torchkbnufft

Release v0.0.0

Matthew Muckley

Nov 23, 2022

USER GUIDE

Documentation | GitHub | Notebook Examples

CHAPTER

ONE

ABOUT

`torchkbnufft` implements a non-uniform Fast Fourier Transform [1, 2] with Kaiser-Bessel gridding in PyTorch. The implementation is completely in Python, facilitating flexible deployment in readable code with no compilation. NUFFT functions are each wrapped as a `torch.autograd.Function`, allowing backpropagation through NUFFT operators for training neural networks.

This package was inspired in large part by the NUFFT implementation in the Michigan Image Reconstruction Toolbox (Matlab).

CHAPTER

TWO

INSTALLATION

Simple installation can be done via PyPI:

```
pip install torchkbnufft
```

`torchkbnufft` only requires `numpy`, `scipy`, and `torch` as dependencies.

CHAPTER

THREE

OPERATION MODES AND STAGES

The package has three major classes of NUFFT operation mode: table-based NUFFT interpolation, sparse matrix-based NUFFT interpolation, and forward/backward operators with Toeplitz-embedded FFTs [3]. Table interpolation is the standard operation mode, whereas the Toeplitz method is always the fastest for forward/backward NUFFTs. For some problems, sparse matrices may be fast. It is generally best to start with Table interpolation and then experiment with the other modes for your problem.

Sensitivity maps can be incorporated by passing them into a `KbNufft` or `KbNufftAdjoint` object. Auxiliary functions for calculating sparse interpolation matrices, density compensation functions, and Toeplitz filter kernels are also included.

For examples, see *Basic Usage*.

CHAPTER

FOUR

REFERENCES

1. Fessler, J. A., & Sutton, B. P. (2003). Nonuniform fast Fourier transforms using min-max interpolation. *IEEE Transactions on Signal Processing*, 51(2), 560-574.
2. Beatty, P. J., Nishimura, D. G., & Pauly, J. M. (2005). Rapid gridding reconstruction with a minimal oversampling ratio. *IEEE Transactions on Medical Imaging*, 24(6), 799-808.
3. Feichtinger, H. G., Gr, K., & Strohmer, T. (1995). Efficient numerical methods in non-uniform sampling theory. *Numerische Mathematik*, 69(4), 423-440.

4.1 Basic Usage

`torchkbnufft` works primarily via PyTorch modules. You create a module with the properties of your imaging setup. The module will calculate a Kaiser-Bessel kernel and some interpolation parameters based on your inputs. Then, you apply the module to your data stored as PyTorch tensors. NUFFT operations are wrapped in `torch.autograd.Function` classes for backpropagation and training neural networks.

The following code loads a Shepp-Logan phantom and computes a single radial spoke of k-space data:

```
import torch
import torchkbnufft as tkbn
import numpy as np
from skimage.data import shepp_logan_phantom

x = shepp_logan_phantom().astype(np.complex)
im_size = x.shape
# convert to tensor, unsqueeze batch and coil dimension
# output size: (1, 1, ny, nx)
x = torch.tensor(x).unsqueeze(0).unsqueeze(0).to(torch.complex64)

klength = 64
ktraj = np.stack(
   (np.zeros(64), np.linspace(-np.pi, np.pi, klength))
)
# convert to tensor, unsqueeze batch dimension
# output size: (2, klength)
ktraj = torch.tensor(ktraj).to(torch.float)

nufft_ob = tkbn.KbNufft(im_size=im_size)
# outputs a (1, 1, klength) vector of k-space data
kdata = nufft_ob(x, ktraj)
```

The package also includes utilities for working with SENSE-NUFFT operators. The above code can be modified to include sensitivity maps.

```
smaps = torch.rand(1, 8, 400, 400) + 1j * torch.rand(1, 8, 400, 400)
sense_data = nufft_ob(x, ktraj, smaps=smaps.to(x))
```

This code first multiplies by the sensitivity coils in `smaps`, then computes a 64-length radial spoke for each coil. All operations are broadcast across coils, which minimizes interaction with the Python interpreter, helping computation speed.

Sparse matrices are an alternative to table interpolation. Their speed can vary, but they are a bit more accurate than standard table mode. The following code calculates sparse interpolation matrices and uses them to compute a single radial spoke of k-space data:

```
adjnufft_ob = tkbn.KbNufftAdjoint(im_size=im_size)

# precompute the sparse interpolation matrices
interp_mats = tkbn.calc_tensor_spmatrix(
   ktraj,
   im_size=im_size
)

# use sparse matrices in adjoint
image = adjnufft_ob(kdata, ktraj, interp_mats)
```

Sparse matrix multiplication is only implemented for real numbers in PyTorch, which can limit their speed.

The package includes routines for calculating embedded Toeplitz kernels and using them as FFT filters for the forward/backward NUFFT operations. This is very useful for gradient descent algorithms that must use the forward/backward ops in calculating the gradient. The following code shows an example:

```
toep_ob = tkbn.ToepNufft()

# precompute the embedded Toeplitz FFT kernel
kernel = tkbn.calc_toeplitz_kernel(ktraj, im_size)

# use FFT kernel from embedded Toeplitz matrix
image = toep_ob(image, kernel)
```

All of the examples included in this repository can be run on the GPU by sending the NUFFT object and data to the GPU prior to the function call, e.g.,

```
adjnufft_ob = adjnufft_ob.to(torch.device('cuda'))
kdata = kdata.to(torch.device('cuda'))
ktraj = ktraj.to(torch.device('cuda'))

image = adjnufft_ob(kdata, ktraj)
```

Similar to programming low-level code, PyTorch will throw errors if the underlying `dtype` and `device` of all objects are not matching. Be sure to make sure your data and NUFFT objects are on the right device and in the right format to avoid these errors.

For more details, please examine the API in *torchkbnufft* or check out the notebooks below on Google Colab.

- Basic Example
- SENSE-NUFFT Example

- Sparse Matrix Example
- Toeplitz Example

4.2 Performance Tips

`torchkbnufft` is primarily written for the goal of scaling parallelism within the PyTorch framework. The performance bottleneck of the package comes from two sources: 1) advanced indexing and 2) multiplications. Multiplications are handled in a way that scales well, but advanced indexing is not due to limitations with PyTorch. As a result, growth in problem size that is independent of the indexing bottleneck is handled very well by the package, such as:

1. Scaling the batch dimension.
2. Scaling the coil dimension.

Generally, you can just add to these dimensions and the package will perform well without adding much compute time. If you're chasing more speed, some strategies that might be helpful are listed below.

4.2.1 Using Batched K-space Trajectories

As of version `1.1.0`, `torchkbnufft` can use batched k-space trajectories. If you pass in a variable for `omega` with dimensions `(N, length(im_size), klength)`, the package will parallelize the execution of all trajectories in the `N` dimension. This is useful when `N` is very large, as might occur in dynamic imaging settings. The following shows an example:

```
import torch
import torchkbnufft as tkbn
import numpy as np
from skimage.data import shepp_logan_phantom

batch_size = 12

x = shepp_logan_phantom().astype(np.complex)
im_size = x.shape
# convert to tensor, unsqueeze batch and coil dimension
# output size: (batch_size, 1, ny, nx)
x = torch.tensor(x).unsqueeze(0).unsqueeze(0).to(torch.complex64)
x = x.repeat(batch_size, 1, 1, 1)

klength = 64
ktraj = np.stack(
    (np.zeros(64), np.linspace(-np.pi, np.pi, klength))
)
# convert to tensor, unsqueeze batch dimension
# output size: (batch_size, 2, klength)
ktraj = torch.tensor(ktraj).to(torch.float)
ktraj = ktraj.unsqueeze(0).repeat(batch_size, 1, 1)

nufft_ob = tkbn.KbNufft(im_size=im_size)
# outputs a (batch_size, 1, klength) vector of k-space data
kdata = nufft_ob(x, ktraj)
```

This code will then compute the 12 different radial spokes while parallelizing as much as possible.

4.2.2 Lowering the Precision

A simple way to save both memory and compute time is to decrease the precision. PyTorch normally operates at a default 32-bit floating point precision, but if you're converting data from NumPy then you might have some data at 64-bit floating precision. To use 32-bit precision, simply do the following:

```
image = image.to(dtype=torch.complex64)
ktraj = ktraj.to(dtype=torch.float32)
forw_ob = forw_ob.to(image)

data = forw_ob(image, ktraj)
```

The `forw_ob.to(image)` command will automagically determine the type for both real and complex tensors registered as buffers under `forw_ob`, so you should be able to do this safely in your code.

In many cases, the tradeoff for going from 64-bit to 32-bit is not severe, so you can securely use 32-bit precision.

4.2.3 Lowering the Oversampling Ratio

If you create a *KbNufft* object using the following code:

```
forw_ob = tkbn.KbNufft(im_size=im_size)
```

then by default it will use a 2-factor oversampled grid. For some applications, this can be overkill. If you can sacrifice some accuracy for your application, you can use a smaller grid with 1.25-factor oversampling by altering how you initialize NUFFT objects like *KbNufft*:

```
grid_size = tuple([int(el * 1.25) for el in im_size])
forw_ob = tkbn.KbNufft(im_size=im_size, grid_size=grid_size)
```

4.2.4 Using Fewer Interpolation Neighbors

Another major speed factor is how many neighbors you use for interpolation. By default, `torchkbnufft` uses 6 nearest neighbors in each dimension. If you can sacrifice accuracy, you can get more speed by using fewer neighbors by altering how you initialize NUFFT objects like *KbNufft*:

```
forw_ob = tkbn.KbNufft(im_size=im_size, numpoints=4)
```

If you know that you can be less accurate in one dimension (e.g., the z-dimension), then you can use less neighbors in only that dimension:

```
forw_ob = tkbn.KbNufft(im_size=im_size, numpoints=(4, 6, 6))
```

4.2.5 Package Limitations

As mentioned earlier, batches and coils scale well, primarily due to the fact that they don't impact the bottlenecks of the package around advanced indexing. Where `torchkbnufft` does not scale well is:

1. Very long k-space trajectories.
2. More imaging dimensions (e.g., 3D).

For these settings, you can first try to use some of the strategies here (lowering precision, fewer neighbors, smaller grid). In some cases, lowering the precision a bit and using a GPU can still give strong performance. If you're still waiting too long for compute after trying all of these, you may be running into the limits of the package.

4.3 torchkbnufft

4.3.1 NUFFT Modules

These are the primary workhorse modules for applying NUFFT operations.

KbInterp	Non-uniform Kaiser-Bessel interpolation layer.
KbInterpAdjoint	Non-uniform Kaiser-Bessel interpolation adjoint layer.
KbNufft	Non-uniform FFT layer.
KbNufftAdjoint	Non-uniform FFT adjoint layer.
ToepNufft	Forward/backward NUFFT with Toeplitz embedding.

KbInterp

class KbInterp(*im_size*, *grid_size=None*, *numpoints=6*, *n_shift=None*, *table_oversamp=1024*, *kbwidth=2.34*, *order=0.0*, *dtype=None*, *device=None*)

Non-uniform Kaiser-Bessel interpolation layer.

This object interpolates a grid of Fourier data to off-grid locations using a Kaiser-Bessel kernel. Mathematically, in one dimension it estimates $Y_m, m \in [0, ..., M-1]$ at frequency locations ω_m from $X_k, k \in [0, ..., K-1]$, the oversampled DFT of $x_n, n \in [0, ..., N-1]$. To perform the estimate, this layer applies

$$Y_m = \sum_{j=1}^{J} X_{\{k_m+j\}_K} u_j^*(\omega_m),$$

where u is the Kaiser-Bessel kernel, k_m is the index of the root offset of nearest samples of X to frequency location ω_m, and J is the number of nearest neighbors to use from X_k. Multiple dimensions are handled separably. For a detailed description of the notation see Nonuniform fast Fourier transforms using min-max interpolation (JA Fessler and BP Sutton).

When called, the parameters of this class define properties of the kernel and how the interpolation is applied.

- `im_size` is the size of the base image, analagous to N (used for calculating the kernel but not for the actual operation).
- `grid_size` is the size of the grid prior to interpolation, analogous to K. To reduce errors, NUFFT operations are done on an oversampled grid to reduce interpolation distances. This will typically be 1.25 to 2 times `im_size`.
- `numpoints` is the number of nearest to use for interpolation, i.e., J.
- `n_shift` is the FFT shift distance, typically `im_size // 2`.

Parameters

- **im_size** (Sequence[int]) – Size of image with length being the number of dimensions.
- **grid_size** (Optional[Sequence[int]]) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`. Default: `2 * im_size`
- **numpoints** (Union[int, Sequence[int]]) – Number of neighbors to use for interpolation in each dimension.
- **n_shift** (Optional[Sequence[int]]) – Size for `fftshift`. Default: `im_size // 2`.
- **table_oversamp** (Union[int, Sequence[int]]) – Table oversampling factor.
- **kbwidth** (float) – Size of Kaiser-Bessel kernel.
- **order** (Union[float, Sequence[float]]) – Order of Kaiser-Bessel kernel.
- **dtype** (Optional[dtype]) – Data type for tensor buffers. Default: `torch.get_default_dtype()`
- **device** (Optional[device]) – Which device to create tensors on. Default: `torch.device('cpu')`

Examples

```
>>> image = torch.randn(1, 1, 8, 8) + 1j * torch.randn(1, 1, 8, 8)
>>> omega = torch.rand(2, 12) * 2 * np.pi - np.pi
>>> kb_ob = tkbn.KbInterp(im_size=(8, 8), grid_size=(8, 8))
>>> data = kb_ob(image, omega)
```

forward(*image*, *omega*, *interp_mats=None*)

Interpolate from gridded data to scattered data.

Input tensors should be of shape `(N, C) + grid_size`, where `N` is the batch size and `C` is the number of sensitivity coils. `omega`, the k-space trajectory, should be of size `(len(grid_size), klength)` or `(N, len(grid_size), klength)`, where `klength` is the length of the k-space trajectory.

Note: If the batch dimension is included in `omega`, the interpolator will parallelize over the batch dimension. This is efficient for many small trajectories that might occur in dynamic imaging settings.

If your tensors are real-valued, ensure that 2 is the size of the last dimension.

Parameters

- **image** (Tensor) – Gridded data to be interpolated to scattered data.
- **omega** (Tensor) – k-space trajectory (in radians/voxel).
- **interp_mats** (Optional[Tuple[Tensor, Tensor]]) – 2-tuple of real, imaginary sparse matrices to use for sparse matrix KB interpolation (overrides default table interpolation).

Return type
Tensor

Returns
`image` calculated at Fourier frequencies specified by `omega`.

KbInterpAdjoint

class KbInterpAdjoint(*im_size*, *grid_size=None*, *numpoints=6*, *n_shift=None*, *table_oversamp=1024*, *kbwidth=2.34*, *order=0.0*, *dtype=None*, *device=None*)

Non-uniform Kaiser-Bessel interpolation adjoint layer.

This object interpolates off-grid Fourier data to on-grid locations using a Kaiser-Bessel kernel. Mathematically, in one dimension it estimates $X_k, k \in [0, ..., K-1]$, the oversampled DFT of $x_n, n \in [0, ..., N-1]$, from a signal $Y_m, m \in [0, ..., M-1]$ at frequency locations ω_m. To perform the estimate, this layer applies

$$X_k = \sum_{j=1}^{J} \sum_{m=0}^{M-1} Y_m u_j(\omega_m) \mathbb{1}_{\{\{k_m + j\}_K = k\}},$$

where u is the Kaiser-Bessel kernel, k_m is the index of the root offset of nearest samples of X to frequency location ω_m, $\mathbb{1}$ is an indicator function, and J is the number of nearest neighbors to use from X_k. Multiple dimensions are handled separably. For a detailed description of the notation see Nonuniform fast Fourier transforms using min-max interpolation (JA Fessler and BP Sutton).

Note: This function is not the inverse of *KbInterp*; it is the adjoint.

When called, the parameters of this class define properties of the kernel and how the interpolation is applied.

- `im_size` is the size of the base image, analagous to N (used for calculating the kernel but not for the actual operation).
- `grid_size` is the size of the grid after adjoint interpolation, analogous to K. To reduce errors, NUFFT operations are done on an oversampled grid to reduce interpolation distances. This will typically be 1.25 to 2 times `im_size`.
- `numpoints` is the number of nearest neighbors to use for interpolation, i.e., J.
- `n_shift` is the FFT shift distance, typically `im_size // 2`.

Parameters

- **im_size** (Sequence[int]) – Size of image with length being the number of dimensions.
- **grid_size** (Optional[Sequence[int]]) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`. Default: `2 * im_size`
- **numpoints** (Union[int, Sequence[int]]) – Number of neighbors to use for interpolation in each dimension.
- **n_shift** (Optional[Sequence[int]]) – Size for `fftshift`. Default: `im_size // 2`.
- **table_oversamp** (Union[int, Sequence[int]]) – Table oversampling factor.
- **kbwidth** (float) – Size of Kaiser-Bessel kernel.
- **order** (Union[float, Sequence[float]]) – Order of Kaiser-Bessel kernel.
- **dtype** (Optional[dtype]) – Data type for tensor buffers. Default: `torch.get_default_dtype()`
- **device** (Optional[device]) – Which device to create tensors on. Default: `torch.device('cpu')`

Examples

```
>>> data = torch.randn(1, 1, 12) + 1j * torch.randn(1, 1, 12)
>>> omega = torch.rand(2, 12) * 2 * np.pi - np.pi
>>> adjkb_ob = tkbn.KbInterpAdjoint(im_size=(8, 8), grid_size=(8, 8))
>>> image = adjkb_ob(data, omega)
```

forward(*data*, *omega*, *interp_mats=None*, *grid_size=None*)

Interpolate from scattered data to gridded data.

Input tensors should be of shape `(N, C) + klength`, where `N` is the batch size and `C` is the number of sensitivity coils. `omega`, the k-space trajectory, should be of size `(len(grid_size), klength)` or `(N, len(grid_size), klength)`, where `klength` is the length of the k-space trajectory.

Note: If the batch dimension is included in `omega`, the interpolator will parallelize over the batch dimension. This is efficient for many small trajectories that might occur in dynamic imaging settings.

If your tensors are real-valued, ensure that 2 is the size of the last dimension.

Parameters

- **data** (Tensor) – Data to be gridded.
- **omega** (Tensor) – k-space trajectory (in radians/voxel).
- **interp_mats** (Optional[Tuple[Tensor, Tensor]]) – 2-tuple of real, imaginary sparse matrices to use for sparse matrix KB interpolation (overrides default table interpolation).

Return type
Tensor

Returns
`data` interpolated to the grid.

KbNufft

class KbNufft(*im_size*, *grid_size=None*, *numpoints=6*, *n_shift=None*, *table_oversamp=1024*, *kbwidth=2.34*, *order=0.0*, *dtype=None*, *device=None*)

Non-uniform FFT layer.

This object applies the FFT and interpolates a grid of Fourier data to off-grid locations using a Kaiser-Bessel kernel. Mathematically, in one dimension it estimates $Y_m, m \in [0, ..., M-1]$ at frequency locations ω_m from $X_k, k \in [0, ..., K-1]$, the oversampled DFT of $x_n, n \in [0, ..., N-1]$. To perform the estimate, this layer applies

$$X_k = \sum_{n=0}^{N-1} s_n x_n e^{-i\gamma kn}$$

$$Y_m = \sum_{j=1}^{J} X_{\{k_m+j\}_K} u_j^*(\omega_m)$$

In the first step, an image-domain signal x_n is converted to a gridded, oversampled frequency-domain signal, X_k. The scaling coefficeints s_n are multiplied to precompensate for NUFFT interpolation errors. The oversampling coefficient is $\gamma = 2\pi/K, K >= N$.

In the second step, u, the Kaiser-Bessel kernel, is used to estimate X_k at off-grid frequency locations ω_m. k_m is the index of the root offset of nearest samples of X to frequency location ω_m, and J is the number of nearest

neighbors to use from X_k. Multiple dimensions are handled separably. For a detailed description see Nonuniform fast Fourier transforms using min-max interpolation (JA Fessler and BP Sutton).

When called, the parameters of this class define properties of the kernel and how the interpolation is applied.

- `im_size` is the size of the base image, analagous to N.
- `grid_size` is the size of the grid after forward FFT, analogous to K. To reduce errors, NUFFT operations are done on an oversampled grid to reduce interpolation distances. This will typically be 1.25 to 2 times `im_size`.
- `numpoints` is the number of nearest neighbors to use for interpolation, i.e., J.
- `n_shift` is the FFT shift distance, typically `im_size // 2`.

Parameters

- **im_size** (Sequence[int]) – Size of image with length being the number of dimensions.
- **grid_size** (Optional[Sequence[int]]) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`. Default: `2 * im_size`
- **numpoints** (Union[int, Sequence[int]]) – Number of neighbors to use for interpolation in each dimension.
- **n_shift** (Optional[Sequence[int]]) – Size for `fftshift`. Default: `im_size // 2`.
- **table_oversamp** (Union[int, Sequence[int]]) – Table oversampling factor.
- **kbwidth** (float) – Size of Kaiser-Bessel kernel.
- **order** (Union[float, Sequence[float]]) – Order of Kaiser-Bessel kernel.
- **dtype** (Optional[dtype]) – Data type for tensor buffers. Default: `torch.get_default_dtype()`
- **device** (Optional[device]) – Which device to create tensors on. Default: `torch.device('cpu')`

Examples

```
>>> image = torch.randn(1, 1, 8, 8) + 1j * torch.randn(1, 1, 8, 8)
>>> omega = torch.rand(2, 12) * 2 * np.pi - np.pi
>>> kb_ob = tkbn.KbNufft(im_size=(8, 8))
>>> data = kb_ob(image, omega)
```

forward(*image*, *omega*, *interp_mats=None*, *smaps=None*, *norm=None*)

Apply FFT and interpolate from gridded data to scattered data.

Input tensors should be of shape `(N, C) + im_size`, where `N` is the batch size and `C` is the number of sensitivity coils. `omega`, the k-space trajectory, should be of size `(len(grid_size), klength)` or `(N, len(grid_size), klength)`, where `klength` is the length of the k-space trajectory.

Note: If the batch dimension is included in `omega`, the interpolator will parallelize over the batch dimension. This is efficient for many small trajectories that might occur in dynamic imaging settings.

If your tensors are real, ensure that 2 is the size of the last dimension.

Parameters

- **image** (Tensor) – Object to calculate off-grid Fourier samples from.
- **omega** (Tensor) – k-space trajectory (in radians/voxel).
- **interp_mats** (Optional[Tuple[Tensor, Tensor]]) – 2-tuple of real, imaginary sparse matrices to use for sparse matrix NUFFT interpolation (overrides default table interpolation).
- **smaps** (Optional[Tensor]) – Sensitivity maps. If input, these will be multiplied before the forward NUFFT.
- **norm** (Optional[str]) – Whether to apply normalization with the FFT operation. Options are "ortho" or None.

Return type
Tensor

Returns
image calculated at Fourier frequencies specified by omega.

KbNufftAdjoint

class KbNufftAdjoint(*im_size*, *grid_size=None*, *numpoints=6*, *n_shift=None*, *table_oversamp=1024*, *kbwidth=2.34*, *order=0.0*, *dtype=None*, *device=None*)

Non-uniform FFT adjoint layer.

This object interpolates off-grid Fourier data to on-grid locations using a Kaiser-Bessel kernel prior to inverse DFT. Mathematically, in one dimension it estimates $x_n, n \in [0, ..., N-1]$ from a off-grid signal $Y_m, m \in [0, ..., M-1]$ where the off-grid frequency locations are ω_m. To perform the estimate, this layer applies

$$X_k = \sum_{j=1}^{J} \sum_{m=0}^{M-1} Y_m u_j(\omega_m) \mathbb{1}_{\{\{k_m + j\}_K = k\}},$$

$$x_n = s_n^* \sum_{k=0}^{K-1} X_k e^{i\gamma k n}$$

In the first step, u, the Kaiser-Bessel kernel, is used to estimate Y at on-grid frequency locations from locations at ω. k_m is the index of the root offset of nearest samples of X to frequency location ω_m, $\mathbb{1}$ is an indicator function, and J is the number of nearest neighbors to use from $X_k, k \in [0, ..., K-1]$.

In the second step, an image-domain signal x_n is estimated from a gridded, oversampled frequency-domain signal, X_k by applying the inverse FFT, after which the complex conjugate scaling coefficients s_n are multiplied. The oversampling coefficient is $\gamma = 2\pi/K, K >= N$. Multiple dimensions are handled separably. For a detailed description see Nonuniform fast Fourier transforms using min-max interpolation (JA Fessler and BP Sutton).

Note: This function is not the inverse of *KbNufft*; it is the adjoint.

When called, the parameters of this class define properties of the kernel and how the interpolation is applied.

- im_size is the size of the base image, analagous to N.
- grid_size is the size of the grid after adjoint interpolation, analogous to K. To reduce errors, NUFFT operations are done on an oversampled grid to reduce interpolation distances. This will typically be 1.25 to 2 times im_size.
- numpoints is the number of nearest neighbors to use for interpolation, i.e., J.

- `n_shift` is the FFT shift distance, typically `im_size // 2`.

Parameters

- **im_size** (Sequence[int]) – Size of image with length being the number of dimensions.
- **grid_size** (Optional[Sequence[int]]) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`. Default: `2 * im_size`
- **numpoints** (Union[int, Sequence[int]]) – Number of neighbors to use for interpolation in each dimension.
- **n_shift** (Optional[Sequence[int]]) – Size for `fftshift`. Default: `im_size // 2`.
- **table_oversamp** (Union[int, Sequence[int]]) – Table oversampling factor.
- **kbwidth** (float) – Size of Kaiser-Bessel kernel.
- **order** (Union[float, Sequence[float]]) – Order of Kaiser-Bessel kernel.
- **dtype** (Optional[dtype]) – Data type for tensor buffers. Default: `torch.get_default_dtype()`
- **device** (Optional[device]) – Which device to create tensors on. Default: `torch.device('cpu')`

Examples

```
>>> data = torch.randn(1, 1, 12) + 1j * torch.randn(1, 1, 12)
>>> omega = torch.rand(2, 12) * 2 * np.pi - np.pi
>>> adjkb_ob = tkbn.KbNufftAdjoint(im_size=(8, 8))
>>> image = adjkb_ob(data, omega)
```

forward(*data*, *omega*, *interp_mats=None*, *smaps=None*, *norm=None*)

Interpolate from scattered data to gridded data and then iFFT.

Input tensors should be of shape `(N, C) + klength`, where `N` is the batch size and `C` is the number of sensitivity coils. `omega`, the k-space trajectory, should be of size `(len(grid_size), klength)` or `(N, len(grid_size), klength)`, where `klength` is the length of the k-space trajectory.

Note: If the batch dimension is included in `omega`, the interpolator will parallelize over the batch dimension. This is efficient for many small trajectories that might occur in dynamic imaging settings.

If your tensors are real, ensure that 2 is the size of the last dimension.

Parameters

- **data** (Tensor) – Data to be gridded and then inverse FFT'd.
- **omega** (Tensor) – k-space trajectory (in radians/voxel).
- **interp_mats** (Optional[Tuple[Tensor, Tensor]]) – 2-tuple of real, imaginary sparse matrices to use for sparse matrix NUFFT interpolation (overrides default table interpolation).
- **smaps** (Optional[Tensor]) – Sensitivity maps. If input, these will be multiplied before the forward NUFFT.
- **norm** (Optional[str]) – Whether to apply normalization with the FFT operation. Options are `"ortho"` or `None`.

Return type
Tensor

Returns
data transformed to the image domain.

ToepNufft

class ToepNufft

Forward/backward NUFFT with Toeplitz embedding.

This module applies Tx, where T is a matrix such that $T \approx A'A$, where A is a NUFFT matrix. Using Toeplitz embedding, this module approximates the $A'A$ operation without interpolations, which is extremely fast.

The module is intended to be used in combination with an FFT kernel computed as frequency response of an embedded Toeplitz matrix. You can use *calc_toeplitz_kernel()* to calculate the kernel.

The FFT kernel should be passed to this module's forward operation, which applies a (zero-padded) FFT filter using the kernel.

Examples

```
>>> image = torch.randn(1, 1, 8, 8) + 1j * torch.randn(1, 1, 8, 8)
>>> omega = torch.rand(2, 12) * 2 * np.pi - np.pi
>>> toep_ob = tkbn.ToepNufft()
>>> kernel = tkbn.calc_toeplitz_kernel(omega, im_size=(8, 8))
>>> image = toep_ob(image, kernel)
```

forward(*image*, *kernel*, *smaps=None*, *norm=None*)

Toeplitz NUFFT forward function.

Parameters

- **image** (Tensor) – The image to apply the forward/backward Toeplitz-embedded NUFFT to.
- **kernel** (Tensor) – The filter response taking into account Toeplitz embedding.
- **norm** (Optional[str]) – Whether to apply normalization with the FFT operation. Options are "ortho" or None.

Return type
Tensor

Returns
image after applying the Toeplitz forward/backward NUFFT.

4.3.2 Utility Functions

Functions for calculating density compensation and Toeplitz kernels.

calc_density_compensation_function	Numerical density compensation estimation.
calc_tensor_spmatrix	Builds a sparse matrix for interpolation.
calc_toeplitz_kernel	Calculates an FFT kernel for Toeplitz embedding.

calc_density_compensation_function

calc_density_compensation_function(*ktraj*, *im_size*, *num_iterations=10*, *grid_size=None*, *numpoints=6*, *n_shift=None*, *table_oversamp=1024*, *kbwidth=2.34*, *order=0.0*)

Numerical density compensation estimation.

This function has optional parameters for initializing a NUFFT object. See *KbInterp* for details.

- `ktraj` should be of size `(len(grid_size), klength)` or `(N, len(grid_size), klength)`, where `klength` is the length of the k-space trajectory.

Based on the method of Pipe.

This code was contributed by Chaithya G.R.

Parameters

- **ktraj** (Tensor) – k-space trajectory (in radians/voxel).
- **im_size** (Sequence[int]) – Size of image with length being the number of dimensions.
- **num_iterations** (int) – Number of iterations.
- **grid_size** (Optional[Sequence[int]]) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`. Default: `2 * im_size`
- **numpoints** (Union[int, Sequence[int]]) – Number of neighbors to use for interpolation in each dimension. Default: `6`
- **n_shift** (Optional[Sequence[int]]) – Size for fftshift. Default: `im_size // 2`.
- **table_oversamp** (Union[int, Sequence[int]]) – Table oversampling factor.
- **kbwidth** (float) – Size of Kaiser-Bessel kernel.
- **order** (Union[float, Sequence[float]]) – Order of Kaiser-Bessel kernel.

Return type
Tensor

Returns
The density compensation coefficients for `ktraj`.

Examples

```
>>> data = torch.randn(1, 1, 12) + 1j * torch.randn(1, 1, 12)
>>> omega = torch.rand(2, 12) * 2 * np.pi - np.pi
>>> dcomp = tkbn.calc_density_compensation_function(omega, (8, 8))
>>> adjkb_ob = tkbn.KbNufftAdjoint(im_size=(8, 8))
>>> image = adjkb_ob(data * dcomp, omega)
```

calc_tensor_spmatrix

calc_tensor_spmatrix(*omega*, *im_size*, *grid_size=None*, *numpoints=6*, *n_shift=None*, *table_oversamp=1024*, *kbwidth=2.34*, *order=0.0*)

Builds a sparse matrix for interpolation.

This builds the interpolation matrices directly from scipy Kaiser-Bessel functions, so using them for a NUFFT should be a little more accurate than table interpolation.

This function has optional parameters for initializing a NUFFT object. See *KbNufft* for details.

- `omega` should be of size `(len(im_size), klength)`, where `klength` is the length of the k-space trajectory.

Parameters

- **omega** (`Tensor`) – k-space trajectory (in radians/voxel).
- **im_size** (`Sequence[int]`) – Size of image with length being the number of dimensions.
- **grid_size** (`Optional[Sequence[int]]`) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`. Default: `2 * im_size`
- **numpoints** (`Union[int, Sequence[int]]`) – Number of neighbors to use for interpolation in each dimension.
- **n_shift** (`Optional[Sequence[int]]`) – Size for fftshift. Default: `im_size // 2`.
- **table_oversamp** (`Union[int, Sequence[int]]`) – Table oversampling factor.
- **kbwidth** (`float`) – Size of Kaiser-Bessel kernel.
- **order** (`Union[float, Sequence[float]]`) – Order of Kaiser-Bessel kernel.
- **dtype** – Data type for tensor buffers. Default: `torch.get_default_dtype()`
- **device** – Which device to create tensors on. Default: `torch.device('cpu')`

Return type
`Tuple[Tensor, Tensor]`

Returns
2-Tuple of (real, imaginary) tensors for NUFFT interpolation.

Examples

```
>>> data = torch.randn(1, 1, 12) + 1j * torch.randn(1, 1, 12)
>>> omega = torch.rand(2, 12) * 2 * np.pi - np.pi
>>> spmats = tkbn.calc_tensor_spmatrix(omega, (8, 8))
>>> adjkb_ob = tkbn.KbNufftAdjoint(im_size=(8, 8))
>>> image = adjkb_ob(data, omega, spmats)
```

calc_toeplitz_kernel

calc_toeplitz_kernel(*omega*, *im_size*, *weights=None*, *norm=None*, *grid_size=None*, *numpoints=6*, *table_oversamp=1024*, *kbwidth=2.34*, *order=0.0*)

Calculates an FFT kernel for Toeplitz embedding.

The kernel is calculated using a adjoint NUFFT object. If the adjoint applies A', then this script calculates D where $F'DF \approx A'WA$, where F is a DFT matrix and W is a set of non-Cartesian k-space weights. D can then be used to approximate $A'WA$ without any interpolation operations.

For details on Toeplitz embedding, see Efficient numerical methods in non-uniform sampling theory (Feichtinger et al.).

This function has optional parameters for initializing a NUFFT object. See *KbNufftAdjoint* for details.

Note: This function is intended to be used in conjunction with *ToepNufft* for forward operations.

- `omega` should be of size `(len(im_size), klength)` or `(N, len(im_size), klength)`, where `klength` is the length of the k-space trajectory.

Parameters

- **omega** (`Tensor`) – k-space trajectory (in radians/voxel).
- **im_size** (`Sequence[int]`) – Size of image with length being the number of dimensions.
- **weights** (`Optional[Tensor]`) – Non-Cartesian k-space weights (e.g., density compensation). Default: `torch.ones(omega.shape[1])`
- **norm** (`Optional[str]`) – Whether to apply normalization with the FFT operation. Options are `"ortho"` or `None`.
- **grid_size** (`Optional[Sequence[int]]`) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`. Default: `2 * im_size`
- **numpoints** (`Union[int, Sequence[int]]`) – Number of neighbors to use for interpolation in each dimension.
- **n_shift** – Size for fftshift. Default: `im_size // 2`.
- **table_oversamp** (`Union[int, Sequence[int]]`) – Table oversampling factor.
- **kbwidth** (`float`) – Size of Kaiser-Bessel kernel.
- **order** (`Union[float, Sequence[float]]`) – Order of Kaiser-Bessel kernel.

Return type
`Tensor`

Returns
The FFT kernel for approximating the forward/adjoint operation.

Examples

```
>>> image = torch.randn(1, 1, 8, 8) + 1j * torch.randn(1, 1, 8, 8)
>>> omega = torch.rand(2, 12) * 2 * np.pi - np.pi
>>> toep_ob = tkbn.ToepNufft()
>>> kernel = tkbn.calc_toeplitz_kernel(omega, im_size=(8, 8))
>>> image = toep_ob(image, kernel)
```

4.3.3 Math Functions

Complex mathematical operations (gradually being removed as of PyTorch 1.7).

absolute	Complex absolute value.
complex_mult	Complex multiplication.
complex_sign	Complex sign function value.
conj_complex_mult	Complex multiplication, conjugating second input.
imag_exp	Imaginary exponential.
inner_product	Complex inner product.

torchkbnufft.absolute

absolute(*val*, *dim=-1*)

Complex absolute value.

Parameters

- **val** (Tensor) – A tensor to have its absolute value computed.
- **dim** (int) – An integer indicating the complex dimension (for real inputs only).

Return type
Tensor

Returns
The absolute value of `val`.

torchkbnufft.complex_mult

complex_mult(*val1*, *val2*, *dim=-1*)

Complex multiplication.

Parameters

- **val1** (Tensor) – A tensor to be multiplied.
- **val2** (Tensor) – A second tensor to be multiplied.
- **dim** (int) – An integer indicating the complex dimension (for real inputs only).

Return type
Tensor

Returns
`val1 * val2`, where * executes complex multiplication.

torchkbnufft.complex_sign

complex_sign(*val*, *dim=-1*)

Complex sign function value.

Parameters

- **val** (Tensor) – A tensor to have its complex sign computed.
- **dim** (int) – An integer indicating the complex dimension (for real inputs only).

Return type
Tensor

Returns
The complex sign of `val`.

torchkbnufft.conj_complex_mult

conj_complex_mult(*val1*, *val2*, *dim=-1*)

Complex multiplication, conjugating second input.

Parameters

- **val1** (Tensor) – A tensor to be multiplied.
- **val2** (Tensor) – A second tensor to be conjugated then multiplied.
- **dim** (int) – An integer indicating the complex dimension (for real inputs only).

Return type
Tensor

Returns
`val3 = val1 * conj(val2)`, where * executes complex multiplication.

torchkbnufft.imag_exp

imag_exp(*val*, *dim=-1*, *return_complex=True*)

Imaginary exponential.

Parameters

- **val** (Tensor) – A tensor to be exponentiated.
- **dim** (int) – An integer indicating the complex dimension of the output (for real outputs only).

Return type
Tensor

Returns
`val2 = exp(i*val)`, where `i` is `sqrt(-1)`.

torchkbnufft.inner_product

inner_product(*val1*, *val2*, *dim=-1*)

Complex inner product.

Parameters

- **val1** (Tensor) – A tensor for the inner product.
- **val2** (Tensor) – A second tensor for the inner product.
- **dim** (int) – An integer indicating the complex dimension (for real inputs only).

Return type
Tensor

Returns
The complex inner product of `val1` and `val2`.

4.4 torchkbnufft.functional

4.4.1 fft_filter

fft_filter(*image*, *kernel*, *norm='ortho'*)

FFT-based filtering on an oversampled grid.

This is a wrapper for the operation

$$\text{output} = iFFT(\text{kernel} * FFT(\text{image}))$$

where $iFFT$ and FFT are both implemented as oversampled FFTs.

Parameters

- **image** (Tensor) – The image to be filtered.
- **kernel** (Tensor) – FFT-domain filter.
- **norm** (Optional[str]) – Whether to apply normalization with the FFT operation. Options are `"ortho"` or `None`.

Return type
Tensor

Returns
Filtered version of `image`.

Interpolation Functions

4.4.2 kb_spmat_interp

kb_spmat_interp(*image*, *interp_mats*)

Kaiser-Bessel sparse matrix interpolation.

See *KbInterp* for an overall description of interpolation.

To calculate the sparse matrix tuple, see *calc_tensor_spmatrix()*.

Parameters

- **image** (Tensor) – Gridded data to be interpolated to scattered data.
- **interp_mats** (Tuple[Tensor, Tensor]) – 2-tuple of real, imaginary sparse matrices to use for sparse matrix KB interpolation.

Return type
Tensor

Returns
image calculated at scattered locations.

4.4.3 kb_spmat_interp_adjoint

kb_spmat_interp_adjoint(*data*, *interp_mats*, *grid_size*)

Kaiser-Bessel sparse matrix interpolation adjoint.

See *KbInterpAdjoint* for an overall description of adjoint interpolation.

To calculate the sparse matrix tuple, see *calc_tensor_spmatrix()*.

Parameters

- **data** (Tensor) – Scattered data to be interpolated to gridded data.
- **interp_mats** (Tuple[Tensor, Tensor]) – 2-tuple of real, imaginary sparse matrices to use for sparse matrix KB interpolation.

Return type
Tensor

Returns
data calculated at gridded locations.

4.4.4 kb_table_interp

kb_table_interp(*image*, *omega*, *tables*, *n_shift*, *numpoints*, *table_oversamp*, *offsets*)

Kaiser-Bessel table interpolation.

See *KbInterp* for an overall description of interpolation and how to construct the function arguments.

Parameters

- **image** (Tensor) – Gridded data to be interpolated to scattered data.
- **omega** (Tensor) – k-space trajectory (in radians/voxel).
- **tables** (List[Tensor]) – Interpolation tables (one table for each dimension).
- **n_shift** (Tensor) – Size for fftshift, usually im_size // 2.
- **numpoints** (Tensor) – Number of neighbors to use for interpolation.
- **table_oversamp** (Tensor) – Table oversampling factor.
- **offsets** (Tensor) – A list of offsets, looping over all possible combinations of numpoints.

Return type
Tensor

Returns
image calculated at scattered locations.

4.4.5 kb_table_interp_adjoint

kb_table_interp_adjoint(*data*, *omega*, *tables*, *n_shift*, *numpoints*, *table_oversamp*, *offsets*, *grid_size*)

Kaiser-Bessel table interpolation adjoint.

See *KbInterpAdjoint* for an overall description of adjoint interpolation.

Parameters

- **data** (Tensor) – Scattered data to be interpolated to gridded data.
- **omega** (Tensor) – k-space trajectory (in radians/voxel).
- **tables** (List[Tensor]) – Interpolation tables (one table for each dimension).
- **n_shift** (Tensor) – Size for fftshift, usually `im_size // 2`.
- **numpoints** (Tensor) – Number of neighbors to use for interpolation.
- **table_oversamp** (Tensor) – Table oversampling factor.
- **offsets** (Tensor) – A list of offsets, looping over all possible combinations of `numpoints`.
- **grid_size** (Tensor) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`.

Return type
Tensor

Returns
`data` calculated at gridded locations.

NUFFT Functions

4.4.6 kb_spmat_nufft

kb_spmat_nufft(*image*, *scaling_coef*, *im_size*, *grid_size*, *interp_mats*, *norm=None*)

Kaiser-Bessel NUFFT with sparse matrix interpolation.

See *KbNufft* for an overall description of the forward NUFFT.

To calculate the sparse matrix tuple, see *calc_tensor_spmatrix()*.

Parameters

- **image** (Tensor) – Image to be NUFFT'd to scattered data.
- **scaling_coef** (Tensor) – Image-domain coefficients to pre-compensate for interpolation errors.
- **im_size** (Tensor) – Size of image with length being the number of dimensions.
- **grid_size** (Tensor) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`.
- **interp_mats** (Tuple[Tensor, Tensor]) – 2-tuple of real, imaginary sparse matrices to use for sparse matrix KB interpolation.
- **norm** (Optional[str]) – Whether to apply normalization with the FFT operation. Options are `"ortho"` or `None`.

Return type
Tensor

Returns
`image` calculated at scattered Fourier locations.

4.4.7 kb_spmat_nufft_adjoint

kb_spmat_nufft_adjoint(*data*, *scaling_coef*, *im_size*, *grid_size*, *interp_mats*, *norm=None*)

Kaiser-Bessel adjoint NUFFT with sparse matrix interpolation.

See *KbNufftAdjoint* for an overall description of the adjoint NUFFT.

To calculate the sparse matrix tuple, see *calc_tensor_spmatrix()*.

Parameters

- **data** (Tensor) – Scattered data to be iNUFFT'd to an image.
- **scaling_coef** (Tensor) – Image-domain coefficients to compensate for interpolation errors.
- **im_size** (Tensor) – Size of image with length being the number of dimensions.
- **grid_size** (Tensor) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`.
- **interp_mats** (Tuple[Tensor, Tensor]) – 2-tuple of real, imaginary sparse matrices to use for sparse matrix KB interpolation.
- **norm** (Optional[str]) – Whether to apply normalization with the FFT operation. Options are `"ortho"` or `None`.

Return type
Tensor

Returns
`data` transformed to an image.

4.4.8 kb_table_nufft

kb_table_nufft(*image*, *scaling_coef*, *im_size*, *grid_size*, *omega*, *tables*, *n_shift*, *numpoints*, *table_oversamp*, *offsets*, *norm=None*)

Kaiser-Bessel NUFFT with table interpolation.

See *KbNufft* for an overall description of the forward NUFFT.

Parameters

- **image** (Tensor) – Image to be NUFFT'd to scattered data.
- **scaling_coef** (Tensor) – Image-domain coefficients to pre-compensate for interpolation errors.
- **im_size** (Tensor) – Size of image with length being the number of dimensions.
- **grid_size** (Tensor) – Size of grid to use for interpolation, typically 1.25 to 2 times `im_size`.
- **omega** (Tensor) – k-space trajectory (in radians/voxel).
- **tables** (List[Tensor]) – Interpolation tables (one table for each dimension).
- **n_shift** (Tensor) – Size for fftshift, usually `im_size // 2`.

- **numpoints** (Tensor) – Number of neighbors to use for interpolation.
- **table_oversamp** (Tensor) – Table oversampling factor.
- **offsets** (Tensor) – A list of offsets, looping over all possible combinations of *numpoints*.
- **norm** (Optional[str]) – Whether to apply normalization with the FFT operation. Options are "ortho" or None.

Return type
Tensor

Returns
image calculated at scattered Fourier locations.

4.4.9 kb_table_nufft_adjoint

kb_table_nufft_adjoint(*data*, *scaling_coef*, *im_size*, *grid_size*, *omega*, *tables*, *n_shift*, *numpoints*, *table_oversamp*, *offsets*, *norm=None*)

Kaiser-Bessel NUFFT adjoint with table interpolation.

See *KbNufftAdjoint* for an overall description of the adjoint NUFFT.

Parameters

- **data** (Tensor) – Scattered data to be iNUFFT'd to an image.
- **scaling_coef** (Tensor) – Image-domain coefficients to compensate for interpolation errors.
- **im_size** (Tensor) – Size of image with length being the number of dimensions.
- **grid_size** (Tensor) – Size of grid to use for interpolation, typically 1.25 to 2 times im_size.
- **omega** (Tensor) – k-space trajectory (in radians/voxel).
- **tables** (List[Tensor]) – Interpolation tables (one table for each dimension).
- **n_shift** (Tensor) – Size for fftshift, usually im_size // 2.
- **numpoints** (Tensor) – Number of neighbors to use for interpolation.
- **table_oversamp** (Tensor) – Table oversampling factor.
- **offsets** (Tensor) – A list of offsets, looping over all possible combinations of *numpoints*.
- **norm** (Optional[str]) – Whether to apply normalization with the FFT operation. Options are "ortho" or None.

Return type
Tensor

Returns
data transformed to an image.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search

INDEX