB/s]
 62%|   | 92.8M/149M [00:02<00:01, 49.8MB/s]
 68%|   | 101M/149M [00:02<00:00, 51.5MB/s]
 74%|  | 110M/149M [00:02<00:00, 51.2MB/s]
 80%|  | 120M/149M [00:02<00:00, 60.9MB/s]
 85%| | 126M/149M [00:03<00:00, 59.5MB/s]
 91%| | 135M/149M [00:03<00:00, 45.5MB/s]
 96%|| 143M/149M [00:03<00:00, 45.8MB/s]
100%|| 149M/149M [00:03<00:00, 41.4MB/s]
Model Loaded: ./model/dc-net_unet_stl10_N0_100_N_64_M_1024_epo_30_lr_0.001_sss_10_sdr_0.
↪5_bs_512_reg_1e-07.pth
```

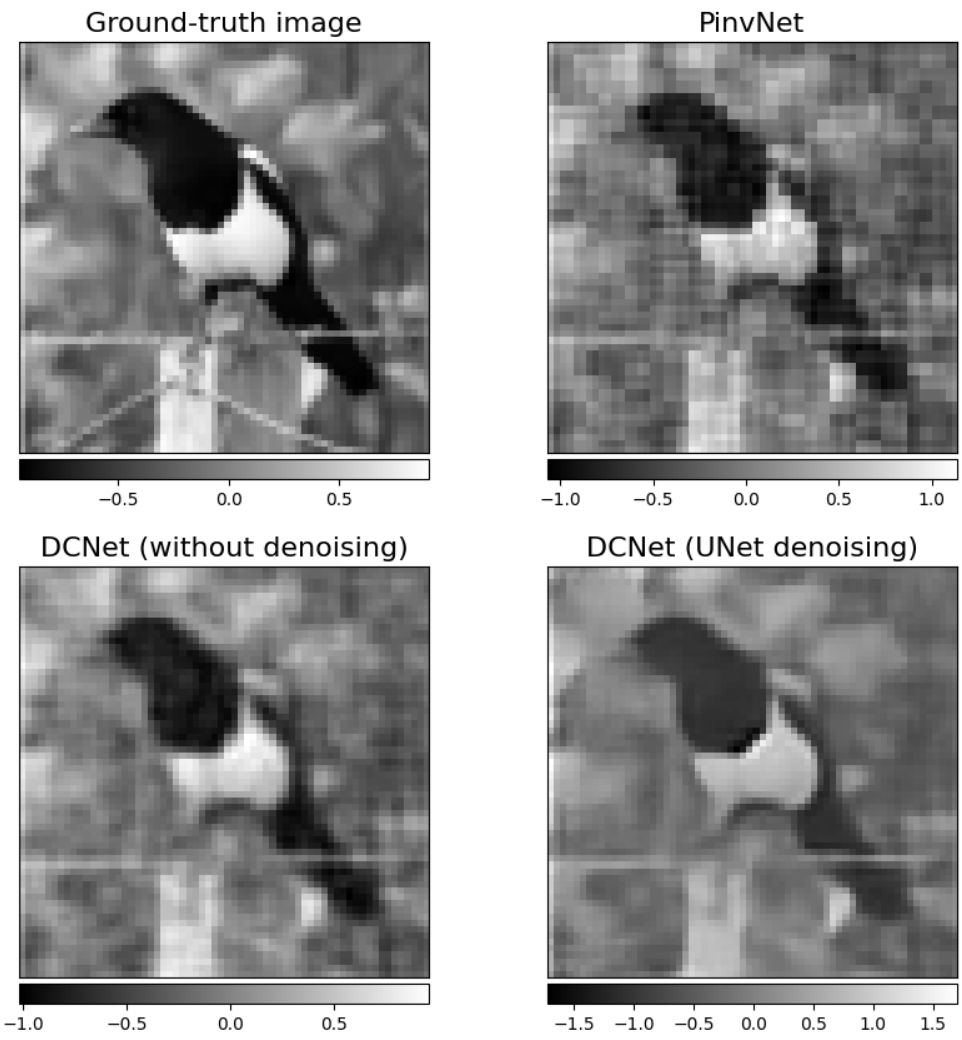
We plot all results

```
# plot reconstruction side by side
x_plot = x.view(-1, h, h).cpu().numpy()
x_plot2 = z_invnet.view(-1, h, h).cpu().numpy()
x_plot3 = z_dcnet.view(-1, h, h).cpu().numpy()
x_plot4 = z_dcnet_unet.view(-1, h, h).cpu().numpy()
f, axs = plt.subplots(2, 2, figsize=(10, 10))
im1 = axs[0, 0].imshow(x_plot[0, :, :], cmap="gray")
axs[0, 0].set_title("Ground-truth image", fontsize=16)
noaxis(axs[0, 0])
add_colorbar(im1, "bottom")

im2 = axs[0, 1].imshow(x_plot2[0, :, :], cmap="gray")
```

(continues on next page)

(continued from previous page)

```
axs[0, 1].set_title("PinvNet", fontsize=16)
noaxis(axs[0, 1])
add_colorbar(im2, "bottom")

im3 = axs[1, 0].imshow(x_plot3[0, :, :], cmap="gray")
axs[1, 0].set_title(f"DCNet (without denoising)", fontsize=16)
noaxis(axs[1, 0])
add_colorbar(im3, "bottom")

im4 = axs[1, 1].imshow(x_plot4[0, :, :], cmap="gray")
axs[1, 1].set_title(f"DCNet (UNet denoising)", fontsize=16)
noaxis(axs[1, 1])
add_colorbar(im4, "bottom")

plt.show()
```



Comparing results, PinvNet provides pixelized reconstruction, DCNet with no denoising leads to a smoother reconstruction, as expected by a Tikonov regularization, and DCNet with UNet denoising provides the best reconstruction.

Note: In this tutorial, we have used DCNet with a UNet denoising layer for split measurements. We refer to spyrit-examples tutorials for a comparison of different solutions for split measurements (pinvNet, DCNet and DRUNet).

Total running time of the script: (0 minutes 6.103 seconds)

Bonus. Advanced methods - Colab

We refer to spyrit-examples/tutorial for a list of tutorials that can be run directly in colab and present more advanced cases than the main spyrit tutorials, such as comparison of methods for split measurements, or comparison of different denoising networks.

The spyrit-examples repository also includes research contributions based on the SPYRIT toolbox.

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER

THREE

CITE US

When using SPyRiT in scientific publications, please cite the following paper:

- G. Beneti-Martin, L Mahieu-Williame, T Baudier, N Ducros, "OpenSpyrit: an Ecosystem for Reproducible Single-Pixel Hyperspectral Imaging," Optics Express, Vol. 31, No. 10, (2023). DOI.

When using SPyRiT specifically for the denoised completion network, please cite the following paper:

- A Lorente Mur, P Leclerc, F Peyrin, and N Ducros, "Single-pixel image reconstruction from experimental data using neural networks," Opt. Express 29, 17097-17110 (2021). DOI.

CHAPTER

FOUR

JOIN THE PROJECT

Feel free to contact us by *e-mail <mailto:nicolas.ducros@creatis.insa-lyon.fr>* for any question. Active developers are currently Nicolas Ducros, Thomas Baudier, Juan Abascal and Romain Phan. Direct contributions via pull requests (PRs) are welcome.

The full list of contributors can be found here.

PYTHON MODULE INDEX

S

INDEX

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W