## Contents

1. Installation  
2. Try Now  
3. Contribute  
4. License  
5. Indices and tables

Bibliography  
Python Module Index  
Index
TomoPy is an open-source Python package for tomographic data processing and image reconstruction.

- Image reconstruction algorithms for tomography.
- Various filters, ring removal algorithms, phase retrieval algorithms.
- Forward projection operator for absorption and wave propagation.
CHAPTER 1

Installation

Have Conda installed first, then open a terminal or a command prompt window and run:

**Warning:** TomoPy will drop support for Python 2 before 1 January 2020. For more information, visit [https://python3statement.org/](https://python3statement.org/).
Try Now

You can try notebooks now using mybinder.org

Contribute

- **Issue Tracker**: [https://github.com/tomopy/tomopy/issues](https://github.com/tomopy/tomopy/issues)
- **Documentation**: [https://github.com/tomopy/tomopy/tree/master/doc](https://github.com/tomopy/tomopy/tree/master/doc)
- **Source Code**: [https://github.com/tomopy/tomopy/tree/master/tomopy](https://github.com/tomopy/tomopy/tree/master/tomopy)
- **Tests**: [https://github.com/tomopy/tomopy/tree/master/test](https://github.com/tomopy/tomopy/tree/master/test)
The project is licensed under the BSD-3 license.
5.1 About

Tomographic reconstruction creates three-dimensional views of an object by combining two-dimensional images taken from multiple directions, for example, this is how a CAT (computer-aided tomography) scanner generates 3D views of the heart or brain.

Data collection can be rapid, but the required computations are massive and often the beamline staff can be overwhelmed by data that are collected far faster than corrections and reconstruction can be performed [C15]. Further, many common experimental perturbations can degrade the quality of tomographs, unless corrections are applied.

To address the needs for image correction and tomographic reconstruction in an instrument independent manner, the TomoPy code was developed [A1], which is a parallelizable high performance reconstruction code.

5.2 Install directions

This section covers the basics of how to download and install TomoPy.

5.2.1 Supported Environments

TomoPy is tested, built, and distributed for python 2.7 3.6 3.7 on Linux/macOS and python 3.6 3.7 on Windows 10.
5.2.2 OPTION 1: Installing from Conda (Recommended)

If you only want to run TomoPy, not develop it, then you should install through a package manager. Conda, our supported package manager, can install TomoPy and its dependencies for you.

First, you must have Conda installed, then open a terminal or a command prompt window and run:

```
$ conda install -c conda-forge tomopy
```

This will install TomoPy and all the dependencies from the conda-forge channel.

Updating the installation

TomoPy is an active project, so we suggest you update your installation frequently. To update the installation run:

```
$ conda update -c conda-forge tomopy
```

For some more information about using Conda, please refer to the docs.

5.2.3 OPTION 2: Installing from source

Please read the devguide for instructions on how to install TomoPy from source code.

5.3 How to read tomographic data

For reading tomography files formatted in different ways, please go check the DXchange package. There are various examples and demonstration scripts about how to use the package for loading your datasets.

The package can be installed by simply running the following in a terminal:

```
conda install -c conda-forge dxchange
```

For a repository of experimental and simulated data sets please check TomoBank [C6].

5.4 Development

This section explains the basics for developers who wish to contribute to the TomoPy project.

Contents:

- Cloning the repository
- Installing dependencies
- Building TomoPy
- Running the Tests
- Coding conventions
- Package versioning
- Committing changes
5.4.1 Cloning the repository

The project is maintained on GitHub, which is a version control and a collaboration platform for software developers. To start, first register on GitHub and fork the TomoPy repository by clicking the Fork button in the header of the TomoPy repository:
This creates a copy of the project in your personal GitHub space. The next thing you want to do is to clone it to your local machine. You can do this by clicking the **Clone in Desktop** button in the bottom of the right hand side bar:
Chapter 5. Indices and tables
This will launch the GitHub desktop application (available for both Mac and Win) and ask you where you want to save it. Select a location in your computer and feel comfortable with making modifications in the code.

5.4.2 Installing dependencies

To build and run TomoPy, you will need to install at least the dependencies listed in envs/{platform}-{version}.yml plus additional dependencies based on your build environment. For example, installing requirements for building the Python 3.6 version on Unix can be accomplished as follows:

```bash
$ conda env create -f envs/linux-36.yml
```

5.4.3 Building TomoPy

After navigating to inside the tomopy directory, you can install TomoPy by running the install script in the typical Python way:

```bash
$ python setup.py install
```

5.4.4 Running the Tests

TomoPy has a suite of Python unit tests that live in the /test directory, where they follow the same tree structure as the packages under /tomopy. These are automatically run by TravisCI when you make a pull request (See below for how to do that). You can run them manually using pytest, or whichever python test runner you prefer. To make it easier to run tests on the changes you make to the code, it is recommended that you install TomoPy in development mode. (python setup.py develop)

The `pytest` test runner, is available through pip or anaconda.

To run the tests open a terminal, navigate to your project folder, then run `py.test`.

To run sections of tests, pass py.test a directory or filepath, as in `py.test test/test_recon` or `py.test test/test_recon/test_rotation.py`.

When writing tests, at minimum we try to check all function returns with synthetic data, together with some dimension, type, etc. Writing tests is highly encouraged!

5.4.5 Coding conventions

We try to keep our code consistent and readable. So, please keep in mind the following style and syntax guidance before you start coding.

First of all the code should be well documented, easy to understand, and integrate well into the rest of the project. For example, when you are writing a new function always describe the purpose and the parameters:

```python
def my_awesome_func(a, b):
    
    Adds two numbers.

    Parameters
    ----------
    a : scalar (float)
        First number to add
```

(continues on next page)
b : scalar (float)
    Second number to add

Returns
------
output : scalar (float)
    Added value

""
return a+b

5.4.6 Package versioning

We follow the X.Y.Z (Major.Minor.Patch) semantic for package versioning. The version should be updated before each pull request accordingly. The patch number is incremented for minor changes and bug fixes which do not change the software’s API. The minor version is incremented for releases which add new, but backward-compatible, API features, and the major version is incremented for API changes which are not backward-compatible. For example, software which relies on version 2.1.5 of an API is compatible with version 2.2.3, but not necessarily with 3.2.4.

5.4.7 Committing changes

After making some changes in the code, you may want to take a snapshot of the edits you made. That’s when you make a commit. To do this, launch the GitHub desktop application and it should provide you all the changes in your code since your last commit. Write a brief Summary and Description about the changes you made and click the Commit button:
5.4. Development
You can continue to make changes, add modules, write your own functions, and take more *Commit snapshots* of your code writing process.

### 5.4.8 Contributing back

Once you feel that the functionality you added would benefit the community, then you should consider contributing back to the TomoPy project. You will need to push your local commits to GitHub, then go to your online GitHub repository of the project and click on the *green* button to compare, review, and create a pull request.
After clicking on this button, you are presented with a review page where you can get a high-level overview of what exactly has changed between your forked branch and the original TomoPy repository. When you’re ready to submit your pull request, click **Create pull request**:
Clicking on Create pull request sends you to a discussion page, where you can enter a title and optional description. It’s important to provide as much useful information and a rationale for why you’re making this Pull Request in the first place.

When you’re ready typing out your heartfelt argument, click on Send pull request. You’re done!

5.5 GPU Support

This section covers building TomoPy with support for GPU offloading and general usage. For the implemented iterative algorithms, reconstruction times per-slice can be reduced by several orders of magnitude depending on the hardware available. On a NVIDIA Volta (VX-100) GPU at NERSC, the per-slice reconstruction time of the SIRT and MLEM algorithms for a 2048p image with 1501 projection angles and 100 iterations reduced from ~6.5 hours to ~40 seconds.

5.5.1 Supported GPUs

TomoPy supports offloading to NVIDIA GPUs through compiled CUDA kernels on Linux and Windows 10. NVIDIA GPUs on macOS are untested but likely supported.

5.5.2 Building TomoPy with CUDA

CMake is configured to automatically enable building GPU support when CMake can detect a valid CUDA compiler. TomoPy requires CMake 3.9+, which has support for CUDA as a first-class language – meaning that the CUDA compiler only needs to be in the PATH. On Unix, this is easily checked with the command: which nvcc. If the command returns a path to the compiler, build TomoPy normally. If not, locate the CUDA compiler and place the path to the compiler in PATH, remove the build directory (rm -r _skbuild or python setup.py clean) and rebuild.

TomoPy includes the Parallel Tasking Library (PTL) as a git submodule to handle the creation of a secondary thread-pool that assists in hiding the communication latency between the CPU and GPU. This submodule is automatically checked out and compiled by the CMake build system.

5.5.3 Reconstructing with GPU offloading

In order to reconstruct efficiently on the GPU, the algorithm has been implemented as a rotation-based reconstruction instead of the standard ray-based reconstruction. The primary implication of the algorithmic change is that when there are important pixels at the corners of the image, it will be necessary to pad the image before reconstruction. This is due to the side-effect of a rotating at an arbitrary angle that is not a factor of 90 degrees:

```python
obj = tomopy.shepp2d()
obj = tomopy.misc.morph.pad(obj, axis=1, mode='constant')
obj = tomopy.misc.morph.pad(obj, axis=2, mode='constant')
```

Currently, the supported algorithms for GPU offloading are:

<table>
<thead>
<tr>
<th>Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIRT</td>
</tr>
<tr>
<td>MLEM</td>
</tr>
</tbody>
</table>

When GPU support is not available, due to either lack of compiler support or no CUDA devices available, the algorithms will execute on the CPU with the same algorithm as the GPU version using OpenCV. When an NVIDIA device
is targeted, the algorithms utilize the NPP (NVIDIA Performance Primitives) library instead of OpenCV which has limited GPU support. However, it is possible that GPU offloading will still occur if OpenCV is configured with GPU support.

The addition of `accelerated=True` to `tomopy.recon(...)` is the only requirement for enabling the accelerated versions of the above algorithms. However, there is support for an additional customization:

<table>
<thead>
<tr>
<th><code>tomopy.recon</code> parameters</th>
<th>Type</th>
<th>Description</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>accelerated</td>
<td>boolean</td>
<td>Enable accelerated algorithm</td>
<td>True, False</td>
</tr>
<tr>
<td>pool_size</td>
<td>int</td>
<td>Size of the secondary thread-pool</td>
<td></td>
</tr>
<tr>
<td>interpolation</td>
<td>string</td>
<td>Interpolation scheme</td>
<td>“NN”, “LINEAR”, “CUBIC”</td>
</tr>
<tr>
<td>device</td>
<td>string</td>
<td>Targeted device</td>
<td>“cpu”, ”gpu”</td>
</tr>
<tr>
<td>grid_size</td>
<td>nparray</td>
<td>GPU grid dimensions</td>
<td>Set to [0,0,0] for auto grid size</td>
</tr>
<tr>
<td>block_size</td>
<td>nparray</td>
<td>GPU block dimensions</td>
<td>Default is [32,32,1]</td>
</tr>
</tbody>
</table>

5.5.4 Multithreading

TomoPy supports multithreading at the Python level through the `ncore` parameter. When offloading to the GPU, it is generally recommended to set `ncore` to the number of GPUs. As the threads started at the Python level drop down into the compiled code of TomoPy, these threads increment a counter that spreads their execution across all of the available GPUs

```cpp
// thread counter for device assignment
static std::atomic<int> ntid;

// increment counter and get a "Python thread-id"
int pythread_num = ntid++;

// set the device to the modulus of number of device available
int device = pythread_num % cuda_device_count();
```

As mentioned previously, TomoPy creates a secondary thread-pool in the accelerated algorithms that assists in hiding the communication latency between the CPU and GPU. Once a thread has been assigned a device, it creates `pool_size` additional threads for this purpose. When offloading to the GPU, the standard recommendation is to over-subscribe the number of threads relative to the number of hardware cores. The ideal number of threads per GPU is around 12-24 threads. The default number of `pool_size` threads is twice the number of hardware threads available divided by the number of threads started at the Python level, e.g. if there are 8 CPU cores and 1 thread started at the Python level, 16 threads will be created in the secondary thread-pool.

5.6 Release Notes

5.6.1 TomoPy 1.0.0 Release Notes

- **New features**
- **New functions**
- **New packages in Conda channel**
New features

- **FFTW** implementation is now adopted. All functions that rely on FFTs such as gridrec, phase retrieval, stripe removal, etc. are now using the FFTW implementation through PyFFTW.

- **sinogram_order** is added to **recon** as an additional argument. It determines whether data is a stack of sinograms (True, y-axis first axis) or a stack of radiographs (False). Default is False, but we plan to make it True in the upcoming release.

- Reconstruction algorithms only copies data if necessary.

- Updated library to support new mproc and recon functions. The data is now passed in sinogram order to recon functions. Also updated tests.

- **ncores** and **nchunks** are now independent.

- Setting **nchunks** to zero removes the dimension. That allows for the functions work on 2D data rather than 3D data.

- Sliced data are used so that each process only receives the data it needs. No more **istart** and **iend** variables for setting up indices in parallel processes.

- Functions will reuse sharedmem arrays if they can.

New functions

- **minus_log**

- **trim_sinogram**

New packages in Conda channel

- **dxchange 0.1.1**
- **fftw 3.3.4**
- **pyfftw 0.9.2**
- **pywavelets 0.4.0**
- **xraylib 3.1.0**

Deprecated features

- All data I/O related functions are deprecated. They are available through **DXchange** package.

- Removed fft.h and fft.c, they are now completely replaced with FFTW.
Backward incompatible changes

- **emission** argument is removed from recon. After this change the tomographic image reconstruction algorithms always assume data to be normalized.

Contributors

- Arthur Glowacki (@aglowacki)
- Daniel Pelt (@dmpelt)
- Dake Feng (@dakefeng)
- Doga Gursoy (@dgursoy)
- Francesco De Carlo (@decarlof)
- Lin Jiao (@yxqd)
- Luis Barroso-Luque (@lbluque)
- Michael Sutherland (@michael-sutherland)
- Rafael Vescovi (@ravescovi)
- Thomas Caswell (@tacaswell)
- Pete R. Jemian (@prjemian)
- Wei Xu (@celiafish)

5.7 API reference

This section contains the API reference and usage information for TomoPy.

TomoPy Modules:

5.7.1 **tomopy.misc.corr**

Module for data correction and masking functions.

Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>adjust_range(arr[, dmin, dmax])</code></td>
<td>Change dynamic range of values in an array.</td>
</tr>
<tr>
<td><code>circ_mask(arr, axis[, ratio, val, ncore])</code></td>
<td>Apply circular mask to a 3D array.</td>
</tr>
<tr>
<td><code>gaussian_filter(arr[, sigma, order, axis, ncore])</code></td>
<td>Apply Gaussian filter to 3D array along specified axis.</td>
</tr>
<tr>
<td><code>median_filter(arr[, size, axis, ncore])</code></td>
<td>Apply median filter to 3D array along specified axis.</td>
</tr>
<tr>
<td><code>median_filter_cuda(arr[, size, axis])</code></td>
<td>Apply median filter to 3D array along 0 axis with GPU support.</td>
</tr>
<tr>
<td><code>sobel_filter(arr[, axis, ncore])</code></td>
<td>Apply Sobel filter to 3D array along specified axis.</td>
</tr>
<tr>
<td><code>remove_nan(arr[, val, ncore])</code></td>
<td>Replace NaN values in array with a given value.</td>
</tr>
<tr>
<td><code>remove_neg(arr[, val, ncore])</code></td>
<td>Replace negative values in array with a given value.</td>
</tr>
</tbody>
</table>

Continued on next page
**Table 1 – continued from previous page**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>remove_outlier(arr, dif[, size, axis, ...])</code></td>
<td>Remove high intensity bright spots from a N-dimensional array by chunking along the specified dimension, and performing (N-1)-dimensional median filtering along the other dimensions.</td>
</tr>
<tr>
<td><code>remove_outlier_cuda(arr, dif[, size, axis])</code></td>
<td>Remove high intensity bright spots from a 3D array along axis 0 dimension using GPU.</td>
</tr>
<tr>
<td><code>remove_ring(rec[, center_x, center_y, ...])</code></td>
<td>Remove ring artifacts from images in the reconstructed domain.</td>
</tr>
</tbody>
</table>

**tomopy.misc.corr.adjust_range (arr, dmin=None, dmax=None)**

Change dynamic range of values in an array.

**Parameters**

- `arr (ndarray)` – Input array.
- `dmin, dmax (float, optional)` – Minimum and maximum values to rescale data.

**Returns** `ndarray` – Output array.

**tomopy.misc.corr.circ_mask (arr, axis, ratio=1, val=0.0, ncore=None)**

Apply circular mask to a 3D array.

**Parameters**

- `arr (ndarray)` – Arbitrary 3D array.
- `axis (int)` – Axis along which mask will be performed.
- `ratio (int, optional)` – Ratio of the mask’s diameter in pixels to the smallest edge size along given axis.
- `val (int, optional)` – Value for the masked region.

**Returns** `ndarray` – Masked array.

**tomopy.misc.corr.gaussian_filter (arr, sigma=3, order=0, axis=0, ncore=None)**

Apply Gaussian filter to 3D array along specified axis.

**Parameters**

- `arr (ndarray)` – Input array.
- `sigma (scalar or sequence of scalars)` – Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- `order (0, 1, 2, 3) or sequence from same set, optional)` – Order of the filter along each axis is given as a sequence of integers, or as a single number. An order of 0 corresponds to convolution with a Gaussian kernel. An order of 1, 2, or 3 corresponds to convolution with the first, second or third derivatives of a Gaussian. Higher order derivatives are not implemented
- `axis (int, optional)` – Axis along which median filtering is performed.
- `ncore (int, optional)` – Number of cores that will be assigned to jobs.

**Returns** `ndarray` – 3D array of same shape as input.

**tomopy.misc.corr.median_filter (arr, size=3, axis=0, ncore=None)**

Apply median filter to 3D array along specified axis.

**Parameters**
• **arr** (*ndarray*) – Input array.
• **size** (*int, optional*) – The size of the filter.
• **axis** (*int, optional*) – Axis along which median filtering is performed.
• **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.

**Returns** *ndarray* – Median filtered 3D array.

tomopy.misc.corr.*median_filter_cuda* (*arr*, *size*=3, *axis*=0)

Apply median filter to 3D array along 0 axis with GPU support. The winAllow is for A6000, Tian X support 3 to 8

**Parameters**

• **arr** (*ndarray*) – Input array.
• **size** (*int, optional*) – The size of the filter.
• **axis** (*int, optional*) – Axis along which median filtering is performed.

**Returns** *ndarray* – Median filtered 3D array.

**Example**

```python
import tomocuda
tomocuda.remove_outlier_cuda(arr, dif, 5)
```

For more information regarding install and using tomocuda, check [https://github.com/kyuepublic/tomocuda](https://github.com/kyuepublic/tomocuda) for more information

tomopy.misc.corr.*sobel_filter* (*arr*, *axis*=0, *ncore*=None)

Apply Sobel filter to 3D array along specified axis.

**Parameters**

• **arr** (*ndarray*) – Input array.
• **axis** (*int, optional*) – Axis along which sobel filtering is performed.
• **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.

**Returns** *ndarray* – 3D array of same shape as input.

tomopy.misc.corr.*remove_nan* (*arr*, *val*=0.0, *ncore*=None)

Replace NaN values in array with a given value.

**Parameters**

• **arr** (*ndarray*) – Input array.
• **val** (*float, optional*) – Values to be replaced with NaN values in array.
• **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.

**Returns** *ndarray* – Corrected array.

tomopy.misc.corr.*remove_neg* (*arr*, *val*=0.0, *ncore*=None)

Replace negative values in array with a given value.

**Parameters**

• **arr** (*ndarray*) – Input array.
• **val** (*float, optional*) – Values to be replaced with negative values in array.
• **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.
**Return**

`ndarray` – Corrected array.

**tomopy.misc.corr.remove_outlier**

```
tomopy.misc.corr.remove_outlier(arr, dif, size=3, axis=0, ncore=None, out=None)
```

Remove high intensity bright spots from a N-dimensional array by chunking along the specified dimension, and performing (N-1)-dimensional median filtering along the other dimensions.

**Parameters**

- `dif` (*float*) – Expected difference value between outlier value and the median value of the array.
- `size` (*int*) – Size of the median filter.
- `axis` (*int, optional*) – Axis along which to chunk.
- `ncore` (*int, optional*) – Number of cores that will be assigned to jobs.
- `out` (*ndarray, optional*) – Output array for result. If same as `arr`, process will be done in-place.

**Returns**

`ndarray` – Corrected array.

**tomopy.misc.corr.remove_outlier1d**

```
tomopy.misc.corr.remove_outlier1d(arr, dif, size=3, axis=0, ncore=None, out=None)
```

Remove high intensity bright spots from an array, using a one-dimensional median filter along the specified axis.

**Parameters**

- `dif` (*float*) – Expected difference value between outlier value and the median value of the array.
- `size` (*int*) – Size of the median filter.
- `axis` (*int, optional*) – Axis along which median filtering is performed.
- `ncore` (*int, optional*) – Number of cores that will be assigned to jobs.
- `out` (*ndarray, optional*) – Output array for result. If same as `arr`, process will be done in-place.

**Returns**

`ndarray` – Corrected array.

**tomopy.misc.corr.remove_outlier_cuda**

```
tomopy.misc.corr.remove_outlier_cuda(arr, dif, size=3, axis=0)
```

Remove high intensity bright spots from a 3D array along axis 0 dimension using GPU.

**Parameters**

- `dif` (*float*) – Expected difference value between outlier value and the median value of the array.
- `size` (*int*) – Size of the median filter.
- `axis` (*int, optional*) – Axis along which outlier removal is performed.

**Returns**

`ndarray` – Corrected array.

**Example**

```python
>>> import tomocuda

>>> tomocuda.remove_outlier_cuda(arr, dif, 5)
```
For more information regarding install and using tomocuda, check https://github.com/kyuepublic/tomocuda for more information

tomopy.misc.corr.remove_ring(rec, center_x=None, center_y=None, thresh=300.0, thresh_max=300.0, thresh_min=-100.0, theta_min=30, rwidth=30, int_mode='WRAP', ncore=None, nchunk=None, out=None)

Remove ring artifacts from images in the reconstructed domain. Descriptions of parameters need to be more clear for sure.

Parameters
- **arr** (*ndarray*) – Array of reconstruction data
- **center_x** (*float, optional*) – abscissa location of center of rotation
- **center_y** (*float, optional*) – ordinate location of center of rotation
- **thresh** (*float, optional*) – maximum value of an offset due to a ring artifact
- **thresh_max** (*float, optional*) – max value for portion of image to filter
- **thresh_min** (*float, optional*) – min value for portion of image to filter
- **theta_min** (*int, optional*) – minimum angle in degrees (int) to be considered ring artifact
- **rwidth** (*int, optional*) – Maximum width of the rings to be filtered in pixels
- **int_mode** (*str, optional*) – ‘WRAP’ for wrapping at 0 and 360 degrees, ‘REFLECT’ for reflective boundaries at 0 and 180 degrees.
- **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.
- **nchunk** (*int, optional*) – Chunk size for each core.
- **out** (*ndarray, optional*) – Output array for result. If same as arr, process will be done in-place.

Returns **ndarray** – Corrected reconstruction data

tomopy.misc.corr.enhance_projs_aps_1id(imgstack, median_ks=5, ncore=None)

Enhance the projection images with weak contrast collected at APS 1ID

This filter uses a median filterer (will be switched to enhanced recursive median filterer, ERMF, in the future) for denoising, and a histogram equalization for dynamic range adjustment to bring out the details.

Parameters
- **imgstack** (*np.ndarray*) – tomopy images stacks (axis_0 is the oemga direction)
- **median_ks** (*int, optional*) – 2D median filter kernel size for local noise suppression
- **ncore** (*int, optional*) – number of cores used for speed up

Returns **ndarray** – 3D enhanced image stacks.

5.7.2 tomopy.misc.morph

Module for data size morphing functions.

Functions:
**TomoPy Documentation**

- **downsample** \(\text{arr, level, axis}\)  
  Downsample along specified axis of a 3D array.

- **upsample** \(\text{arr, level, axis}\)  
  Upsample along specified axis of a 3D array.

- **pad** \(\text{arr, axis[, npad, mode, ncore]}\)  
  Pad an array along specified axis.

- **sino_360_to_180** \(\text{data[, overlap, rotation]}\)  
  Converts 0-360 degrees sinogram to a 0-180 sinogram.

- **trim_sinogram** \(\text{data, center, x, y, diameter}\)  
  Provide sinogram corresponding to a circular region of interest by trimming the complete sinogram of a compact object.

```python
import tomopy.misc.morph as morph

downsample = morph.downsample(arr, level=1, axis=2)  
Downsample along specified axis of a 3D array.

Parameters
  - \text{arr \ (ndarray)} \ – \ 3D input array.
  - \text{level \ (int, optional)} \ – \ Downsampling level in powers of two.
  - \text{axis \ (int, optional)} \ – \ Axis along which downsampling will be performed.

Returns  \text{ndarray} \ – \ Downsampling level in powers of two.

upsample = morph.upsample(arr, level=1, axis=2)  
Upsample along specified axis of a 3D array.

Parameters
  - \text{arr \ (ndarray)} \ – \ 3D input array.
  - \text{level \ (int, optional)} \ – \ Downsampling level in powers of two.
  - \text{axis \ (int, optional)} \ – \ Axis along which upsampling will be performed.

Returns  \text{ndarray} \ – \ Upsampled 3D array in float32.

pad = morph.pad(arr, axis, npad=None, mode='constant', ncore=None, **kwargs)  
Pad an array along specified axis.

Parameters
  - \text{arr \ (ndarray)} \ – \ Input array.
  - \text{axis \ (int)} \ – \ Axis along which padding will be performed.
  - \text{npad \ (int, optional)} \ – \ New dimension after padding.
  - \text{mode \ (str or function)} \ – \ One of the following string values or a user supplied function.
  - \text{constant} \ – \ Pads with a constant value.
  - \text{edge} \ – \ Pads with the edge values of array.
  - \text{constant values \ (float, optional)} \ – \ Used in ‘constant’. Pad value
  - \text{ncore \ (int, optional)} \ – \ Number of cores that will be assigned to jobs.

Returns  \text{ndarray} \ – \ Padded 3D array.

sino_360_to_180 = morph.sino_360_to_180(data, overlap=0, rotation='left')  
Converts 0-360 degrees sinogram to a 0-180 sinogram.

If the number of projections in the input data is odd, the last projection will be discarded.

Parameters
  - \text{data \ (ndarray)} \ – \ Input 3D data.
- **overlap** *(scalar, optional)* – Overlapping number of pixels.
- **rotation** *(string, optional)* – Left if rotation center is close to the left of the field-of-view, right otherwise.

**Returns** *ndarray* – Output 3D data.

tomopy.misc.morph.sino_360_to_180 *(data, overlap=0, rotation='left')*

Converts 0-360 degrees sinogram to a 0-180 sinogram.

If the number of projections in the input data is odd, the last projection will be discarded.

**Parameters**
- **data** *(ndarray)* – Input 3D data.
- **overlap** *(scalar, optional)* – Overlapping number of pixels.
- **rotation** *(string, optional)* – Left if rotation center is close to the left of the field-of-view, right otherwise.

**Returns** *ndarray* – Output 3D data.

tomopy.misc.morph.trim_sinogram *(data, center, x, y, diameter)*

Provide sinogram corresponding to a circular region of interest by trimming the complete sinogram of a compact object.

**Parameters**
- **data** *(ndarray)* – Input 3D data.
- **center** *(float)* – Rotation center location.
- **x, y** *(int, int)* – x and y coordinates in pixels (image center is (0, 0))
- **diameter** *(float)* – Diameter of the circle of the region of interest.

**Returns** *ndarray* – Output 3D data.

### 5.7.3 *tomopy.misc.phantom*

Module for generating synthetic phantoms.

**Functions:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>baboon*(size, dtype)*</td>
<td>Load test baboon image array.</td>
</tr>
<tr>
<td>barbara*(size, dtype)*</td>
<td>Load test Barbara image array.</td>
</tr>
<tr>
<td>cameraman*(size, dtype)*</td>
<td>Load test cameraman image array.</td>
</tr>
<tr>
<td>checkerboard*(size, dtype)*</td>
<td>Load test checkerboard image array.</td>
</tr>
<tr>
<td>lena*(size, dtype)*</td>
<td>Load test Lena image array.</td>
</tr>
<tr>
<td>peppers*(size, dtype)*</td>
<td>Load test peppers image array.</td>
</tr>
<tr>
<td>phantom*(size, params[, dtype])</td>
<td>Generate a cube of given size using a list of ellipsoid parameters.</td>
</tr>
<tr>
<td>shepp2d*(size, dtype)*</td>
<td>Load test Shepp-Logan image array.</td>
</tr>
<tr>
<td>shepp3d*(size, dtype)*</td>
<td>Load 3D Shepp-Logan image array.</td>
</tr>
</tbody>
</table>

tomopy.misc.phantom.baboon *(size=512, dtype='float32')*

Load test baboon image array.

**Parameters**

### 5.7. API reference
- **size** (*int or tuple of int, optional*) – Size of the output image.
- **dtype** (*str, optional*) – The desired data-type for the array.

**Returns**

`ndarray` – Output 3D test image.

tomopy.misc.phantom.barbara

`size=512, dtype='float32'`  
Load test Barbara image array.

**Parameters**

- **size** (*int or tuple of int, optional*) – Size of the output image.
- **dtype** (*str, optional*) – The desired data-type for the array.

**Returns**

`ndarray` – Output 3D test image.

tomopy.misc.phantom.cameraman

`size=512, dtype='float32'`  
Load test cameraman image array.

**Parameters**

- **size** (*int or tuple of int, optional*) – Size of the output image.
- **dtype** (*str, optional*) – The desired data-type for the array.

**Returns**

`ndarray` – Output 3D test image.

tomopy.misc.phantom.checkerboard

`size=512, dtype='float32'`  
Load test checkerboard image array.

**Parameters**

- **size** (*int or tuple of int, optional*) – Size of the output image.
- **dtype** (*str, optional*) – The desired data-type for the array.

**Returns**

`ndarray` – Output 3D test image.

tomopy.misc.phantom.lena

`size=512, dtype='float32'`  
Load test Lena image array.

**Parameters**

- **size** (*int or tuple of int, optional*) – Size of the output image.
- **dtype** (*str, optional*) – The desired data-type for the array.

**Returns**

`ndarray` – Output 3D test image.

tomopy.misc.phantom.peppers

`size=512, dtype='float32'`  
Load test peppers image array.

**Parameters**

- **size** (*int or tuple of int, optional*) – Size of the output image.
- **dtype** (*str, optional*) – The desired data-type for the array.

**Returns**

`ndarray` – Output 3D test image.

tomopy.misc.phantom.shepp2d

`size=512, dtype='float32'`  
Load test Shepp-Logan image array.

**Parameters**

- **size** (*int or tuple of int, optional*) – Size of the output image.
- **dtype** (*str, optional*) – The desired data-type for the array.
Returns \(\text{ndarray}\) – Output 3D test image.

tomopy.misc.phantom.\texttt{shepp3d}(\texttt{size}=128, \texttt{dtype}=\texttt{’float32’})

Load 3D Shepp-Logan image array.

Parameters

- \texttt{size} (\textit{int or tuple, optional}) – Size of the 3D data.
- \texttt{dtype} (\textit{str, optional}) – The desired data-type for the array.

Returns \(\text{ndarray}\) – Output 3D test image.

tomopy.misc.phantom.\texttt{phantom}(\texttt{size}, \texttt{params}, \texttt{dtype}=\texttt{’float32’})

Generate a cube of given size using a list of ellipsoid parameters.

Parameters

- \texttt{size} (\textit{tuple of int}) – Size of the output cube.
- \texttt{params} (\textit{list of dict}) – List of dictionaries with the parameters defining the ellipsoids to include in the cube.
- \texttt{dtype} (\textit{str, optional}) – Data type of the output \(\text{ndarray}\).

Returns \(\text{ndarray}\) – 3D object filled with the specified ellipsoids.

5.7.4 \texttt{tomopy.prep.alignment}

Functions:

- \texttt{align_seq}(\texttt{prj}, \texttt{ang[, fdir, iters, pad, ...]})
  Aligns the projection image stack using the sequential re-projection algorithm [C8].

- \texttt{align_joint}(\texttt{prj}, \texttt{ang[, fdir, iters, pad, ...]})
  Aligns the projection image stack using the joint re-projection algorithm [C8].

- \texttt{add_jitter}(\texttt{prj[, low, high]})
  Simulates jitter in projection images.

- \texttt{add_noise}(\texttt{prj[, ratio]})
  Adds Gaussian noise with zero mean and a given standard deviation as a ratio of the maximum value in data.

- \texttt{blur_edges}(\texttt{prj[, low, high]})
  Blurs the edge of the projection images.

- \texttt{shift_images}(\texttt{prj, low, high])
  Shift projections images for a given set of shift values in horizontal and vertical directions.

- \texttt{scale}(\texttt{prj})
  Linearly scales the projection images in the range between -1 and 1.

- \texttt{tilt}(\texttt{obj[, rad, phi]})
  Tilt object at a given angle from the rotation axis.

- \texttt{distortion_correction_proj}(\texttt{tomo, xcenter, ...})
  Apply distortion correction to projections using the polynomial model.

- \texttt{distortion_correction_sino}(\texttt{tomo, ind, ...})
  Generate an unwarped sinogram of a 3D tomographic data using the polynomial model.

\texttt{tomopy.prep.alignment.\texttt{align_seq}(\texttt{prj, ang[, fdir=’’, iters=10, pad=(0, 0), blur=True, center=None, algorithm=’sirt’, upsample_factor=10, rin=0.5, rout=0.8, save=False, debug=True})}

Aligns the projection image stack using the sequential re-projection algorithm [C8].

Parameters

- \texttt{prj} (\texttt{ndarray}) – 3D stack of projection images. The first dimension is projection axis, second and third dimensions are the x- and y-axes of the projection image, respectively.
• **ang** *(ndarray)* – Projection angles in radians as an array.
• **iters** *(scalar, optional)* – Number of iterations of the algorithm.
• **pad** *(list-like, optional)* – Padding for projection images in x and y-axes.
• **blur** *(bool, optional)* – Blurs the edge of the image before registration.
• **center** *(array, optional)* – Location of rotation axis.
• **algorithm** *(str, function)* – One of the following string values.
  • ‘art’ Algebraic reconstruction technique [C2].
  • ‘gridrec’ Fourier grid reconstruction algorithm [C5], [C21].
  • ‘mlem’ Maximum-likelihood expectation maximization algorithm [C3].
  • ‘sirt’ Simultaneous algebraic reconstruction technique.
  • ‘tv’ Total Variation reconstruction technique [C7].
  • ‘grad’ Gradient descent method with a constant step size
• **upsample_factor** *(integer, optional)* – The upsampling factor. Registration accuracy is inversely proportional to upsample_factor.
• **rin** *(scalar, optional)* – The inner radius of blur function. Pixels inside rin is set to one.
• **rout** *(scalar, optional)* – The outer radius of blur function. Pixels outside rout is set to zero.
• **save** *(bool, optional)* – Saves projections and corresponding reconstruction for each algorithm iteration.
• **debug** *(book, optional)* – Provides debugging info such as iterations and error.

**Returns**

• **ndarray** – 3D stack of projection images with jitter.
• **ndarray** – Error array for each iteration.

```python
from tomopy.prep.alignment import align_joint

align_joint(prj, ang, fdir='.', iters=10, pad=(0, 0), blur=True, center=None, algorithm='sirt', upsample_factor=10, rin=0.5, rout=0.8, save=False, debug=True)
```

Aligns the projection image stack using the joint re-projection algorithm [C8].

**Parameters**

• **prj** *(ndarray)* – 3D stack of projection images. The first dimension is projection axis, second and third dimensions are the x- and y-axes of the projection image, respectively.
• **ang** *(ndarray)* – Projection angles in radians as an array.
• **iters** *(scalar, optional)* – Number of iterations of the algorithm.
• **pad** *(list-like, optional)* – Padding for projection images in x and y-axes.
• **blur** *(bool, optional)* – Blurs the edge of the image before registration.
• **center** *(array, optional)* – Location of rotation axis.
• **algorithm** *(str, function)* – One of the following string values.
  • ‘art’ Algebraic reconstruction technique [C2].
  • ‘gridrec’ Fourier grid reconstruction algorithm [C5], [C21].
  • ‘mlem’ Maximum-likelihood expectation maximization algorithm [C3].
‘sirt’ Simultaneous algebraic reconstruction technique.

‘tv’ Total Variation reconstruction technique [C7].

‘grad’ Gradient descent method with a constant step size

- **upsample_factor (integer, optional)** – The upsampling factor. Registration accuracy is inversely proportional to upsample_factor.
- **rin (scalar, optional)** – The inner radius of blur function. Pixels inside rin is set to one.
- **rout (scalar, optional)** – The outer radius of blur function. Pixels outside rout is set to zero.
- **save (bool, optional)** – Saves projections and corresponding reconstruction for each algorithm iteration.
- **debug (book, optional)** – Provides debugging info such as iterations and error.

**Returns**

- **ndarray** – 3D stack of projection images with jitter.
- **ndarray** – Error array for each iteration.

```python
tomopy.prep.alignment.scale(prj)
```

Linearly scales the projection images in the range between -1 and 1.

**Parameters**

- **prj (ndarray)** – 3D stack of projection images. The first dimension is projection axis, second and third dimensions are the x- and y-axes of the projection image, respectively.

**Returns**

- **ndarray** – Scaled 3D stack of projection images.

```python
tomopy.prep.alignment.tilt(obj, rad=0, phi=0)
```

Tilt object at a given angle from the rotation axis.

**Warning:** Not implemented yet.

**Parameters**

- **obj (ndarray)** – 3D discrete object.
- **rad (scalar, optional)** – Radius in polar coordinates to define tilt angle. The value is between 0 and 1, where 0 means no tilt and 1 means a tilt of 90 degrees. The tilt angle can be obtained by arcsin(rad).
- **phi (scalar, optional)** – Angle in degrees to define tilt direction from the rotation axis. 0 degree means rotation in sagittal plane and 90 degree means rotation in coronal plane.

**Returns**

- **ndarray** – Tilted 3D object.

```python
tomopy.prep.alignment.add_jitter(prj, low=0, high=1)
```

Simulates jitter in projection images. The jitter is simulated by drawing random samples from a uniform distribution over the half-open interval [low, high).

**Parameters**

- **prj (ndarray)** – 3D stack of projection images. The first dimension is projection axis, second and third dimensions are the x- and y-axes of the projection image, respectively.
- **low (float, optional)** – Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.
- **high** *(float)* – Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

**Returns** *ndarray* – 3D stack of projection images with jitter.

tomopy.prep.alignment.add_noise *(prj, ratio=0.05)*

Adds Gaussian noise with zero mean and a given standard deviation as a ratio of the maximum value in data.

**Parameters**

- **prj** *(ndarray)* – 3D stack of projection images. The first dimension is projection axis, second and third dimensions are the x- and y-axes of the projection image, respectively.
- **ratio** *(float, optional)* – Ratio of the standard deviation of the Gaussian noise distribution to the maximum value in data.

**Returns** *ndarray* – 3D stack of projection images with added Gaussian noise.

tomopy.prep.alignment.blur_edges *(prj, low=0, high=0.8)*

Blurs the edge of the projection images.

**Parameters**

- **prj** *(ndarray)* – 3D stack of projection images. The first dimension is projection axis, second and third dimensions are the x- and y-axes of the projection image, respectively.
- **low** *(scalar, optional)* – Min ratio of the blurring frame to the image size.
- **high** *(scalar, optional)* – Max ratio of the blurring frame to the image size.

**Returns** *ndarray* – Edge-blurred 3D stack of projection images.

tomopy.prep.alignment.shift_images *(prj, sx, sy)*

Shift projections images for a given set of shift values in horizontal and vertical directions.

tomopy.prep.alignment.find_slits_corners_aps_1id *(img, method='quadrant+', medfilt2_kernel_size=3, medfilt_kernel_size=23)*

Automatically locate the slit box location by its four corners.

**Parameters**

- **img** *(np.ndarray)* – 2D images
- **method** *(str, ['simple', 'quadrant', 'quadrant+'], optional)* –
  - **simple** :: assume a rectangle slit box, fast but less accurate (1 pixel precision)
  - **quadrant** :: subdivide the image into four quadrant, then use an explicit method to find the corner (1 pixel precision)
  - **quadrant+** :: similar to quadrant, but use curve_fit (gauss1d) to find the corner (0.1 pixel precision)
- **medfilt2_kernel_size** *(int, optional)* – 2D median filter kernel size for noise reduction
- **medfilt_kernel_size** *(int, optional)* – 1D median filter kernel size for noise reduction
Returns `tuple` – autodetected slit corners (counter-clockwise order) (upperLeft, lowerLeft, lowerRight, upperRight)

tomopy.prep.alignment.calc_slit_box_aps_1id(slit_box_corners, inclip=(1, 10, 1, 10))

Calculate the clip box based on given slip corners.

Parameters

- `slit_box_corners` (`np.ndarray`) – Four corners of the slit box as a 4x2 matrix
- `inclip` (`tuple`, `optional`) – Extra inclipping to avoid clipping artifacts

Returns `Tuple` – Clipping indices as a tuple of four (clipFromTop, clipToBottom, clipFromLeft, clipToRight)

tomopy.prep.alignment.remove_slits_aps_1id(imgstacks, slit_box_corners, inclip=(1, 10, 1, 10))

Remove the slits from still images

Parameters

- `imgstacks` (`np.ndarray`) – tomopy images stacks (axis_0 is the omega direction)
- `slit_box_corners` (`np.ndarray`) – four corners of the slit box
- `inclip` (`tuple`, `optional`) – Extra inclipping to avoid clipping artifacts

Returns `np.ndarray` – tomopy images stacks without regions outside slits

tomopy.prep.alignment.distortion_correction_proj(tomo, xcenter, ycenter, list_fact, ncore=None, nchunk=None)

Apply distortion correction to projections using the polynomial model. Coefficients are calculated using Vounwarp package.

Parameters

- `tomo` (`ndarray`) – 3D tomographic data.
- `xcenter` (`float`) – Center of distortion in x-direction. From the left of the image.
- `ycenter` (`float`) – Center of distortion in y-direction. From the top of the image.
- `list_fact` (`list of floats`) – Polynomial coefficients of the backward model.
- `ncore` (`int`, `optional`) – Number of cores that will be assigned to jobs.
- `nchunk` (`int`, `optional`) – Chunk size for each core.

Returns `ndarray` – Corrected 3D tomographic data.

tomopy.prep.alignment.distortion_correction_sino(tomo, ind, xcenter, ycenter, list_fact)

Generate an unwarped sinogram of a 3D tomographic data using the polynomial model. Coefficients are calculated using Vounwarp package.

Parameters

- `tomo` (`ndarray`) – 3D tomographic data.
- `ind` (`int`) – Index of the unwarped sinogram.
- `xcenter` (`float`) – Center of distortion in x-direction. From the left of the image.
- `ycenter` (`float`) – Center of distortion in y-direction. From the top of the image.
- `list_fact` (`list of floats`) – Polynomial coefficients of the backward model.

Returns `2D array` – Corrected sinogram.
tomopy.prep.alignment.load_distortion_coefs (file_path)
Load distortion coefficients from a text file. Order of the infor in the text file: xcenter ycenter factor_0 factor_1 factor_2 ...

Parameters

file_path (Path to the file.)

Returns

Tuple of (xcenter, ycenter, list_fact).

5.7.5 tomopy.prep.normalize
Module for data normalization.

Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minus_log(arr[, ncore, out])</td>
<td>Computation of the minus log of a given array.</td>
</tr>
<tr>
<td>normalize(arr, flat, dark[, cutoff, ncore, out])</td>
<td>Normalize raw projection data using the flat and dark field projections.</td>
</tr>
<tr>
<td>normalize_bg(tomo[, air, ncore, nchunk])</td>
<td>Normalize 3D tomography data based on background intensity.</td>
</tr>
<tr>
<td>normalize_nf(tomo, flats, dark, flat_loc[, ...])</td>
<td>Normalize raw 3D projection data with flats taken more than once during tomography.</td>
</tr>
<tr>
<td>normalize_roi(arr[, roi, ncore])</td>
<td>Normalize raw projection data using an average of a selected window on projection images.</td>
</tr>
</tbody>
</table>

tomopy.prep.normalize.minus_log (arr, ncore=None, out=None)
Computation of the minus log of a given array.

Parameters

- **arr** (ndarray) – 3D stack of projections.
- **ncore** (int, optional) – Number of cores that will be assigned to jobs.
- **out** (ndarray, optional) – Output array for result. If same as arr, process will be done in-place.

Returns

ndarray – Minus-log of the input data.

tomopy.prep.normalize.normalize (arr, flat, dark, cutoff=None, ncore=None, out=None)
Normalize raw projection data using the flat and dark field projections.

Parameters

- **arr** (ndarray) – 3D stack of projections.
- **flat** (ndarray) – 3D flat field data.
- **dark** (ndarray) – 3D dark field data.
- **cutoff** (float, optional) – Permitted maximum value for the normalized data.
- **ncore** (int, optional) – Number of cores that will be assigned to jobs.
- **out** (ndarray, optional) – Output array for result. If same as arr, process will be done in-place.

Returns

ndarray – Normalized 3D tomographic data.

tomopy.prep.normalize.normalize_bg (tomo, air=1, ncore=None, nchunk=None)
Normalize 3D tomography data based on background intensity.
Weight sinogram such that the left and right image boundaries (i.e., typically the air region around the object) are set to one and all intermediate values are scaled linearly.

**Parameters**

- **tomo** *(ndarray)* – 3D tomographic data.
- **air** *(int, optional)* – Number of pixels at each boundary to calculate the scaling factor.
- **ncore** *(int, optional)* – Number of cores that will be assigned to jobs.
- **nchunk** *(int, optional)* – Chunk size for each core.

**Returns** *ndarray* – Corrected 3D tomographic data.

tomopy.prep.normalize.normalize_roi(arr, roi=[0, 0, 10, 10], ncore=None)

Normalize raw projection data using an average of a selected window on projection images.

**Parameters**

- **arr** *(ndarray)* – 3D tomographic data.
- **roi** *(list of int, optional)* – [top-left, top-right, bottom-left, bottom-right] pixel coordinates.
- **ncore** *(int, optional)* – Number of cores that will be assigned to jobs.

**Returns** *ndarray* – Normalized 3D tomographic data.

tomopy.prep.normalize.normalize_nf(tomo, flats, dark, flat_loc, cutoff=None, ncore=None, out=None)

Normalize raw 3D projection data with flats taken more than once during tomography. Normalization for each projection is done with the mean of the nearest set of flat fields (nearest flat fields).

**Parameters**

- **tomo** *(ndarray)* – 3D tomographic data.
- **flats** *(ndarray)* – 3D flat field data.
- **dark** *(ndarray)* – 3D dark field data.
- **flat_loc** *(list of int)* – Indices of flat field data within tomography
- **ncore** *(int, optional)* – Number of cores that will be assigned to jobs.
- **out** *(ndarray, optional)* – Output array for result. If same as arr, process will be done in-place.

**Returns** *ndarray* – Normalized 3D tomographic data.

### 5.7.6 tomopy.prep.phase

Module for phase retrieval.

**Functions:**

- **retrieve_phase**(tomo[, pixel_size, dist, ...]) Perform single-step phase retrieval from phase-contrast measurements [C9].
Perform single-step phase retrieval from phase-contrast measurements [C9].

Parameters

- **tomo** (*ndarray*) – 3D tomographic data.
- **pixel_size** (*float, optional*) – Detector pixel size in cm.
- **dist** (*float, optional*) – Propagation distance of the wavefront in cm.
- **energy** (*float, optional*) – Energy of incident wave in keV.
- **alpha** (*float, optional*) – Regularization parameter.
- **pad** (*bool, optional*) – If True, extend the size of the projections by padding with zeros.
- **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.
- **nchunk** (*int, optional*) – Chunk size for each core.

Returns *ndarray* – Approximated 3D tomographic phase data.

5.7.7 **tomopy.prep.stripe**

Module for pre-processing tasks.

**Functions:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>remove_stripe_fw</code></td>
<td>Remove horizontal stripes from sinogram using the Fourier-Wavelet (FW) based method [C4].</td>
</tr>
<tr>
<td><code>remove_stripe_ti</code></td>
<td>Remove horizontal stripes from sinogram using Titarenko’s approach [C11].</td>
</tr>
<tr>
<td><code>remove_stripe_sf</code></td>
<td>Normalize raw projection data using a smoothing filter approach.</td>
</tr>
<tr>
<td><code>remove_stripe_based_sorting</code></td>
<td>Remove stripe artifacts from sinogram using Nghia Vo’s approach [C22] Algorithm 3 in the paper.</td>
</tr>
<tr>
<td><code>remove_stripe_based_filtering</code></td>
<td>Remove stripe artifacts from sinogram using Nghia Vo’s approach [C22] Algorithm 2 in the paper.</td>
</tr>
<tr>
<td><code>remove_stripe_based_fitting</code></td>
<td>Remove horizontal stripes from sinogram using Nghia Vo’s approach [C22] Algorithm 1 in the paper.</td>
</tr>
<tr>
<td><code>remove_large_stripe</code></td>
<td>Remove stripe artifacts from sinogram using Nghia Vo’s approach [C22] Algorithm 5 in the paper.</td>
</tr>
<tr>
<td><code>remove_dead_stripe</code></td>
<td>Remove stripe artifacts from sinogram using Nghia Vo’s approach [C22] Algorithm 6 in the paper.</td>
</tr>
<tr>
<td><code>remove_all_stripe</code></td>
<td>Remove stripe artifacts from sinogram using Nghia Vo’s approach [C22] Combine algorithms 6,5,4,3 to remove all types of stripes.</td>
</tr>
</tbody>
</table>

**tomopy.prep.stripe.remove_stripe_fw**

Remove horizontal stripes from sinogram using the Fourier-Wavelet (FW) based method [C4].

Parameters

- **tomo** (*ndarray*) – 3D tomographic data.
• **level** (*int, optional*) – Number of discrete wavelet transform levels.
• **wname** (*str, optional*) – Type of the wavelet filter. ‘haar’, ‘db5’, ‘sym5’, etc.
• **sigma** (*float, optional*) – Damping parameter in Fourier space.
• **pad** (*bool, optional*) – If True, extend the size of the sinogram by padding with zeros.
• **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.
• **nchunk** (*int, optional*) – Chunk size for each core.

**Returns** *ndarray* – Corrected 3D tomographic data.

tomopy.prep.stripe.remove_stripe_ti(tomo, nblock=0, alpha=1.5, ncore=None, nchunk=None)

Remove horizontal stripes from sinogram using Titarenko’s approach [C11].

**Parameters**

• **tomo** (*ndarray*) – 3D tomographic data.
• **nblock** (*int, optional*) – Number of blocks.
• **alpha** (*int, optional*) – Damping factor.
• **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.
• **nchunk** (*int, optional*) – Chunk size for each core.

**Returns** *ndarray* – Corrected 3D tomographic data.

tomopy.prep.stripe.remove_stripe_sf(tomo, size=5, ncore=None, nchunk=None)

Normalize raw projection data using a smoothing filter approach.

**Parameters**

• **tomo** (*ndarray*) – 3D tomographic data.
• **size** (*int*) – Size of the smoothing filter.
• **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.
• **nchunk** (*int, optional*) – Chunk size for each core.

**Returns** *ndarray* – Corrected 3D tomographic data.

tomopy.prep.stripe.remove_stripe_based_sorting(tomo, size=None, ncore=None, nchunk=None)

Remove stripe artifacts from sinogram using Nghia Vo’s approach [C22] Algorithm 3 in the paper. Remove stripes using the sorting technique. Work particularly well for removing partial stripes.

**Parameters**

• **tomo** (*ndarray*) – 3D tomographic data.
• **size** (*int*) – Window size of the median filter.
• **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.
• **nchunk** (*int, optional*) – Chunk size for each core.

**Returns** *ndarray* – Corrected 3D tomographic data.

tomopy.prep.stripe.remove_stripe_based_filtering(tomo, sigma=3, size=None, ncore=None, nchunk=None)

Remove stripe artifacts from sinogram using Nghia Vo’s approach [C22] Algorithm 2 in the paper. Remove stripes using the filtering technique.

**Parameters**
• **tomo** *(ndarray)* – 3D tomographic data.
• **sigma** *(float)* – Sigma of the Gaussian window which is used to separate the low-pass and high-pass components of the intensity profiles of each column.
• **size** *(int)* – Window size of the median filter.
• **ncore** *(int, optional)* – Number of cores that will be assigned to jobs.
• **nchunk** *(int, optional)* – Chunk size for each core.

**Returns** *ndarray* – Corrected 3D tomographic data.

```python
tomopy.prep.stripe.remove_stripe_based_fitting(tomo, order=3, sigma=(5, 20), ncore=None, nchunk=None)
```

Remove horizontal stripes from sinogram using Nghia Vo’s approach [C22] Algorithm 1 in the paper. Remove stripes using the fitting technique.

**Parameters**

• **tomo** *(ndarray)* – 3D tomographic data.
• **order** *(int)* – Polynomial fit order.
• **sigma** *(tuple of 2 floats)* – Sigmas of a 2D Gaussian window in x and y direction.
• **ncore** *(int, optional)* – Number of cores that will be assigned to jobs.
• **nchunk** *(int, optional)* – Chunk size for each core.

**Returns** *ndarray* – Corrected 3D tomographic data.

```python
tomopy.prep.stripe.remove_large_stripe(tomo, snr=3, size=51, ncore=None, nchunk=None)
```

Remove stripe artifacts from sinogram using Nghia Vo’s approach [C22] Algorithm 5 in the paper. Remove large stripes.

**Parameters**

• **tomo** *(ndarray)* – 3D tomographic data.
• **snr** *(float)* – Ratio used to locate of large stripes. Greater is less sensitive.
• **size** *(int)* – Window size of the median filter.
• **ncore** *(int, optional)* – Number of cores that will be assigned to jobs.
• **nchunk** *(int, optional)* – Chunk size for each core.

**Returns** *ndarray* – Corrected 3D tomographic data.

```python
tomopy.prep.stripe.remove_dead_stripe(tomo, snr=3, size=51, ncore=None, nchunk=None)
```

Remove unresponsive and fluctuating stripes.

**Parameters**

• **tomo** *(ndarray)* – 3D tomographic data.
• **snr** *(float)* – Ratio used to detect locations of large stripes. Greater is less sensitive.
• **size** *(int)* – Window size of the median filter.
• **ncore** *(int, optional)* – Number of cores that will be assigned to jobs.
• **nchunk** *(int, optional)* – Chunk size for each core.

**Returns** *ndarray* – Corrected 3D tomographic data.
tomopy.prep.stripe.remove_all_stripe(tomo, snr=3, la_size=61, sm_size=21, ncore=None, nchunk=None)

Remove stripe artifacts from sinogram using Nghia Vo’s approach [C22] Combine algorithms 6, 5, 4, 3 to remove all types of stripes.

Parameters

- **tomo** (*ndarray*) – 3D tomographic data.
- **snr** (*float*) – Ratio used to locate large stripes. Greater is less sensitive.
- **la_size** (*int*) – Window size of the median filter to remove large stripes.
- **sm_size** (*int*) – Window size of the median filter to remove small-to-medium stripes.
- **ncore** (*int, optional*) – Number of cores that will be assigned to jobs.
- **nchunk** (*int, optional*) – Chunk size for each core.

Returns *ndarray* – Corrected 3D tomographic data.

### 5.7.8 `tomopy.recon.algorithm`

Module for reconstruction algorithms.

**Functions:**

```
recon(tomo, theta[, center, sinogram_order, ...])  Reconstruct object from projection data.
```

tomopy.recon.algorithm.recon(tomo, theta, center=None, sinogram_order=False, algorithm=None, init_recon=None, ncore=None, nchunk=None, **kwargs)

Reconstruct object from projection data.

Parameters

- **tomo** (*ndarray*) – 3D tomographic data.
- **theta** (*array*) – Projection angles in radian.
- **center** (*array, optional*) – Location of rotation axis.
- **sinogram_order** (*bool, optional*) – Determines whether data is a stack of sinograms (True, y-axis first axis) or a stack of radiographs (False, theta first axis).
- **algorithm** (*{str, function}*]) – One of the following string values.
  - ‘art’ Algebraic reconstruction technique [C2].
  - ‘bart’ Block algebraic reconstruction technique.
  - ‘fbp’ Filtered back-projection algorithm.
  - ‘gridrec’ Fourier grid reconstruction algorithm [C5], [C21].
  - ‘mlem’ Maximum-likelihood expectation maximization algorithm [C3].
  - ‘osem’ Ordered-subset expectation maximization algorithm [C16].
  - ‘ospml_hybrid’ Ordered-subset penalized maximum likelihood algorithm with weighted linear and quadratic penalties.

---

5.7. API reference 45
‘ospml_quad’ Ordered-subset penalized maximum likelihood algorithm with quadratic penalties.

‘pml_hybrid’ Penalized maximum likelihood algorithm with weighted linear and quadratic penalties [C17].

‘pml_quad’ Penalized maximum likelihood algorithm with quadratic penalty.

‘sirt’ Simultaneous algebraic reconstruction technique.

‘tv’ Total Variation reconstruction technique [C7].

‘grad’ Gradient descent method.

‘tikh’ Tikhonov regularization with identity Tikhonov matrix.

• num_gridx, num_gridy (int, optional) – Number of pixels along x- and y-axes in the reconstruction grid.

• filter_name (str, optional) – Name of the filter for analytic reconstruction.
  ‘none’ No filter.
  ‘shepp’ Shepp-Logan filter (default).
  ‘cosine’ Cosine filter.
  ‘hann’ Cosine filter.
  ‘hamming’ Hamming filter.
  ‘ramlak’ Ram-Lak filter.
  ‘parzen’ Parzen filter.
  ‘butterworth’ Butterworth filter.
  ‘custom’ A numpy array of size next_power_of_2(num_detector_columns)/2 specifying a custom filter in Fourier domain. The first element of the filter should be the zero-frequency component.
  ‘custom2d’ A numpy array of size num_projections*next_power_of_2(num_detector_columns)/2 specifying a custom angle-dependent filter in Fourier domain. The first element of each filter should be the zero-frequency component.

• filter_par (list, optional) – Filter parameters as a list.

• num_iter (int, optional) – Number of algorithm iterations performed.

• num_block (int, optional) – Number of data blocks for intermediate updating the object.

• ind_block (array of int, optional) – Order of projections to be used for updating.

• reg_par (float, optional) – Regularization parameter for smoothing.

• init_recon (ndarray, optional) – Initial guess of the reconstruction.

• ncore (int, optional) – Number of cores that will be assigned to jobs.

• nchunk (int, optional) – Chunk size for each core.

Returns ndarray – Reconstructed 3D object.

Warning: Filtering is not implemented for fbp.
Example

```python
>>> import tomopy
>>> obj = tomopy.shepp3d() # Generate an object.
>>> ang = tomopy.angles(180) # Generate uniformly spaced tilt angles.
>>> sim = tomopy.project(obj, ang) # Calculate projections.
>>> rec = tomopy.recon(sim, ang, algorithm='art') # Reconstruct object.

# Show 64th slice of the reconstructed object.
>>> import pylab
>>> pylab.imshow(rec[64], cmap='gray')
>>> pylab.show()
```

Example using the ASTRA toolbox for reconstruction

For more information, see http://sourceforge.net/p/astra-toolbox/wiki/Home/ and https://github.com/astra-toolbox/astra-toolbox. To install the ASTRA toolbox with conda, use:

```bash
conda install -c https://conda.binstar.org/astra-toolbox astra-toolbox
```

```python
>>> import tomopy
>>> obj = tomopy.shepp3d() # Generate an object.
>>> ang = tomopy.angles(180) # Generate uniformly spaced tilt angles.
>>> sim = tomopy.project(obj, ang) # Calculate projections.

# Reconstruct object:
>>> rec = tomopy.recon(sim, ang, algorithm=tomopy.astra,
                       options={'method':'SART', 'num_iter':10*180,
                                'proj_type':'linear',
                                'extra_options':{'MinConstraint':0}})

# Show 64th slice of the reconstructed object.
>>> import pylab
>>> pylab.imshow(rec[64], cmap='gray')
>>> pylab.show()
```

tomopy.recon.algorithm.init_tomo(tomo, sinogram_order, sharedmem=True)

5.7.9 tomopy.recon.rotation

Module for functions related to finding axis of rotation.

Functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find_center</td>
<td>Find rotation axis location.</td>
</tr>
<tr>
<td>find_center_vo</td>
<td>Find rotation axis location using Nghia Vo’s method.</td>
</tr>
<tr>
<td>find_center_pcf</td>
<td>Find rotation axis location by finding the offset between the first projection and a mirrored projection 180 degrees apart using phase correlation in Fourier space.</td>
</tr>
<tr>
<td>write_center</td>
<td>Save images reconstructed with a range of rotation centers.</td>
</tr>
</tbody>
</table>

5.7. API reference
Find rotation axis location.

The function exploits systematic artifacts in reconstructed images due to shifts in the rotation center. It uses image entropy as the error metric and “Nelder-Mead” routine (of the scipy optimization module) as the optimizer [C10].

**Parameters**

- **tomo** *(ndarray)* – 3D tomographic data.
- **theta** *(array)* – Projection angles in radian.
- **ind** *(int, optional)* – Index of the slice to be used for reconstruction.
- **init** *(float)* – Initial guess for the center.
- **tol** *(scalar)* – Desired sub-pixel accuracy.
- **mask** *(bool, optional)* – If True, apply a circular mask to the reconstructed image to limit the analysis into a circular region.
- **ratio** *(float, optional)* – The ratio of the radius of the circular mask to the edge of the reconstructed image.
- **sinogram_order** *(bool, optional)* – Determines whether data is a stack of sinograms (True, y-axis first axis) or a stack of radiographs (False, theta first axis).

**Returns** *float* – Rotation axis location.

Find rotation axis location using Nghia Vo’s method. [C23].

**Parameters**

- **tomo** *(ndarray)* – 3D tomographic data.
- **ind** *(int, optional)* – Index of the slice to be used for reconstruction.
- **smin**, **smax** *(int, optional)* – Coarse search radius. Reference to the horizontal center of the sinogram.
- **srad** *(float, optional)* – Fine search radius.
- **step** *(float, optional)* – Step of fine searching.
- **ratio** *(float, optional)* – The ratio between the FOV of the camera and the size of object. It’s used to generate the mask.
- **drop** *(int, optional)* – Drop lines around vertical center of the mask.

**Returns** *float* – Rotation axis location.

Find rotation axis location by finding the offset between the first projection and a mirrored projection 180 degrees apart using phase correlation in Fourier space. The register_translation function uses cross-correlation in Fourier space, optionally employing an upsampled matrix-multiplication DFT to achieve arbitrary subpixel precision. [C14].

**Parameters**

- **proj1** *(ndarray)* – 2D projection data.
- **proj2** *(ndarray)* – 2D projection data.
• **tol** *(scalar, optional)* – Subpixel accuracy
• **rotc_guess** *(float, optional)* – Initial guess value for the rotation center

**Returns** *float* – Rotation axis location.

tomopy.recon.rotation.*write_center* *(tomo, theta, dpath='tmp/center', cen_range=None, ind=None, mask=False, ratio=1.0, sinogram_order=False, algorithm='gridrec', filter_name='parzen')*

Save images reconstructed with a range of rotation centers.

Helps finding the rotation center manually by visual inspection of images reconstructed with a set of different centers. The output images are put into a specified folder and are named by the center position corresponding to the image.

**Parameters**

• **tomo** *(ndarray)* – 3D tomographic data.
• **theta** *(array)* – Projection angles in radian.
• **dpath** *(str, optional)* – Folder name to save output images.
• **cen_range** *(list, optional)* – [start, end, step] Range of center values.
• **ind** *(int, optional)* – Index of the slice to be used for reconstruction.
• **mask** *(bool, optional)* – If True, apply a circular mask to the reconstructed image to limit the analysis into a circular region.
• **ratio** *(float, optional)* – The ratio of the radius of the circular mask to the edge of the reconstructed image.
• **sinogram_order** *(bool, optional)* – Determines whether data is a stack of sinograms (True, y-axis first axis) or a stack of radiographs (False, theta first axis).
• **algorithm** *(str, function)* – One of the following string values.
  • *art* – Algebraic reconstruction technique [C2].
  • *bart* – Block algebraic reconstruction technique.
  • *fbp* – Filtered back-projection algorithm.
  • *gridrec* – Fourier grid reconstruction algorithm [C5], [C21].
  • *mlem* – Maximum-likelihood expectation maximization algorithm [C3].
  • *osem* – Ordered-subset expectation maximization algorithm [C16].
  • *ospml_hybrid* – Ordered-subset penalized maximum likelihood algorithm with weighted linear and quadratic penalties.
  • *ospml_quad* – Ordered-subset penalized maximum likelihood algorithm with quadratic penalties.
  • *pml_hybrid* – Penalized maximum likelihood algorithm with weighted linear and quadratic penalties [C17].
  • *pml_quad* – Penalized maximum likelihood algorithm with quadratic penalty.
  • *sirt* – Simultaneous algebraic reconstruction technique.
  • *tv* – Total Variation reconstruction technique [C7].
  • *grad* – Gradient descent method with a constant step size
‘tikh’ Tikhonov regularization with identity Tikhonov matrix.

filter_name [str, optional] Name of the filter for analytic reconstruction.

‘none’ No filter.
‘shepp’ Shepp-Logan filter (default).
‘cosine’ Cosine filter.
‘hann’ Cosine filter.
‘hamming’ Hamming filter.
‘ramlak’ Ram-Lak filter.
‘parzen’ Parzen filter.
‘butterworth’ Butterworth filter.
‘custom’ A numpy array of size next_power_of_2(num_detector_columns)/2 specifying a custom filter in Fourier domain. The first element of the filter should be the zero-frequency component.
‘custom2d’ A numpy array of size num_projections*next_power_of_2(num_detector_columns)/2 specifying a custom angle-dependent filter in Fourier domain. The first element of each filter should be the zero-frequency component.

tomopy.recon.rotation.mask_empty_slice(tomo, threshold=0.25)
Generate a mask to indicate whether current slice contains sample

At APS 1ID, some of the projection images contains large empty area above the sample, resulting in empty layers.

Parameters

• tomo (ndarray) – 3D tomographic data.
• threshold (float, optional) – determine whether a layer is considered to be empty

Returns nparray – a mask indicate the emptiness of each layer

5.7.10 tomopy.sim.project
Module for simulation of x-rays.

Functions:

| angles(nang[, ang1, ang2]) | Return uniformly distributed projection angles in radian. |
| project(obj, theta[, center, emission, pad, ...]) | Project x-rays through a given 3D object. |
| fan_to_para(tomo, dist, geom) | Convert fan-beam data to parallel-beam data. |
| para_to_fan(tomo, dist, geom) | Convert parallel-beam data to fan-beam data. |
| add_gaussian(tomo[, mean, std]) | Add Gaussian noise. |
| add_poisson(tomo) | Add Poisson noise. |
| add_salt_pepper(tomo[, prob, val]) | Add salt and pepper noise. |
| add_focal_spot_blur(tomo, spotsize) | Add focal spot blur. |
tomopy.sim.project.angles (nang, ang1=0.0, ang2=180.0)
Return uniformly distributed projection angles in radian.

Parameters
- nang (int, optional) – Number of projections.
- ang1 (float, optional) – First projection angle in degrees.
- ang2 (float, optional) – Last projection angle in degrees.

Returns array – Projection angles

tomopy.sim.project.project (obj, theta, center=None, emission=True, pad=True, sinogram_order=False, ncore=None, nchunk=None)
Project x-rays through a given 3D object.

Parameters
- obj (ndarray) – Voxelized 3D object.
- theta (array) – Projection angles in radian.
- center (array, optional) – Location of rotation axis.
- emission (bool, optional) – Determines whether output data is emission or transmission type.
- pad (bool, optional) – Determines if the projection image width will be padded or not. If True, then the diagonal length of the object cross-section will be used for the output size of the projection image width.
- sinogram_order (bool, optional) – Determines whether output data is a stack of sinograms (True, y-axis first axis) or a stack of radiographs (False, theta first axis).
- ncore (int, optional) – Number of cores that will be assigned to jobs.
- nchunk (int, optional) – Chunk size for each core.

Returns ndarray – 3D tomographic data.

tomopy.sim.project.project2 (objx, objy, theta, center=None, emission=True, pad=True, sinogram_order=False, axis=0, ncore=None, nchunk=None)
Project x-rays through a given 3D object.

Parameters
- objx (ndarray) – (x, y) components of vector of a voxelized 3D object.
- theta (array) – Projection angles in radian.
- center (array, optional) – Location of rotation axis.
- emission (bool, optional) – Determines whether output data is emission or transmission type.
- pad (bool, optional) – Determines if the projection image width will be padded or not. If True, then the diagonal length of the object cross-section will be used for the output size of the projection image width.
- sinogram_order (bool, optional) – Determines whether output data is a stack of sinograms (True, y-axis first axis) or a stack of radiographs (False, theta first axis).
- ncore (int, optional) – Number of cores that will be assigned to jobs.
- nchunk (int, optional) – Chunk size for each core.

Returns ndarray – 3D tomographic data.
Project x-rays through a given 3D object.

**Parameters**

- **objx** (ndarray) – (x, y) components of vector of a voxelized 3D object.
- **theta** (array) – Projection angles in radian.
- **center** (array, optional) – Location of rotation axis.
- **emission** (bool, optional) – Determines whether output data is emission or transmission type.
- **pad** (bool, optional) – Determines if the projection image width will be padded or not. If True, then the diagonal length of the object cross-section will be used for the output size of the projection image width.
- **sinogram_order** (bool, optional) – Determines whether output data is a stack of sinograms (True, y-axis first axis) or a stack of radiographs (False, theta first axis).
- **ncore** (int, optional) – Number of cores that will be assigned to jobs.
- **nchunk** (int, optional) – Chunk size for each core.

**Returns**

ndarray – 3D tomographic data.

**tomopy.sim.project.fan_to_para** (tomo, dist, geom)

Convert fan-beam data to parallel-beam data.

**Warning:** Not implemented yet.

**Parameters**

- **tomo** (ndarray) – 3D tomographic data.
- **dist** (float) – Distance from fan-beam vertex to rotation center.
- **geom** (str) – Fan beam geometry. ‘arc’ or ‘line’.

**Returns**

ndarray – Transformed 3D tomographic data.

**tomopy.sim.project.para_to_fan** (tomo, dist, geom)

Convert parallel-beam data to fan-beam data.

**Warning:** Not implemented yet.

**Parameters**

- **tomo** (ndarray) – 3D tomographic data.
- **dist** (float) – Distance from fan-beam vertex to rotation center.
- **geom** (str) – Fan beam geometry. ‘arc’ or ‘line’.

**Returns**

ndarray – Transformed 3D tomographic data.

**tomopy.sim.project.add_gaussian** (tomo, mean=0, std=None)

Add Gaussian noise.
Parameters

- **tomo** *(ndarray)* – 3D tomographic data.
- **mean** *(float, optional)* – Mean of the Gaussian distribution.
- **std** *(float, optional)* – Standard deviation of the Gaussian distribution.

**Returns** *ndarray* – 3D tomographic data after Gaussian noise added.

tomopy.sim.project.add_poisson(tomo)

Add Poisson noise.

**Parameters**

- **tomo** *(ndarray)* – 3D tomographic data.

**Returns** *ndarray* – 3D tomographic data after Poisson noise added.

tomopy.sim.project.add_salt_pepper(tomo, prob=0.01, val=None)

Add salt and pepper noise.

**Parameters**

- **tomo** *(ndarray)* – 3D tomographic data.
- **prob** *(float, optional)* – Independent probability that each element of a pixel might be corrupted by the salt and pepper type noise.
- **val** *(float, optional)* – Value to be assigned to the corrupted pixels.

**Returns** *ndarray* – 3D tomographic data after salt and pepper noise added.

tomopy.sim.project.add_focal_spot_blur(tomo, spotsize)

Add focal spot blur.

**Warning:** Not implemented yet.

**Parameters**

- **tomo** *(ndarray)* – 3D tomographic data.
- **spotsize** *(float)* – Focal spot size of circular x-ray source.

5.7.11 tomopy.sim.propagate

Module for simulation of x-rays.

**Functions:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>propagate_tie</strong> (mu, delta, pixel_size, dist)</td>
<td>Propagate emitting x-ray wave based on Transport of Intensity.</td>
</tr>
<tr>
<td>tomopy.sim.propagate.calc_intensity (probe, proj, shift_x=None, shift_y=None, mode='near')</td>
<td>Calculate far field intensity.</td>
</tr>
</tbody>
</table>

**Parameters**

- **probe** *(ndarray)* – Rectangular x-ray source kernel.
- **proj** *(ndarray)* – Object plane intensity image.
TomoPy Documentation

- **shift_x, shift_y** *(int, optional)* – Shift amount of probe along x and y axes.
- **mode** *(str, optional)* – Specify the regime. ‘near’ or ‘far’

**Returns** *ndarray* – Individual raster scanned far field images as 3D array.

```
import tomopy

# Propagate emitting x-ray wave based on Transport of Intensity.

mu, delta, pixel_size, dist =..., ...

propagate_tie(mu, delta, pixel_size, dist)
```

**Parameters**

- **mu** *(ndarray, optional)* – 3D tomographic data for attenuation.
- **delta** *(ndarray)* – 3D tomographic data for refractive index.
- **pixel_size** *(float)* – Detector pixel size in cm.
- **dist** *(float)* – Propagation distance of the wavefront in cm.

**Returns** *ndarray* – 3D propagated tomographic intensity.

```
import tomopy

# Simulate incident x-ray beam (probe) as a square Gaussian kernel.

nx, ny =..., ...

fwhm =..., ...

center =..., ...

max_int = 1

probe_gauss(nx, ny, fwhm=None, center=None, max_int=1)
```

**Parameters**

- **nx, ny** *(int)* – Grid size along x and y axes.
- **fwhm** *(float, optional)* – Effective radius of the source.
- **center** *(array_like, optional)* – x and y coordinates of the center of the gaussian function.
- **max_int** *(int)* – Maximum x-ray intensity.

**Returns** *ndarray* – 2D source intensity distribution.

### 5.8 Examples

This section contains Jupyter Notebooks and Python scripts examples for various tomoPy functions.

To run these examples in a notebooks install Jupyter or run the python scripts from here

#### 5.8.1 Gridrec

Here is an example on how to use the gridrec [C5] reconstruction algorithm with TomoPy [A1]. You can download the python script here or the Jupyter notebook here

```
%pylab inline
```

Install TomoPy then:

```
import tomopy
```

Tomographic data input in TomoPy is supported by DXchange.

```
import dxchange
```

matplotlib provide plotting of the result in this notebook. Paraview or other tools are available for more sophisticated 3D rendering.
```python
import matplotlib.pyplot as plt

Set the path to the micro-CT data to reconstruct.
fname = '../../tomopy/data/tooth.h5'

Select the sinogram range to reconstruct.
start = 0
end = 2

tooth.h5 data set file format follows the APS beamline 2-BM and 32-ID data-exchange file format definition. Major synchrotron file format examples are available at DXchange.

proj, flat, dark, theta = dxchange.read_aps_32id(fname, sino=(start, end))

Plot the sinogram:
plt.imshow(proj[:, 0, :], cmap='Greys_r')
plt.show()

|
| 0 |
| 50 |
| 100 |
| 150 |
| 0 | 100 | 200 | 300 | 400 | 500 | 600 |

If the angular information is not available from the raw data you need to set the data collection angles. In this case theta is set as equally spaced between 0-180 degrees.

```python
if (theta is None):
    theta = tomopy.angles(proj.shape[0])
else:
    pass
```

Perform the flat-field correction of raw data:

```
proj = tomopy.normalize(proj, flat, dark)
```

Tomopy provides various methods ([C10], [C23], [C14]) to find the rotation center.

```
rot_center = tomopy.find_center(proj, theta, init=290, ind=0, tol=0.5)
tomopy.rotation:Trying center: [ 290.]
tomopy.rotation:Trying center: [ 304.5]
tomopy.rotation:Trying center: [ 275.5]
tomopy.rotation:Trying center: [ 282.75]
tomopy.rotation:Trying center: [ 297.25]
tomopy.rotation:Trying center: [ 304.5]
```

(continues on next page)
Calculate

\[ -\log(\text{proj}) \]

```
proj = tomopy.minus_log(proj)
```

Reconstruction using Gridrec algorithm. Tomopy provides various reconstruction methods including the one part of the ASTRA toolbox.

```
recon = tomopy.recon(proj, theta, center=rot_center, algorithm='gridrec')
```

Mask each reconstructed slice with a circle.

```
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
```

```
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```

### 5.8.2 Using the ASTRA toolbox through TomoPy

Here is an example on how to use the ASTRA toolbox through its integration with TomoPy, as published in [A2].

```
%pylab inline
```

Install the ASTRA toolbox and TomoPy then:
```python
import tomopy

DXchange is installed with tomopy to provide support for tomographic data loading. Various data format from all major synchrotron facilities are supported.

```python
import dxchange
```

Matplotlib provide plotting of the result in this notebook. Paraview or other tools are available for more sophisticated 3D rendering.

```python
import matplotlib.pyplot as plt
```

Set the path to the micro-CT data to reconstruct.

```python
fname = '../../tomopy/data/tooth.h5'
```

Select the sinogram range to reconstruct.

```python
start = 0
end = 2
```

This data set file format follows the APS beamline 2-BM and 32-ID definition. Other file format readers are available at DXChange.

```python
proj, flat, dark, theta = dxchange.read_aps_32id(fname, sino=(start, end))
```

Plot the sinogram:

```python
plt.imshow(proj[:, 0, :], cmap='Greys_r')
plt.show()
```

If the angular information is not available from the raw data you need to set the data collection angles. In this case theta is set as equally spaced between 0-180 degrees.

```python
if (theta is None):
    theta = tomopy.angles(proj.shape[0])
else:
    pass
```

Perform the flat-field correction of raw data:

```
proj − dark
flat − dark
```

```python
proj = tomopy.normalize(proj, flat, dark)
```

Tomopy provides various methods to find rotation center.

---

**5.8. Examples**
rot_center = tomopy.find_center(proj, theta, init=290, ind=0, tol=0.5)

Calculate

$$-\log(proj)$$

proj = tomopy.minus_log(proj)

Reconstruction with TomoPy

Reconstruction can be performed using either TomoPy’s algorithms, or the algorithms of the ASTRA toolbox.

To compare, we first show how to reconstruct an image using TomoPy’s Gridrec algorithm:

recon = tomopy.recon(proj, theta, center=rot_center, algorithm='gridrec')

Mask each reconstructed slice with a circle.

recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)

plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
If you have a CUDA-capable NVIDIA GPU, reconstruction times can be greatly reduced by using GPU-based algorithms of the ASTRA toolbox, especially for iterative reconstruction methods.

To use the GPU, change the `proj_type` option to 'cuda', and use CUDA-specific algorithms (e.g. 'FBP_CUDA' for FBP):

```python
options = {'proj_type':'cuda', 'method':'FBP_CUDA'}
recon = tomopy.recon(proj, theta, center=rot_center, algorithm=tomopy.astra,_
→options=options)
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```

Many algorithms of the ASTRA toolbox support additional options, which can be found in the documentation. These options can be specified using the `extra_options` keyword.
For example, to use the GPU-based iterative SIRT method with a nonnegativity constraint, use:

```python
extra_options = {'MinConstraint': 0}
options = {'proj_type': 'cuda', 'method': 'SIRT_CUDA', 'num_iter': 200, 'extra_options': extra_options}
recon = tomopy.recon(proj, theta, center=rot_center, algorithm=tomopy.astra, options=options)
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```

### 5.8.3 Reconstruction with UFO

UFO is a general-purpose image processing framework developed at the Karlsruhe Institute of Technology and uses OpenCL to execute processing tasks on multiple accelerator devices such as NVIDIA and AMD GPUs, AMD and Intel CPUs as well as Intel Xeon Phi cards.

Here is an example on how to use TomoPy with UFO and its accompanying reconstruction algorithms.

Install TomoPy, ufo-core and ufo-filters. Make sure to install the Python NumPy interfaces in the `python` subdirectory of ufo-core.

DXchange is installed with tomopy to provide support for tomographic data loading. Various data format from all major synchrotron facilities are supported.

```python
import dxchange
import matplotlib.pyplot as plt
```

Set the path to the micro-CT dataset and the sinogram range to reconstruct.

```python
fname = 'tooth.h5'
start, end = (0, 2)
```

This dataset file format follows the APS beamline 2-BM and 32-ID definition. Other file format readers are available at DXchange.
proj, flat, dark, theta = dxchange.read_aps_32id(fname, sino=(start, end))

Plot the sinogram:

```python
plt.imshow(proj[:, 0, :], cmap='Greys_r')
plt.show()
```

If the angular information is not available from the raw data you need to set the data collection angles. In this case theta is set as equally spaced between 0-180 degrees.

```python
if (theta is None):
    theta = tomopy.angles(proj.shape[0])
else:
    pass
```

Perform the flat-field correction of raw data:

$$\frac{proj - dark}{flat - dark}$$

```python
proj = tomopy.normalize(proj, flat, dark)
```

Tomopy provides various methods to find rotation center.

```python
center = tomopy.find_center(proj, theta, init=290, ind=0, tol=0.5)
```

Calculate

$$-\log(proj)$$

```python
proj = tomopy.minus_log(proj)
```

Now, reconstruct using UFO’s filtered backprojection algorithm. Note, that we must set ncore to 1 in order to let UFO do the multi-threading. If left to the default value or set to a value other than 1 will crash the reconstruction.

```python
recon = tomopy.recon(proj, theta, center=center, algorithm=ufo_fbp, ncore=1)
```

Mask each reconstructed slice with a circle.

```python
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
```
5.8.4 Vector Reconstruction

The vector reconstruction algorithm can be used for instance, to reconstruct the magnetization vector field inside a magnetic sample.

Here is an example on how to use the vector reconstruction algorithm [B2] [A4] with TomoPy[A1].

From a reconstructed 3D object to its projections and back

In order to test the algorithm, the projections of a reconstructed object can be computed, and from these projections we can come back to the reconstructed model object. Finally we will compare the results of the vector field reconstruction against the initial object.

All datasets used in this tutorial are available in tomoBank.

First, let’s make the necessary imports

```python
import dxchange
import tomopy
import numpy as np
import matplotlib.pyplot as plt
import time

Let’s load the object: the three components of the magnetization vector all throughout the object. The object will be padded in order to have a cubic object. Afterwards it will be downsampled to make faster computations.

```python
obx = dxchange.read_tiff('M4R1_mx.tif').astype('float32')
obx = dxchange.read_tiff('M4R1_my.tif').astype('float32')
obx = dxchange.read_tiff('M4R1_mz.tif').astype('float32')
```
npad = ((182, 182), (64, 64), (0, 0))
obx = np.pad(obx, npad, mode='constant', constant_values=0)
oby = np.pad(oby, npad, mode='constant', constant_values=0)
obz = np.pad(obz, npad, mode='constant', constant_values=0)

obx = tomopy.downsample(obx, level=2, axis=0)
obx = tomopy.downsample(obx, level=2, axis=1)
obx = tomopy.downsample(obx, level=2, axis=2)
oby = tomopy.downsample(oby, level=2, axis=0)
oby = tomopy.downsample(oby, level=2, axis=1)
oby = tomopy.downsample(oby, level=2, axis=2)
obz = tomopy.downsample(obz, level=2, axis=0)
obz = tomopy.downsample(obz, level=2, axis=1)
obz = tomopy.downsample(obz, level=2, axis=2)

Define the projection angles: 31 angles, from 90 to 270 degrees:

ang = tomopy.angles(31, 90, 270)

And calculate the projections of the object taking rotation axes around the three perpendicular cartesian axes:

prj1 = tomopy.project3(obx, oby, obz, ang, axis=0, pad=False)
prj2 = tomopy.project3(obx, oby, obz, ang, axis=1, pad=False)
prj3 = tomopy.project3(obx, oby, obz, ang, axis=2, pad=False)

The three coordinates of a given projection can be visualized as follows:

fig = plt.figure(figsize=(15, 8))
ax1 = fig.add_subplot(1, 3, 1)
ax1.imshow(obx[52,:,:])
ax2 = fig.add_subplot(1, 3, 2)
ax2.imshow(oby[52,:,:])
ax3 = fig.add_subplot(1, 3, 3)
ax3.imshow(obz[52,:,:])
Finally we will reconstruct the vector field components, taking as input the projections that we have calculated thanks to the first 3D initial object. The number of iterations can be adapted to have a faster but more imprecise reconstruction, or to have a more precise reconstruction.

```python
rec1, rec2, rec3 = tomopy.vector3(prj1, prj2, prj3, ang, ang, ang, axis1=0, axis2=1, axis3=2, num_iter=100)
dxchange.write_tiff(rec1)
dxchange.write_tiff(rec2)
dxchange.write_tiff(rec3)
```

**Comparison of results against input object**

In this section, we compare the results of the vector field components obtained thanks to the tomopy reconstruction, against the vector field components of the object given as input:

Comparison of the first magnetization vector component against the input data object (for a given slice).

```python
fig = plt.figure(figsize=(9, 7))
ax1 = fig.add_subplot(1, 2, 1)
ax1.imshow(obx[52, :, :])
ax2 = fig.add_subplot(1, 2, 2)
ax2.imshow(rec1[52, :, :])
```
Comparison of the second magnetization vector component against the input data object (for a given slice):

```python
fig = plt.figure(figsize=(9, 7))
ax1 = fig.add_subplot(1, 2, 1)
ax1.imshow(oby[52,:,:])
ax2 = fig.add_subplot(1, 2, 2)
ax2.imshow(rec2[52,:,:])
```

Comparison of the third magnetization vector component against the input data object (for a given slice):

```python
fig = plt.figure(figsize=(9, 7))
ax1 = fig.add_subplot(1, 2, 1)
ax1.imshow(obz[52,:,:])
ax2 = fig.add_subplot(1, 2, 2)
ax2.imshow(rec3[52,:,:])
```
Other examples

Three jupyter notebooks with examples as well as with some mathematical concepts related to the vector reconstruction, can be found in the tomopy/doc/demo folder:

Examples using vector3: input data projections from 3 orthogonal tilt angles:
  - vectorrec_1.ipynb
  - vectorrec_disk.ipynb

Example using vector2: input data projections from 2 orthogonal tilt angles:
  - vector_heterostructure.ipynb

The Vector Reconstruction examples html slides can be build by applying (from the doc/demo folder) the following commands:

```
jupyter-nbconvert --to slides --post serve vectorrec_1.ipynb
jupyter-nbconvert --to slides --post serve vectorrec_disk.ipynb
jupyter-nbconvert --to slides --post serve vector_heterostructure.ipynb
```

### 5.8.5 LPrec

Here is an example on how to use the log-polar based method (https://github.com/math-vrn/lprec) for reconstruction in Tomopy

```
%pylab inline
```

Install lprec from github, then

```
import tomopy
```

DXchange is installed with tomopy to provide support for tomographic data loading. Various data format from all major synchrotron facilities are supported.

```
import dxchange
```
matplotlib provide plotting of the result in this notebook. Paraview or other tools are available for more sophisticated 3D rendering.

```python
import matplotlib.pyplot as plt
```

Set the path to the micro-CT data to reconstruct.

```python
fname = '../../tomopy/data/tooth.h5'
```

Select the sinogram range to reconstruct.

```python
start = 0
end = 2
```

This data set file format follows the APS beamline 2-BM and 32-ID definition. Other file format readers are available at DXchange.

```python
proj, flat, dark, theta = dxchange.read_aps_32id(fname, sino=(start, end))
```

Plot the sinogram:

```python
plt.imshow(proj[:, 0, :], cmap='Greys_r')
plt.show()
```

If the angular information is not available from the raw data you need to set the data collection angles. In this case theta is set as equally spaced between 0-180 degrees.

```python
theta = tomopy.angles(proj.shape[0])
```

Perform the flat-field correction of raw data:

\[
\frac{proj - dark}{flat - dark}
\]

```python
proj = tomopy.normalize(proj, flat, dark)
```

Select the rotation center manually

```python
rot_center = 296
```

Calculate

\[-log(proj)\]

```python
proj = tomopy.minus_log(proj)
proj[proj<0] = 0
```
Reconstruction using FBP method with the log-polar coordinates

$$\hat{f} = \mathcal{W}R^*g$$

```python
recon = tomopy.recon(proj, theta, center=rot_center, algorithm=tomopy.lprec, lpmethod='fbp', filter_name='parzen')
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```

Reconstruction using the gradient descent method with the log-polar coordinates

$$\hat{f} = \arg\min_f \| \mathcal{R}f - g \|_2^2$$

```python
recon = tomopy.recon(proj, theta, center=rot_center, algorithm=tomopy.lprec, lpmethod='grad', ncore=1, num_iter=64, reg_par=-1)
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```

Reconstruction using the conjugate gradient method with the log-polar coordinates

$$\hat{f} = \arg\min_f \| \mathcal{R}f - g \|_2^2$$

```python
recon = tomopy.recon(proj, theta, center=rot_center, algorithm=tomopy.lprec, lpmethod='cg', ncore=1, num_iter=64, reg_par=-1)
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```
TomoPy Documentation

```python
recon = tomopy.recon(proj, theta, center=rot_center, algorithm=tomopy.lprec, lpmethod='cg', ncore=1, num_iter=16, reg_par=-1)
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```

Reconstruction using the TV method with the log-polar coordinates. It gives piecewise constant reconstructions and can be used for denoising.

\[
\hat{f} = \arg\min_f \|\mathcal{R} f - g\|^2 + \lambda \|\nabla f\|_1
\]

```python
recon = tomopy.recon(proj, theta, center=rot_center, algorithm=tomopy.lprec, lpmethod='tv', ncore=1, num_iter=512, reg_par=5e-4)
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```

Reconstruction using the TV-entropy method with the log-polar coordinates. It can be used for suppressing Poisson noise.

\[
\hat{f} = \arg\min_f \lambda \|\nabla f\|_1 + \int_\Omega \mathcal{R} f - g \log(\mathcal{R} f) df
\]
```python
recon = tomopy.recon(proj, theta, center=rot_center, algorithm=tomopy.lprec, lpmethod='tve', ncore=1, num_iter=512, reg_par=2e-4)
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```

Reconstruction using the TV-l1 method with the log-polar coordinates. It can be used to remove structures of an image of a certain scale, and the regularization parameter \( \lambda \) can be used for scale selection.

\[
\hat{f} = \arg\min_f \| \mathcal{R} f - g \|_1 + \lambda \| \nabla f \|_1
\]

```python
recon = tomopy.recon(proj, theta, center=rot_center, algorithm=tomopy.lprec, lpmethod='tvl1', ncore=1, num_iter=512, reg_par=3e-2)
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```

Reconstruction using the MLEM method with the log-polar coordinates.
```python
recon = tomopy.recon(proj, theta, center=rot_center, algorithm=tomopy.lprec, lmethod='em', ncore=1, num_iter=64, reg_par=0.05)
recon = tomopy.circ_mask(recon, axis=0, ratio=0.95)
plt.imshow(recon[0, :, :], cmap='Greys_r')
plt.show()
```

## 5.9 Frequently asked questions

Here’s a list of questions.

### Questions

- **How can I report bugs?**
- **Are there any video tutorials?**
- **Can I run this on a HPC cluster?**
- **Are there any segmentation routines?**
- **Are there any tools for aligning projections?**
- **What is ASTRA toolbox?**
- **Why TomoPy and ASTRA were integrated?**
- **Which platforms are supported?**
- **Does ASTRA support all GPUs?**
- **What is UFO?**

### 5.9.1 How can I report bugs?

The easiest way to report bugs or get help is to open an issue on GitHub. Simply go to the project GitHub page, click on Issues in the right menu tab and submit your report or question.
5.9.2 Are there any video tutorials?

We currently do not have specific plans in this direction, but we agree that it would be very helpful.

5.9.3 Can I run this on a HPC cluster?

In their default installation packages, TomoPy and the ASTRA toolbox are limited to running on a multicore single machine. The ASTRA toolbox, and TomoPy through the presented ASTRA integration, are able to use multiple GPUs that are installed in a single machine. Both toolboxes can be run on a HPC cluster through parallelization using MPI, but since installation and running on a HPC cluster is often cluster specific, the default installation packages do not include these capabilities.

As such, the integrated packages that is presented in the manuscript currently does not support running on a HPC cluster. Note that the ASTRA toolbox provides a separate MPI enabled package for use on a HPC cluster. We refer to [C24] for more details about TomoPy’s planned HPC implementation. It is a MapReduce type MPI implementation layer, which was successfully used on many clusters, i.e. Stampede, Cori, Mira. There are plans to allow user access to TomoPy on a HPC cluster (e.g. through a client or webportal), but these projects will take some time before they are being matured for user’s use.

5.9.4 Are there any segmentation routines?

Some data processing operations can be applied after reconstruction. Examples of these type of operations are image based ring removal methods, and gaussian filtering or median filtering the reconstructed image. Typically, these methods are called “postprocessing algorithms, since they occur after the reconstruction.

The package does not include segmentation algorithms, since we are currently focused on tomography, while we feel that segmentation are more part of the application specific data analysis that occurs after tomographic processing. An important exception is when segmentation steps are used as part of the tomographic reconstruction algorithm, such as in the DART algorithm.

5.9.5 Are there any tools for aligning projections?

Yes we have. Please check the Examples section for details.

5.9.6 What is ASTRA toolbox?

The ASTRA toolbox provides highly efficient tomographic reconstruction methods by implementing them on graphic processing units (GPUs). It includes advanced iterative methods and allows for very flexible scanning geometries. The ASTRA toolbox also includes building blocks which can be used to develop new reconstruction methods, allowing for easy and efficient implementation and modification of advanced reconstruction methods. However, the toolbox is only focused on reconstruction, and does not include pre-processing or post-processing methods that are typically required for correctly processing synchrotron data. Furthermore, no routines to read data from disk are provided by the toolbox.

5.9.7 Why TomoPy and ASTRA were integrated?

The TomoPy toolbox is specifically designed to be easy to use and deploy at a synchrotron facility beamline. It supports reading many common synchrotron data formats from disk [C13], and includes several other processing algorithms commonly used for synchrotron data. TomoPy also includes several reconstruction algorithms, which can be run on multi-core workstations and large-scale computing facilities. The algorithms in TomoPy are all CPU-based, however,
which can make them prohibitively slow in the case of iterative methods, which are often required for advanced
tomographic experiments.

By integrating the ASTRA toolbox in the TomoPy framework, the optimized GPU-based reconstruction methods
become easily available for synchrotron beamline users, and users of the ASTRA toolbox can more easily read data
and use TomoPy’s other functionality for data filtering and cleaning.

5.9.8 Which platforms are supported?

TomoPy supports Linux and Mac OS X, and the ASTRA toolbox supports Linux and Windows. As such, the combined
package currently supports only Linux, but we are working on supporting more operating systems.

5.9.9 Does ASTRA support all GPUs?

The GPU algorithms are all implemented used nVidia CUDA. As a result, only nVidia CUDA enabled video cards can
be used to run them.

5.9.10 What is UFO?

UFO is a general purpose image processing framework, optimized for heterogeneous compute systems and streams
of data. Arbitrary data processing tasks are plugged together to form larger processing pipelines. These pipelines are
then mapped to the hardware resources available at run-time, i.e. both multiple GPUs and CPUs.

One specific use case that has been integrated into the TomoPy is fast reconstruction using the filtered backprojection
and direct Fourier inversion methods although others for pre- and post-processing might be added in the future.

5.10 Credits

We kindly request that you cite the following article(s) [A1] if you use TomoPy (and also cite [A2] if you use ASTRA
or [A3] if you use UFO). For vector reconstructions please additionally cite [A4].

5.10.1 Applications

5.10.2 References


Python Module Index

t

tomopy, 71
tomopy.misc.corr, 27
tomopy.misc.morph, 31
tomopy.misc.phantom, 33
tomopy.prep.alignment, 35
tomopy.prep.normalize, 40
tomopy.prep.phase, 41
tomopy.prep.stripe, 42
tomopy.recon.algorithm, 45
tomopy.recon.rotation, 47
tomopy.sim.project, 50
tomopy.sim.propagate, 53
Index

A
add_focal_spot_blur() (in module tomopy.sim.project), 53
add_gaussian() (in module tomopy.sim.project), 52
add_jitter() (in module tomopy.prep.alignment), 37
add_noise() (in module tomopy.prep.alignment), 38
add_salt_pepper() (in module tomopy.sim.project), 53
adjust_range() (in module tomopy.misc.corr), 28
align_joint() (in module tomopy.prep.alignment), 36
align_seq() (in module tomopy.prep.alignment), 35
angles() (in module tomopy.sim.project), 51

B
baboon() (in module tomopy.misc.phantom), 33
barbara() (in module tomopy.misc.phantom), 34
blur_edges() (in module tomopy.prep.alignment), 38

C
calc_intensity() (in module tomopy.sim.propagate), 53
calc_slit_box_aps_lid() (in module tomopy.prep.alignment), 39
cameraman() (in module tomopy.misc.phantom), 34
checkerboard() (in module tomopy.misc.phantom), 34
circ_mask() (in module tomopy.misc.corr), 28

distortion_correction_proj() (in module tomopy.prep.alignment), 39
distortion_correction_sino() (in module tomopy.prep.alignment), 39
downsampal() (in module tomopy.misc.morph), 32

E
enhance_projs_aps_lid() (in module tomopy.misc.corr), 31

F
fan_to_para() (in module tomopy.sim.project), 52
find_center() (in module tomopy.recon.rotation), 48
find_center_pc() (in module tomopy.recon.rotation), 48
find_center_vo() (in module tomopy.recon.rotation), 48
find_slits_corners_aps_lid() (in module tomopy.prep.alignment), 38

gaussian_filter() (in module tomopy.misc.corr), 28

I
init_tomo() (in module tomopy.recon.algorithm), 47

L
lena() (in module tomopy.misc.phantom), 34
load_distortion_coefs() (in module tomopy.prep.alignment), 39

M
mask_empty_slice() (in module tomopy.recon.rotation), 50
median_filter() (in module tomopy.misc.corr), 28
median_filter_cuda() (in module tomopy.misc.corr), 29
minus_log() (in module tomopy.prep.normalize), 40

N
normalize() (in module tomopy.prep.normalize), 40
normalize_bg() (in module tomopy.prep.normalize), 40
normalize_nf() (in module tomopy.prep.normalize), 41
normalize_roi() (in module tomopy.prep.normalize), 41
TomoPy Documentation

**P**
- `pad()` (in module `tomopy.misc.morph`), 32
- `para_to_fan()` (in module `tomopy.sim.project`), 52
- `peppers()` (in module `tomopy.misc.phantom`), 34
- `phantom()` (in module `tomopy.misc.phantom`), 35
- `probe_gauss()` (in module `tomopy.sim.propagate`), 54
- `project()` (in module `tomopy.sim.project`), 51
- `project2()` (in module `tomopy.sim.project`), 51
- `project3()` (in module `tomopy.sim.project`), 51
- `propagate_tie()` (in module `tomopy.sim.propagate`), 54

**R**
- `recon()` (in module `tomopy.recon.algorithm`), 45
- `remove_all_stripe()` (in module `tomopy.prep.stripe`), 44
- `remove_dead_stripe()` (in module `tomopy.prep.stripe`), 44
- `remove_large_stripe()` (in module `tomopy.prep.stripe`), 44
- `remove_nan()` (in module `tomopy.misc.corr`), 29
- `remove_neg()` (in module `tomopy.misc.corr`), 29
- `remove_outlier()` (in module `tomopy.misc.corr`), 30
- `remove_outlier1d()` (in module `tomopy.misc.corr`), 30
- `remove_outlier_cuda()` (in module `tomopy.misc.corr`), 30
- `remove_ring()` (in module `tomopy.misc.corr`), 31
- `remove_slits_aps_lid()` (in module `tomopy.alignment`), 39
- `remove_stripe_based_filtering()` (in module `tomopy.prep.stripe`), 43
- `remove_stripe_based_fitting()` (in module `tomopy.prep.stripe`), 44
- `remove_stripe_based_sorting()` (in module `tomopy.prep.stripe`), 43
- `remove_stripe_fw()` (in module `tomopy.prep.stripe`), 42
- `remove_stripe_sf()` (in module `tomopy.prep.stripe`), 43
- `remove_stripe_ti()` (in module `tomopy.prep.stripe`), 43
- `retrieve_phase()` (in module `tomopy.prep.phase`), 42

**S**
- `scale()` (in module `tomopy.prep.alignment`), 37
- `shepp2d()` (in module `tomopy.misc.phantom`), 34
- `shepp3d()` (in module `tomopy.misc.phantom`), 35
- `shift_images()` (in module `tomopy.prep.alignment`), 38
- `sino_360_to_180()` (in module `tomopy.misc.morph`), 32
- `sobel_filter()` (in module `tomopy.misc.corr`), 29

**T**
- `tilt()` (in module `tomopy.prep.alignment`), 37
- `tomopy` (module), 54, 71
- `tomopy.misc.corr` (module), 27
- `tomopy.misc.morph` (module), 31
- `tomopy.misc.phantom` (module), 33
- `tomopy.prep.alignment` (module), 35
- `tomopy.prep.normalize` (module), 40
- `tomopy.prep.phase` (module), 41
- `tomopy.prep.stripe` (module), 42
- `tomopy.recon.algorithm` (module), 45
- `tomopy.recon.rotation` (module), 47
- `tomopy.sim.project` (module), 50
- `tomopy.sim.propagate` (module), 53
- `trim_sinogram()` (in module `tomopy.misc.morph`), 33

**U**
- `upsample()` (in module `tomopy.misc.morph`), 32

**W**
- `write_center()` (in module `tomopy.recon.rotation`), 49

`sino_360_to_180()` (in module `tomopy.misc.morph`), 32
`sobel_filter()` (in module `tomopy.misc.corr`), 29