Algotom Documentation

NSLS-II, Brookhaven National Lab, US; Diamond Light Source, UK

Mar 27, 2024

CONTENTS

Welcome to Algotom's documentation about data processing algorithms for tomography. This documentation is not only to explain functions available in the Algotom package but also to present tomography-related tutorials, technical notes, and applications.

Source code: https://github.com/algotom/algotom

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Basic tutorials

1.1.1 Python for tomography scientists as beginners

It is common that well-made software cannot provide all the tools for scientists to perform their analysis. In such cases, knowing how to program becomes crucial. There are many open-source programming languages to choose, in which Python and its rich ecosystem are dominantly used in the science community for its ease-of-use. This section dedicates to whom would like to write Python codes to process their data but don't know where to start. There are many ways/resources to install/learn Python, however, the section focuses to present approaches which are easy-to-follow and practical.

Installing Python and tools for writing codes

To start, users need to install two software: one is Python and one is for writing codes, known as IDE (Integrated Development Environment) software. The second one is optional but it's important for coding and debugging efficiently. Python can be downloaded and installed through Anaconda which not only distributes Python and its ecosystem but also Conda, a package management software, to install Python libraries with easy. These open-source libraries, contributed by the developer community, are the main reason for the popularity of Python.

After installing Anaconda, users can run Anaconda Powershell Prompt (e.g. on WinOS) to manage and install Python packages (i.e. libraries). A collection of Python packages installed is known as an environment. An environment created by a package manager (e.g. Conda) helps to deal with the conflict of Python packages using different versions of dependencies. This link is useful for whom want to know more about Python environment. There is a list of popular Python libraries shipped with Anaconda, known as the *base* environment. To install Python packages out of the list, it's a good practice that users should create a separate environment from the base. Instructions of how to create a new environment and how to install new packages are here and here.

The next step is to install an IDE software for writing codes. There are many free choices: Pycharm (Community edition), Pydev, Spyder, or VS Code. Here, we recommend to use Pycharm because it is charming as the name suggested. After installing Pycharm, users have to configure the software to link to a Python interpreter by pointing to the location of Python packages installed (Fig. 1.1.2).

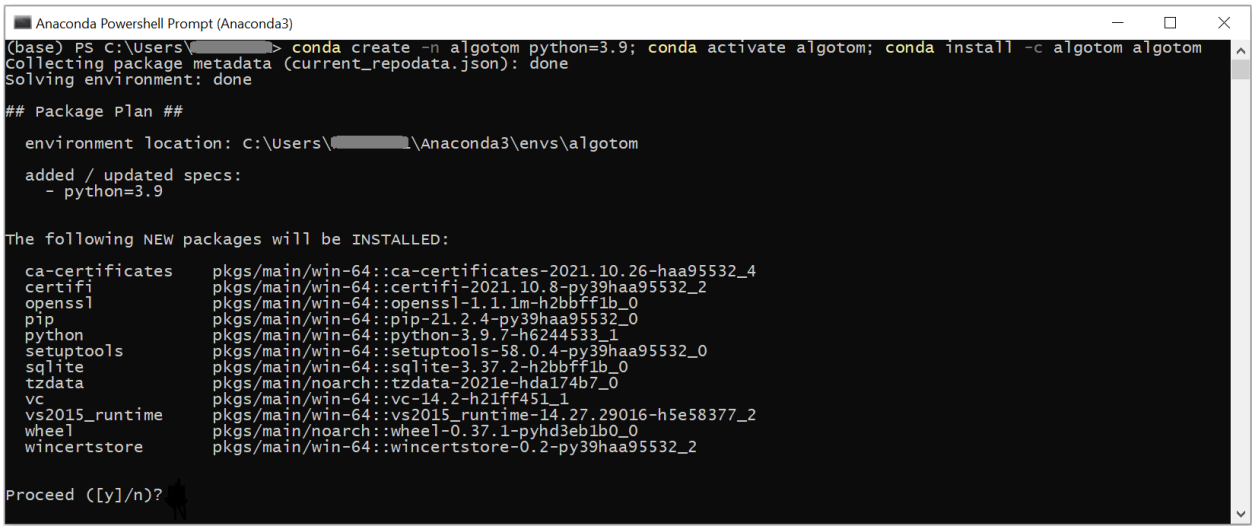


Fig. 1.1.1: Combination of conda commands to: create an environment named algotom, install Python 3.9, activate the environment, then install the algotom package from the algotom channel.

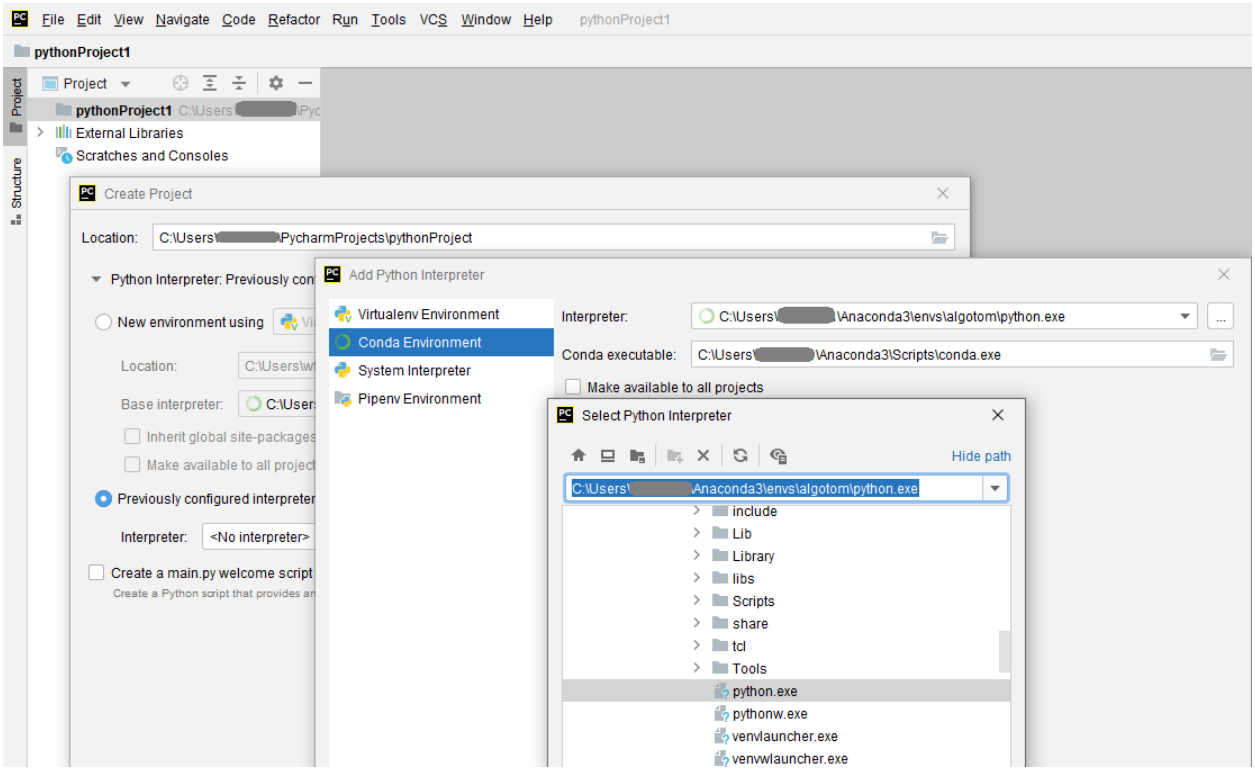


Fig. 1.1.2: Demonstration of how to configure Pycharm to link to a Python environment.

Python ecosystem of libraries

The power and popularity of Python come from its enormous ecosystem. Crucially, supporting tools such as Conda, Pip, and Github make it very easy for developers/users to develop/install Python libraries. Nowadays, imaging scientists can use Python libraries to perform almost every task in the workflow from data acquisition, data processing, data analysis, to data visualization. Python libraries can be classified into a few types. The first one is the standard library, i.e. the built-in packages. They are shipped with Python. The second type is well-developed and popular libraries maintained by dedicated software development teams. List of such libraries can be found in this link which are shipped with Anaconda software. The third type are libraries developed by organizations, academic institutions, or research groups dedicated to specific technical/scientific areas. The last type of libraries is contributed by individuals who would like to share their works.

A Python package is commonly built based on other Python libraries known as *dependencies*. This can cause conflicts between libraries using different versions of the same libraries. In such cases, a package manager like Conda is crucially needed. Python libraries are distributed through https://anaconda.org/ and https://pypi.org/. Users can search packages they need in these websites where instructions of how to install these packages using the *conda* or *pip* command are shown on the page of each package. Lots of Python packages are distributed on both platforms. However, there are packages only available in one platform. Fortunately, Conda allows to use *pip* to install packages as well. Users are recommended to check this tutorial to know more about the difference between *conda* and *pip*.

The following list shows some Python packages which are useful for the tomography community. The selected packages are installable using *conda/pip* and work across OS (Windows, Linux, Mac).

- Numerical computing: Numpy, Scipy, Pyfftw, Pywavelets, ...
- Image processing: Scikit-image, Pillow, Discorpy, Opencv, ...
- Tomographic data processing: Tomopy, Astra Toolbox, Algotom, Cil, ...
- GPU computing: Numba, Cupy, ...
- Hdf file handling: H5py
- Data visualisation: Matplotlib, Vtk, ...
- Parallel processing: Joblib, Dask, ...

There are other Python software for processing tomographic data such as Savu, Tigre, tofu-ufo, or Pyhst2. However, they either don't work across OS or are not distributed with *conda/pip*.

Where/how to start coding

Python is the programming language that one can learn easily using the top-down approach instead of the bottom-up one which takes time. For example, one can start by asking questions such as: how to read an image, apply a smoothing filter, and save the result; then finding the answers using Google, Stackoverflow, or referring codes shared on Github. The following presents notes and tips about Python users may find useful before diving into coding.

- For quickly getting to know the syntax of the Python language, the python-course.eu website is a good place to start.
- In computational applications, we don't often use the standard library of Python but the Numpy library. Almost all of computational Python-libraries are built on top of Numpy. Although it is a backbone for the success of Python, Numpy is not included into the standard library of Python. Users have to install it separately, or they can just install a package which has Numpy as a dependency. The following codes show an example of how to find the sum of a list of float numbers using both approaches: the standard library and Numpy. A rule of thumb is to avoid using the standard library for computational works which use looping operations. Numpy provides most of basic tools, optimized for speed, to perform math operations on n-dimension arrays. Users can build complex applications on top of these tools.

```
import numpy as np

vals = [1.0, 3.0, 5.0, 7.0, 8.0]
# Using the standard lib
sum = 0.0
for i in vals:
    sum = sum + i
# Using Numpy
sum = np.sum(np.asarray(vals))
```

- Functions (known as methods) in each Python library is organized into folders, sub-folders (known as packages), then Python files (known as modules). Users can use functions needed by importing a whole package, specific sub-packages, specific modules, or specific methods.

```
import scipy  # Load the whole package
from scipy import ndimage as ndi # Import sub-package, give it an alias name.
import scipy.ndimage as ndi # Another way to import sub-packge.
import scipy.ndimage.filters as fil # Import a module, give it an alias name.
from scipy.ndimage.filters import gaussian_filter # Import a specific method in a␣
↪module.
```

Because Python libraries are a huge collection of functions, users better use the help of IDE software to find the right functions as demonstrated in Fig. 1.1.3. Using alias names for importing packages is a good practice to avoid the naming conflict, i.e. a user-defined function is named the same as a function in the library.

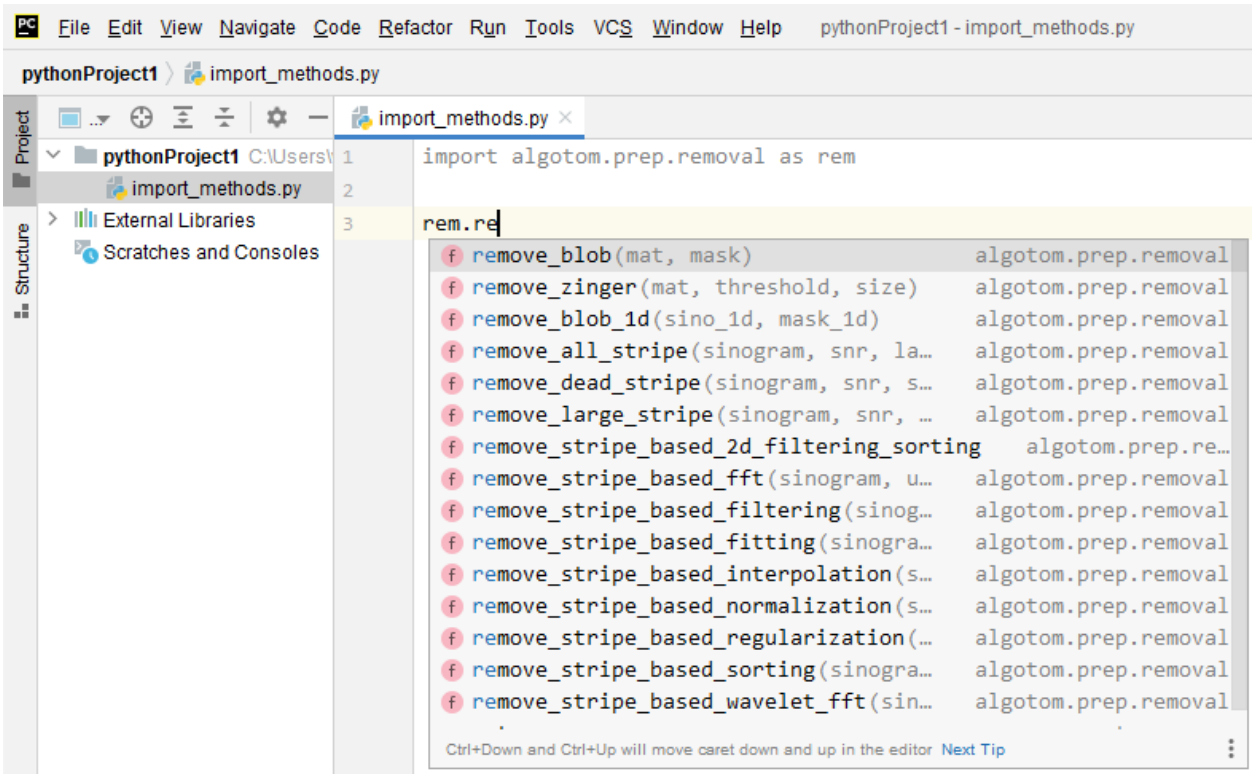


Fig. 1.1.3: Demonstration of how Pycharm can help to see a list of available functions.

- There are n-dimension array objects created by different Python libraries which look similar but their properties and uses are different. Users have to make sure that they use the right methods on the right objects.

```
import numpy as np
import dask.array as da
import cupy as cp

data = [[1.0, 2.0, 3.0], [3.0, 4.0, 5.0]] # Python-list object
data_np = np.asarray(data) # Numpy-array object
data_da = da.from_array(data_np) # Dask-array object
data_cp = cp.array(data) # Cupy-array object
```

- To use functions from Python packages in a script, users use the *import* command as shown above. When the command is executed, the Python interpreter automatically checks a few places to find such packages: paths in the system environment variables (e.g. WinOS: "Edit the system environment variables" -> "Environment variable"), paths in the current Python environment (e.g. WinOS: "C:Users<user_ID>Anaconda3envs<env_name>"), and the current location of the script. If the import fails, users need to check: if the package is installed (e.g. using *conda list* in an activated environment) and/or if the package is at the same path as the script.

  In Pycharm, if a package keeps failing to import , even though the package is installed and the Pycharm project is configured to the right Python environment, users can try one of the following ways:

  - Run *conda init*.
  - Run Pycharm from the activated environment (e.g Win OS: Powershell Prompt -> conda activate <env_name> -> pycharm)

  If users want to add the path to a package manually, they can do that as follows.

```
import sys
sys.path.insert(0, "C:/<Path-to-package>")
import <package-name>
```

  this is handy when users download a Python package somewhere and want to import its functions to the current script without installing the package. Note that Python libs (dependencies) used by the package need to be installed.

- Video tutorials are the best resources to learn new things quickly. There are many amazing tutorials on Youtube.com (free), Udemy.com (not free but at affordable price). They teach nearly everything about Python and its ecosystem. For tomography scientists, the youtube channel of Dr. Sreenivas Bhattiprolu is highly recommended. The uploaded tutorials accompanied by Python codes cover from basic topics of image processing to advanced topics such as image segmentation and deep-learning.

1.1.2 Common data format at synchrotron facilities

Two types of data format often used at most of synchrotron facilities are tiff and hdf. Hdf (Hierarchical Data Format) format allows to store multiple data-sets, multiple data-types in a single file. This solves a practical problem of collecting all data associated with an experiment such as images from a detector, stage positions, or furnace temperatures into one place for easy of management. More than that, hdf format allows to read/write subsets of data to memory/disk. This capability enables to process a large size dataset using a normal computer. Tiff format is used because it is supported by most of image-related software and it can store 32-bit grayscale values.

Hdf format

How to view the structure of a hdf file

To work with a hdf file, we need to know its structure or how to access its contents. This can be done using a lightweight software such as Hdfview (Fig. 1.1.4). Version 2.14 seems stable and is easy-to-install for WinOS. List of other hdf-viewer software can be found in this link. A wrapper of the hdf format known as the nexus format is commonly used at neutron, X-ray, and muon science facilities. We can use the same software and Python libraries to access both hdf and nxs files.

Another way to display a tree view of a hdf/nxs file is to use an Algotom's function as shown below.

There are many GUI software in Python for viewing hdf/nxs/h5 files such as: Broh5, Nexpy, or Vitables.

How to load datasets from a hdf file

Utilities for accessing a hdf/nxs file in Python are available in the h5py library. To load/read a dataset to a Python workspace, we need a key, or path, to that dataset in a hdf/nxs file.

```
import h5py

file_path = "E:/Tomo_data/68067.nxs" # https://doi.org/10.5281/zenodo.1443568
hdf_object = h5py.File(file_path, 'r')
key = "entry1/tomo_entry/data/data"
tomo_data = hdf_object[key]
print("Shape of tomo-data: {}".format(tomo_data.shape))
#>> Shape of tomo-data: (1861, 2160, 2560)
```

An important feature of a hdf format is that we can load subsets of data as demonstrated below.

```
import psutil

mem_start = psutil.Process().memory_info().rss / (1024 * 1024)
projection = tomo_data[100, :, :]
mem_stop = psutil.Process().memory_info().rss / (1024 * 1024)
print("Memory used for loading 1 projection : {} MB".format(mem_stop - mem_
↪start))
#>> Memory used for loading 1 projection : 11.3828125 MB

mem_start = psutil.Process().memory_info().rss / (1024 * 1024)
projections = tomo_data[102:104, :, :]
mem_stop = psutil.Process().memory_info().rss / (1024 * 1024)
print("Memory used for loading 2 projections : {} MB".format(mem_stop - mem_
↪start))
#>> Memory used for loading 2 projections : 21.09765625 MB
```

Using functions of h5py's library directly is quite inconvenient. Algotom's API provides wrappers for these functions to make them more easy-to-use. Users can load hdf files, find keys to datasets, or save data in the hdf format by using a single line of code.

```
import algotom.io.loadersaver as losa

file_path = "E:/Tomo_data/68067.nxs"
keys = losa.find_hdf_key(file_path, "data")[0] # Find keys having "data" in␣
↪the path.
print(keys)
```

(continues on next page)

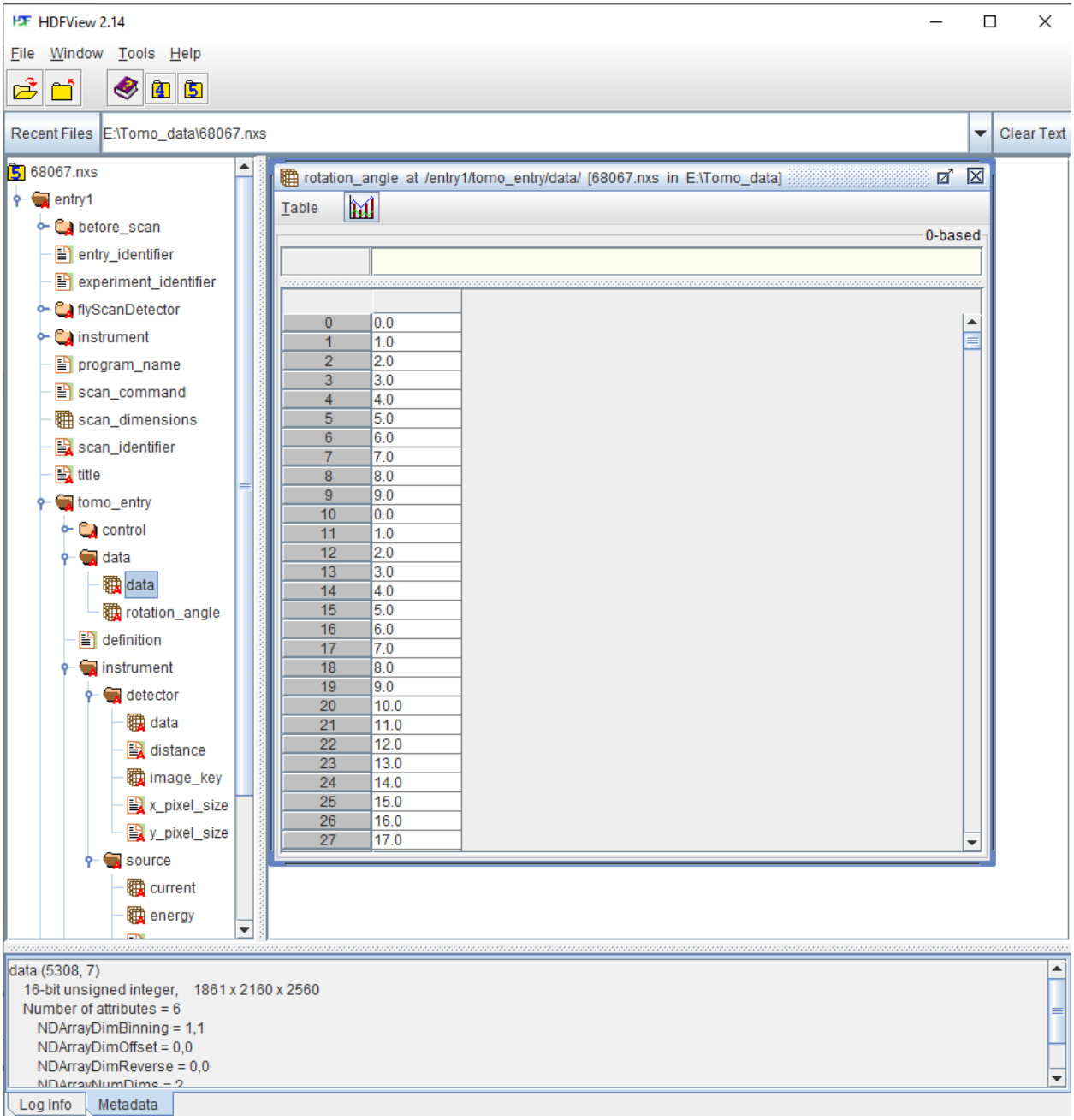


Fig. 1.1.4: Viewing the structure of a nxs/hdf file using the Hdfview software.

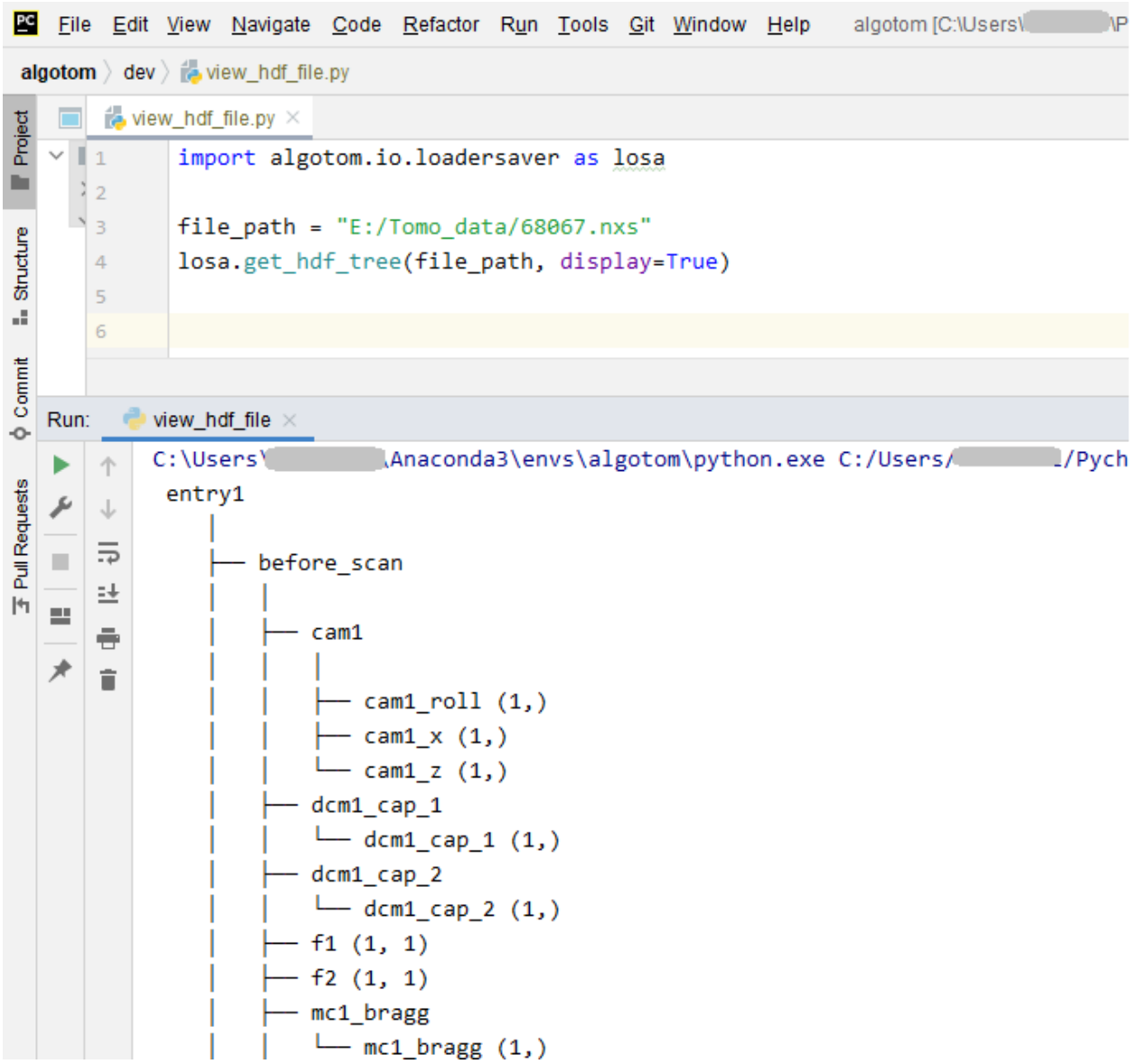


Fig. 1.1.5: Displaying the tree view of a nxs/hdf file using an Algotom's function.

(continued from previous page)

```
tomo_data = losa.load_hdf(file_path, keys[0]) # Load a dataset object
print(tomo_data.shape)
```

Notes on working with a hdf file

When working with multiple slices of a 3d data, it's faster to load them into memory chunk-by-chunk then process each slice, instead of loading and processing slice-by-slice. Demonstration is as follows.

```
import timeit
import scipy.ndimage as ndi
import algotom.io.loadersaver as losa

file_path = "E:/Tomo_data/68067.nxs"
tomo_data = losa.load_hdf(file_path, "entry1/tomo_entry/data/data")
chunk = 16

t_start = timeit.default_timer()
for i in range(1000, 1000 + chunk):
    mat = tomo_data[:, i, :]
    mat = ndi.gaussian_filter(mat, 11)
t_stop = timeit.default_timer()
print("Time cost if loading and processing each slice: {}".format(t_stop - t_
↪start))
#>> Time cost if loading and processing each slice: 10.1719189000000001

t_start = timeit.default_timer()
mat_chunk = tomo_data[:, 1000:1000 + chunk, :]  # Load 16 slices in one go.
for i in range(chunk):
    mat = mat_chunk[i]
    mat = ndi.gaussian_filter(mat, 11)
t_stop = timeit.default_timer()
print("Time cost if loading multiple-slices: {}".format(t_stop - t_start))
#>>Time cost if loading multiple-slices: 0.10050070000000133
```

Parallel loading datasets from a hdf file is possible. However, this feature may be not enabled for WinOS. When working with large datasets using a small RAM computer, we may have to write/read intermediate results to/from disk as hdf files. In such cases, it is worth to check tutorials on how to optimize hdf I/O performance.

Tiff format

This is a very popular file format and supported by most of image-related software. There are 8-bit, 16-bit, and 32-bit format. 8-bit format can store grayscale values as 8-bit unsigned integers (range of 0 to 255 = 2^8 - 1). 16-bit format can store unsigned integers in the range of 0 to 65535 (2^{16} - 1). 32-bit format is used to store 32-bit float data. Most of image viewer software can display a 8-bit or 16-bit, but not 32-bit tiff image. Users may see a black or white image if opening a 32-bit tiff image using common photo viewer software. In such cases, Imagej or Fiji software can be used.

Sometimes users may want to extract a 2D slice of 3D tomographic data and save the result as a tiff image for checking using ImageJ or photo viewer software. This can be done as shown below.

```
import algotom.io.loadersaver as losa

file_path = "E:/Tomo_data/68067.nxs"
```

(continues on next page)

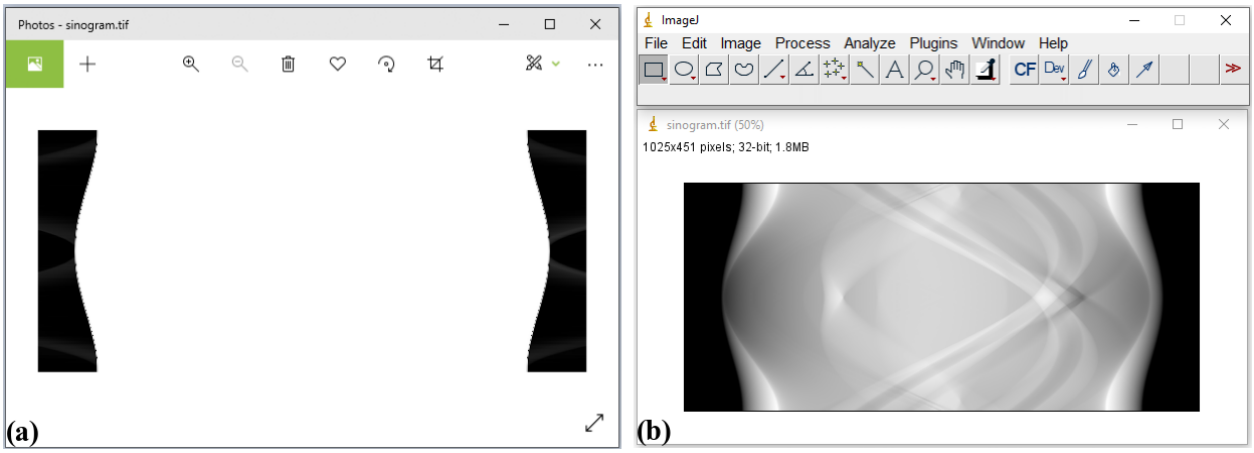


Fig. 1.1.6: Opening a 32-tiff image using Photos software (a) and Imagej software (b).

(continued from previous page)

```
tomo_data = losa.load_hdf(file_path, "entry1/tomo_entry/data/data")
losa.save_image("E:/Tomo_data/Output/proj.tif", tomo_data[100, :, :])
```

If tomographic data are acquired as a list of tiff files, it can be useful to convert them to a single hdf file first. This allows to extract subsets of the converted data for reconstructing a few slices or tweaking artifact removal methods before performing full reconstruction.

```
import numpy as np
import algotom.io.loadersaver as losa
import algotom.io.converter as conv

proj_path = "E:/Tomo_data/68067/projections/"
flat_path = "E:/Tomo_data/68067/flats/"
dark_path = "E:/Tomo_data/68067/darks/"
output_file = "E:/Tomo_data/68067/tomo_68067.hdf"

# Load flat images, average them.
flat_path = losa.find_file(flat_path + "/*.tif*")
height, width = np.shape(losa.load_image(flat_path[0]))
num_flat = len(flat_path)
flat = np.zeros((num_flat, height, width), dtype=np.float32)
for i in range(num_flat):
    flat[i] = losa.load_image(flat_path[i])
flat = np.mean(flat, axis=0)

# Load dark images, average them.
dark_path = losa.find_file(dark_path + "/*.tif*")
num_dark = len(dark_path)
dark = np.zeros((num_dark, height, width), dtype=np.float32)
for i in range(num_dark):
    dark[i] = losa.load_image(dark_path[i])
dark = np.mean(dark, axis=0)

# Generate angles
```

(continues on next page)

(continued from previous page)

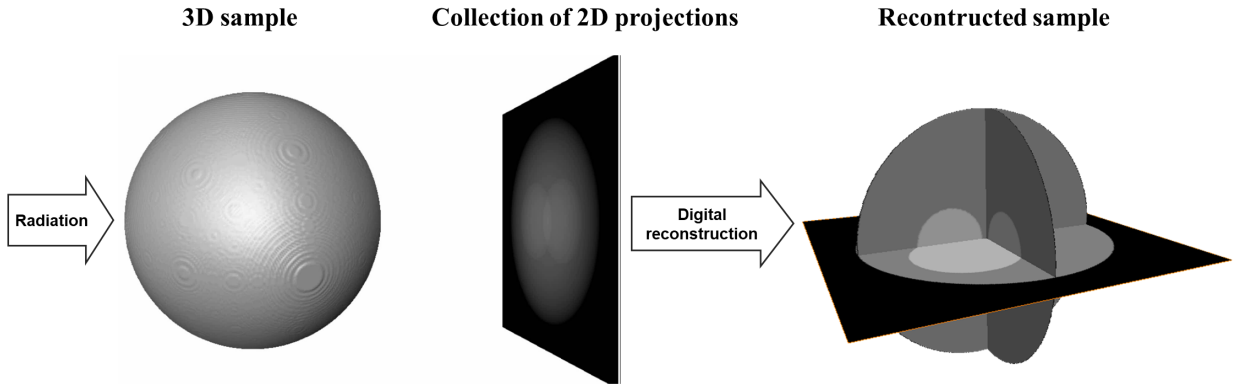
```
num_angle = len(losa.find_file(proj_path + "/*.tif*"))
angles = np.linspace(0.0, 180.0, num_angle)
# Save tiffs as a single hdf file.
conv.convert_tif_to_hdf(proj_path, output_file, key_path="entry/projection",
                        option={"entry/flat": np.float32(flat),
                                "entry/dark": np.float32(dark),
                                "entry/rotation_angle": np.float32(angles)})
```

Reconstructed slices from tomographic data are of 32-bit data, which often saved as 32-bit tiff images for easy to work with using analysis software such as Avizo, Dragon Fly, or Paraview. Some of these software may not support 32-bit tiff images or the 32-bit data volume is too big for computer memory. In such cases, we can rescale these images to 8-bit tiffs or 16-bit tiffs. It is important to be aware that rescaling causes information loss. The global extrema or user-chosen percentile of a 3D dataset or 4D dataset (time-series tomography) need to be used for rescaling to limit the loss. This functionality is available in Algotom as demonstrated below. Users can refer to Algotom's API to know how data are rescaled to lower bits.

```
import algotom.post.postprocessing as post

file_path = "E:/Tomo_data/recon_68067.hdf"
output_path = "E:/Tomo_data/rescale_8_bit/"
post.rescale_dataset(file_path, output_path, nbit=8, minmax=None)
```

1.1.3 Basic components of an X-ray tomography system

How tomography works



As demonstrated above, tomography is an imaging technique by which the internal 3D structure of a sample can be reconstructed from 2D projections formed by the penetration of radiation through the sample at a series of different angles in the range of [0; 180-degree]. If the radiation rays are parallel, the obtained 2D projections can be separated into independent 1D-projection rows. The sequence of these rows throughout the angular projection range forms a sinogram, i.e. a 2D data array corresponding to each individual row. Applying a reconstruction method on an individual sinogram yields a reconstructed 2D slice of the sample (Fig. 1.1.7). Combining all slices creates the 3D image of the sample.

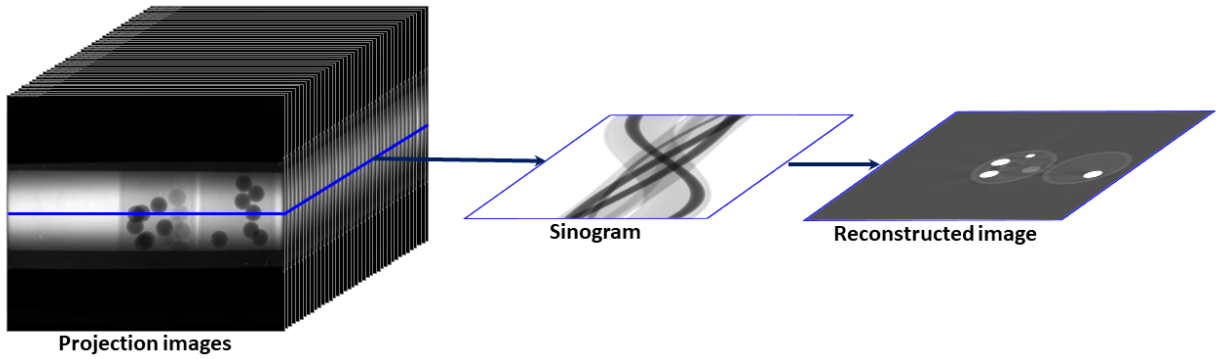


Fig. 1.1.7: Steps for reconstructing a slice in parallel-beam tomography.

Basic components of an X-ray tomography system

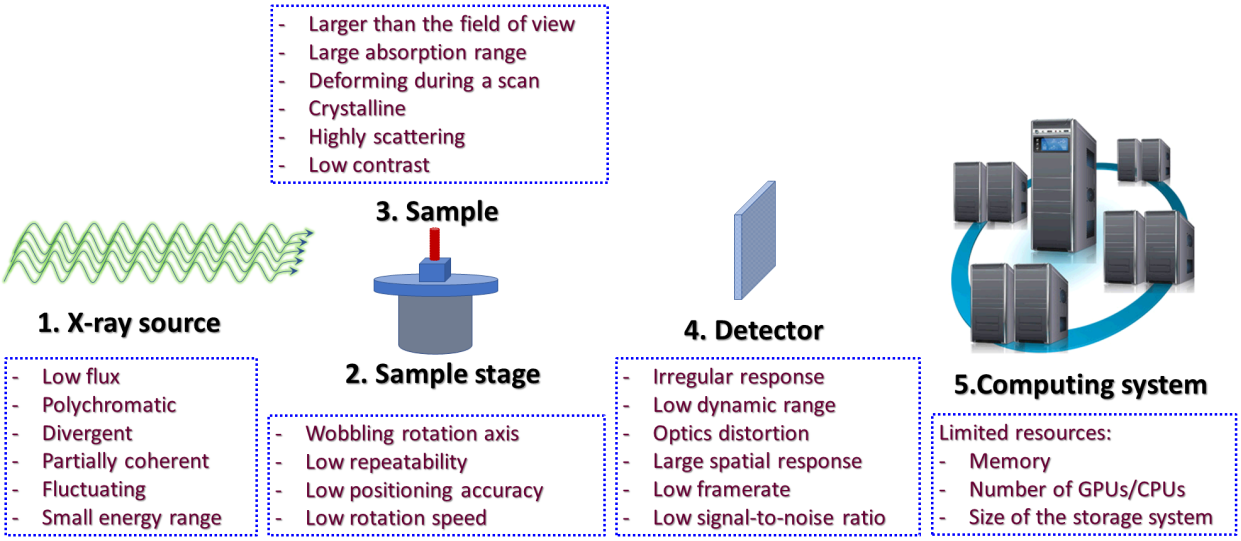


Fig. 1.1.8: Basic components of an X-ray tomography system and problems associated with them.

X-ray source

An ideal X-ray source for tomography experiments is monochromatic, stable, non-coherent, energy-tunable, high flux, and generates parallel beams. This allows to produce projections of a sample closest to the prediction of a mathematical model which is a necessary condition for reconstructing the sample with high quality. Unfortunately, there is no such source in practice. There are two basic ways of making X-ray sources: by hitting electrons to a target or by changing the direction of electrons moving at near-light speed. The first way is used in lab-based systems. The second way is used at synchrotron facilities.

Synchrotron-based X-ray sources are high-flux, monochromatic (by using a monochromator), energy-tunable, and close to the parallel-beam condition. However, their beams are partially coherent resulting in the interference between transmission beams and scattering beams after going through samples. This, known as the edge-enhanced effect, alters X-ray intensities at the interfaces between different materials of samples as can be seen in Fig. 1.1.9

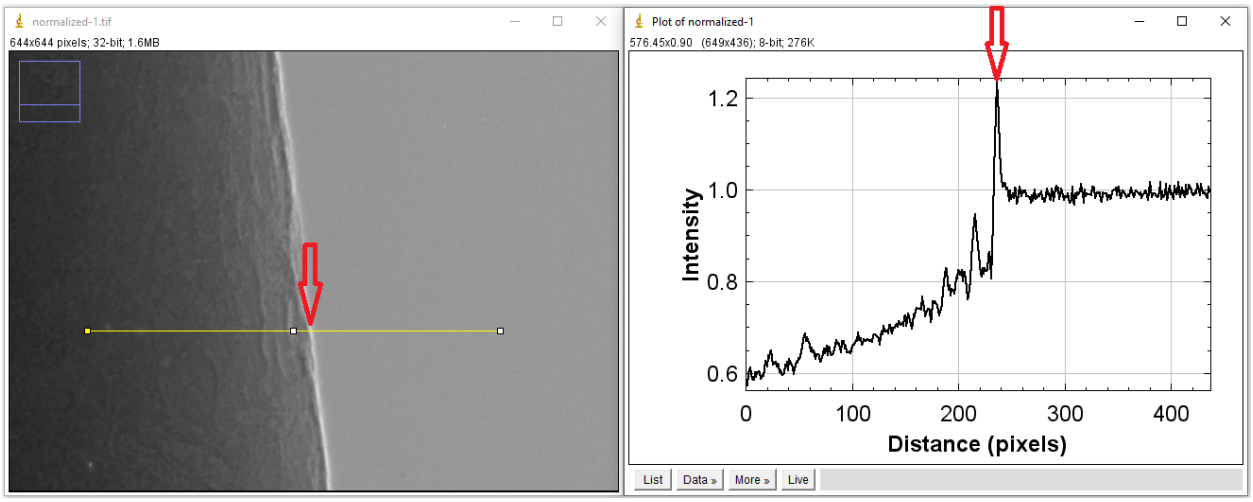


Fig. 1.1.9: Coherent source causes the edge-enhanced effect in a projection-image.

The edge-enhanced effect is useful for applications where the interfaces between materials are more important than their densities, such as studying crack formation in steels, rocks, or bones. However, this effect gives rise to streak artifacts and causes strong fluctuations of gray scales between interfaces in reconstructed images. These hampers the performance of post-processing methods such as image segmentation or image rescaling.

Other problems often seen at synchrotron-based sources come from high-heat-load monochromators. They can cause the fluctuation of source intensity or the shift of intensity profile. These problems impact the process of flat-field correction in tomography which results in artifacts.

Stage

In a micro-scale system, a major problem caused by the same-stage is the positioning repeatability of the rotation axis. For collecting tomographic data, we have to move a sample in-and-out the field of view to acquire images without the sample (known as flat-field/white-field images) and images with the sample (projection images). It's quite common that the rotation axis can be shifted a few pixels because of that. As a result, the center of rotation (COR) in the reconstruction space is changed (Fig. 1.1.12). This is inconvenient for the case that one collects multiple-datasets but can't use the same value of COR across.

In a nano-scale system, the main problem is the positioning accuracy of the stage. This causes the shift between projections of a tomographic dataset. To process such data, we have to apply image alignment/registration methods.

Sample

Samples can impact to the quality of reconstructed images as demonstrated in a few examples as follows

For samples with strong variation of absorption characteristic, i.e. flat samples, X-rays may not penetrate at some angles or detectors (mostly coupled to a 16-bit or 8-bit CCD chip) can not record such a large dynamic range of intensity. These impacts can result in different types of artifacts as shown in Fig. 1.1.14.

For crystalline samples, they can block X-rays at certain angles causing partially horizontal dark-stripes in sinograms. This can affect algebraic reconstruction-methods as shown in Fig. 1.1.15.

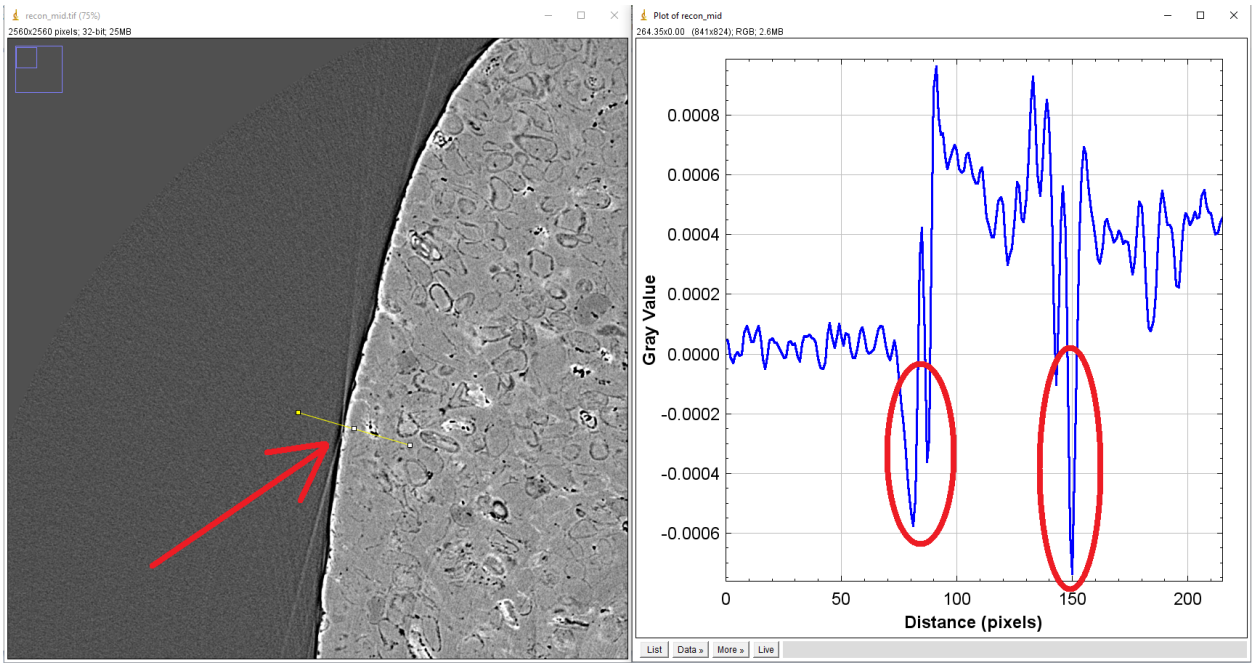


Fig. 1.1.10: Impacts of the edge-enhanced effect to a reconstructed image: streak artifacts (arrowed), negative attenuation coefficients (circled).

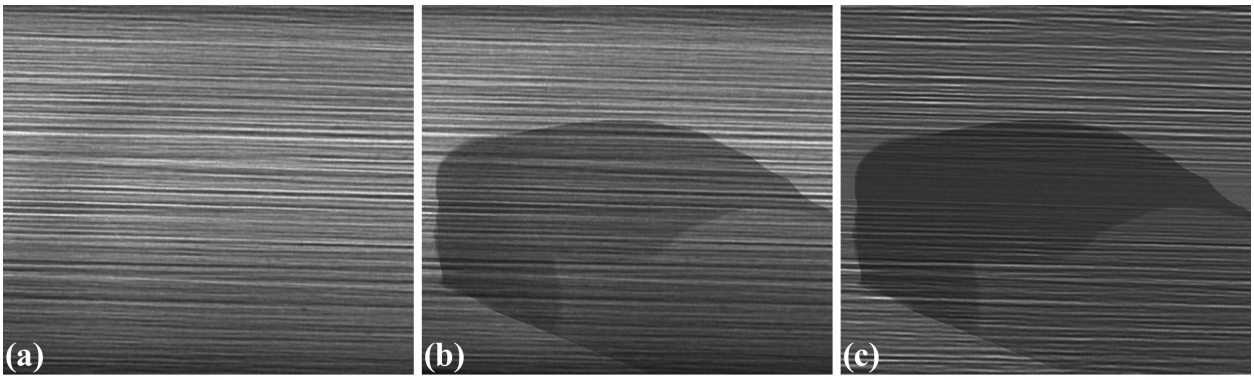


Fig. 1.1.11: Impacts of a monochromator to the intensity profile of a source. (a) Flat-field image. (b) Sample image. (c) Flat-field-corrected image

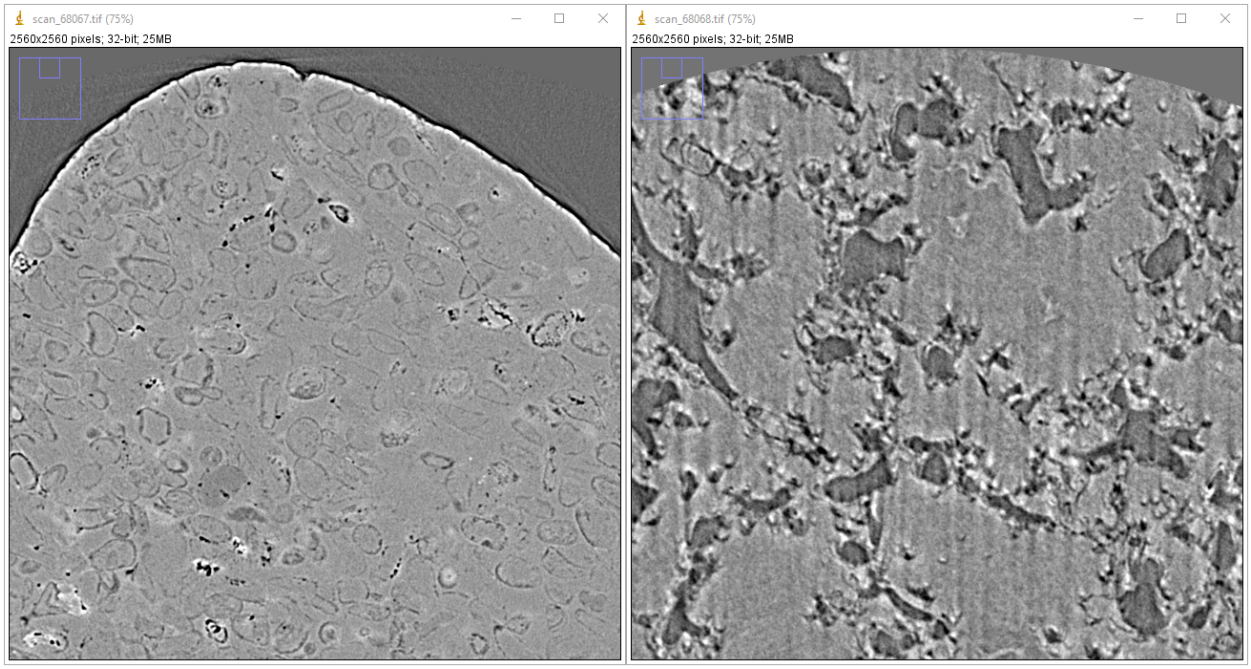


Fig. 1.1.12: Center of rotation was changed between two scans

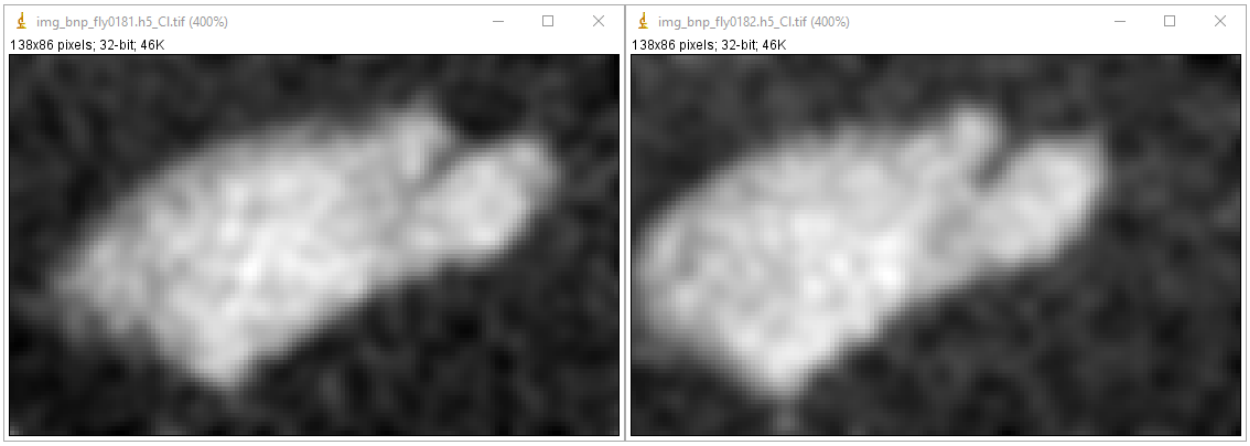


Fig. 1.1.13: Shift between two projections acquired by a nanoprobe X-ray fluorescence imaging system.

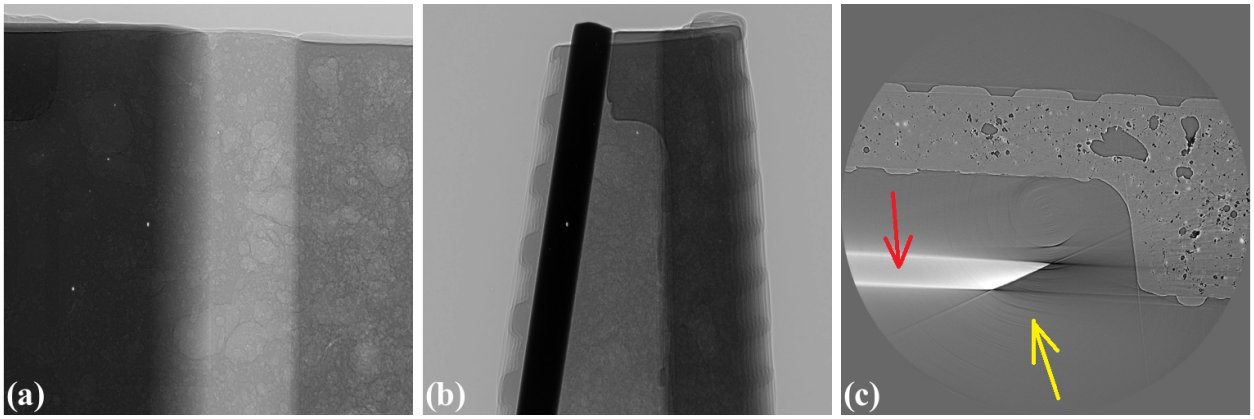


Fig. 1.1.14: Artifacts caused by a flat sample. (a) Projection at 0-degree. (b) Projection at 90-degree. (c) Reconstructed image with partial ring artifacts (yellow arrow) and cupping artifacts (red arrow).

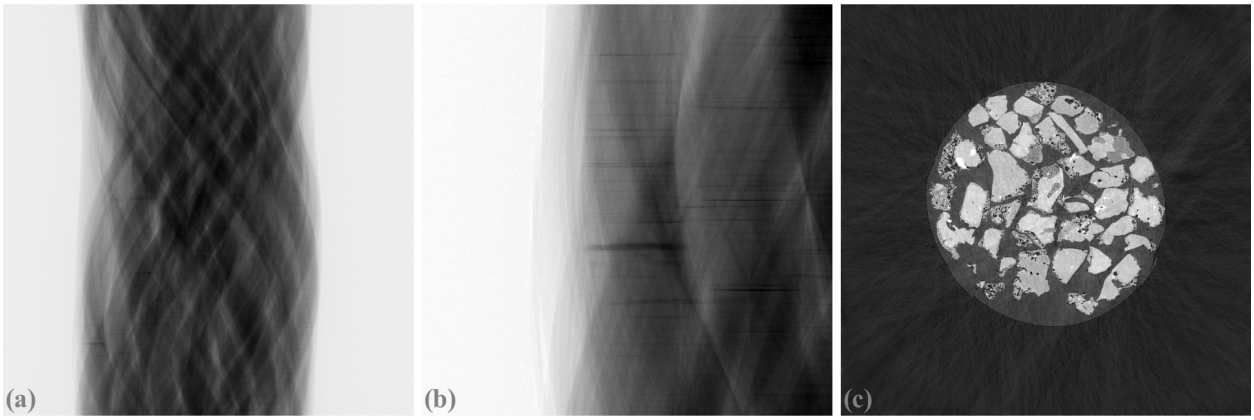


Fig. 1.1.15: Artifacts caused by a crystalline sample. (a) Sinogram. (b) Zoom-in at the bottom-left area of (a). (c) Reconstructed image using the SART method.

Scanning biological samples using hard X-rays can result in low-contrast images (Fig. 1.1.16). which affects the performance of post-processing methods such as segmentation or feature detection.

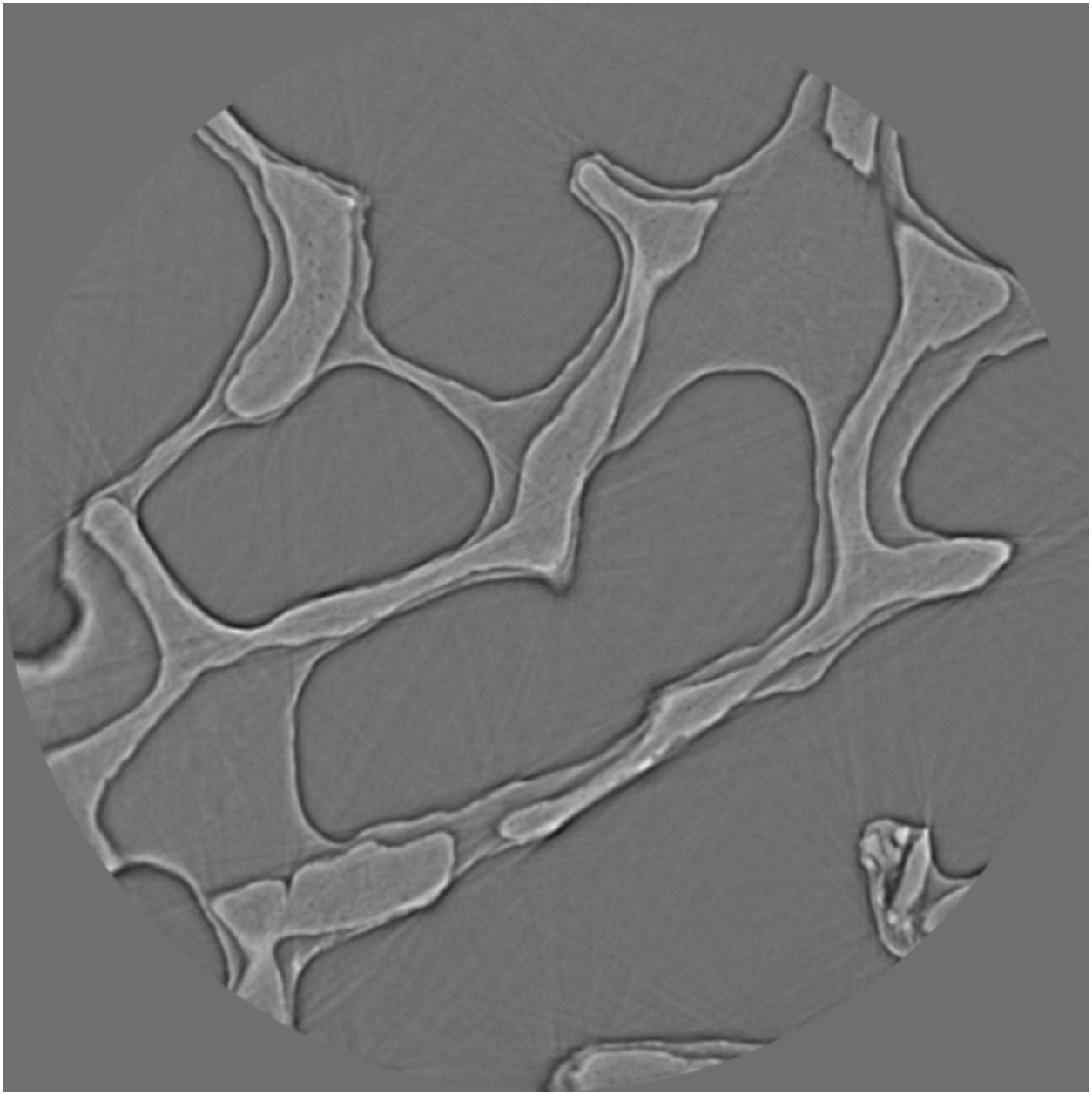


Fig. 1.1.16: Reconstructed image of a trabecular bone sample using a 53keV X-ray source.

Detector

Technical problems or limitations of a detecting system can cause various types of artifacts. The irregular response caused by defects in its hardware components, such as a scintillator or CCD chip, gives rise to ring artifacts as described in detail here and shown in Fig. 1.1.17

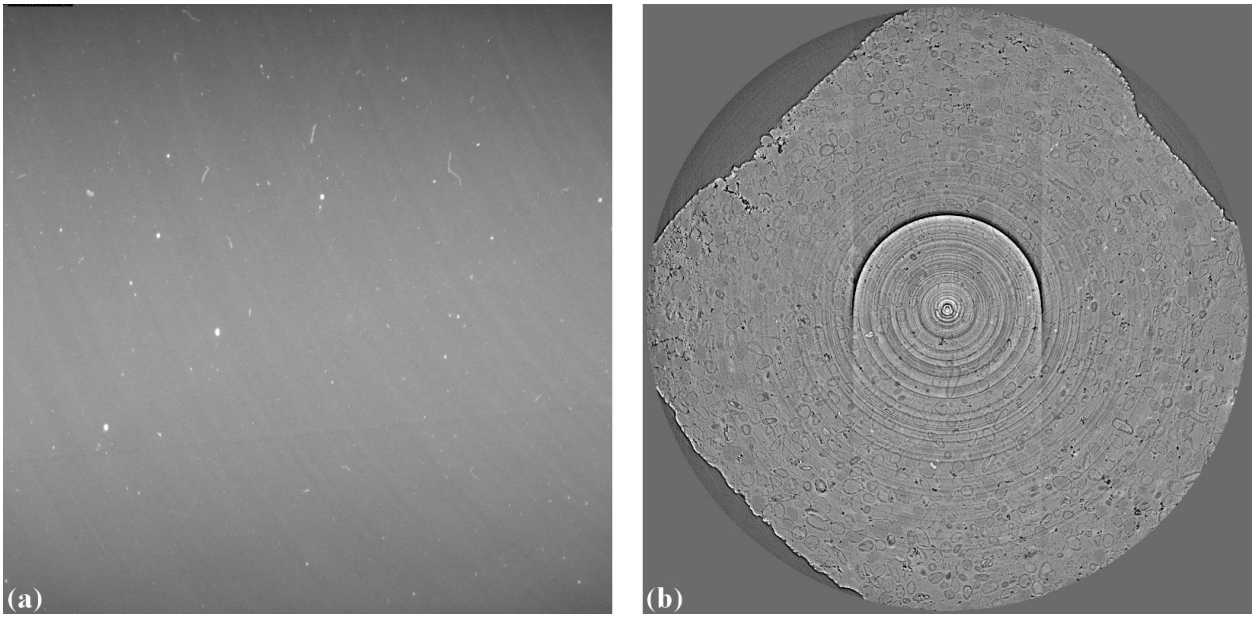


Fig. 1.1.17: Ring artifacts caused by defects of a scintillator of a detecting system. (a) Visible defects (white blobs) on a flat field image. (b) Ring artifacts caused by these defects.

The scattering of scintillation photons in a scintillator, of an indirect X-ray detector, has a strong effect to the linear response of the system and cause cupping artifacts in reconstructed images (Fig. 1.1.18).

Another common component of a detecting system is a lens which can has radial distortion problem. This problem gives raise to distinguishable artifacts in reconstructed images where artifacts only appear at some areas (Fig. 1.1.19).

Most of cameras are 16-bit, 14-bit, 12-bit, or 8-bit types. The number of bit dictates the dynamic range of intensity a camera can record. For example, a 16-bit camera can record intensities in the range of 0-65535 counts. In cases that the dynamic range (min-max) of incoming intensities are out of this range no matter how we adjust the exposure time, the acquired images can have underexposed areas or overexposed areas as shown in Fig. 1.1.20. In tomography, for samples giving a high dynamic range of transmission intensities we have to accept the underexposed areas which can give raise to cupping artifacts (Fig. 1.1.14).

Computing resources

Available computing resources such as GPU, multicore CPU, RAM, or storage system can dictate the choice of algorithms used for processing tomographic data. Fig. 1.1.21 shows the results of using two reconstruction methods: FBP and SIRT on a slice of a dataset of experiments using time-series tomography at beamline I12, Diamond Light Source, UK. The SIRT returns better result. However, it can't be used in practice due to the huge number of datasets acquired by the experiments. The total amount of data is ~250 TB and it would take years to reconstruct all of them using the SIRT method.

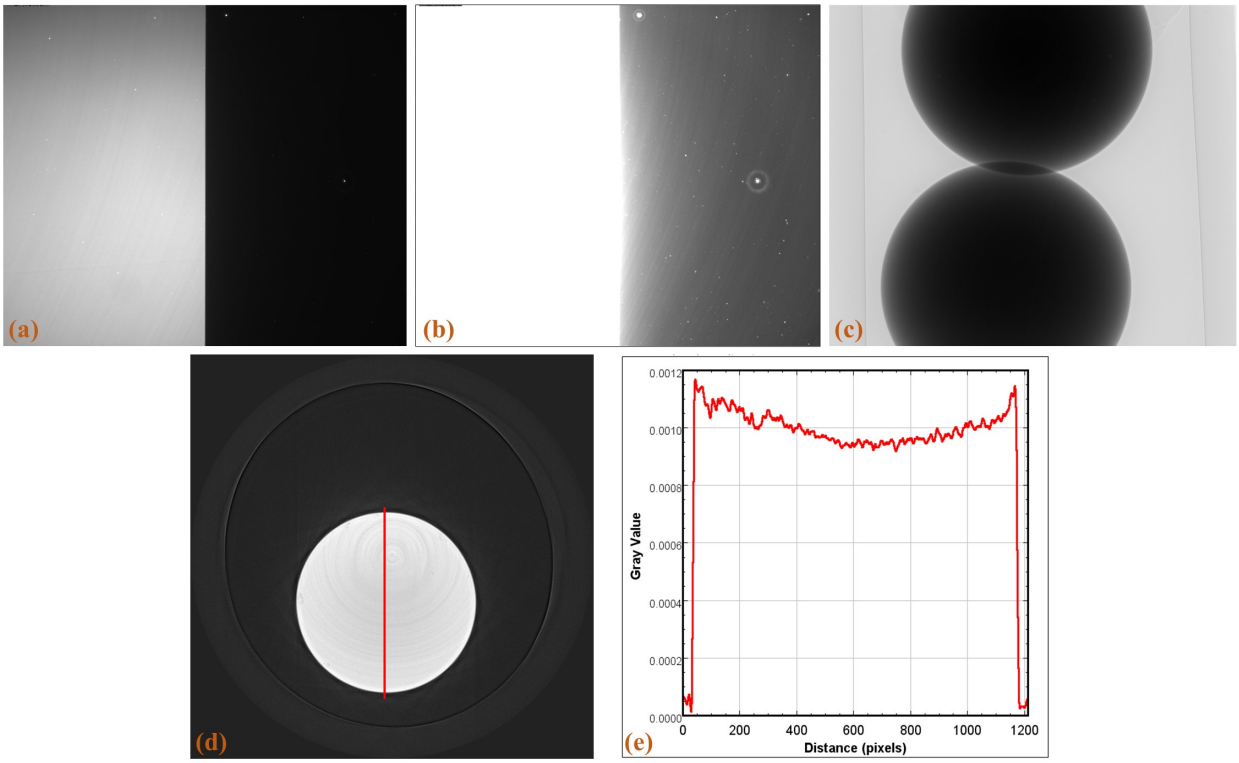


Fig. 1.1.18: Cupping artifacts caused by the scattering of scintillation photons. (a) Flat-field image with a half field of view completely blocked using 0.05 s of exposure time. (b) Same as (a) using 0.5 s of exposure time. (c) Projection image of a strong absorber. (d) Reconstructed image. (e) Line profile along the red line in (d).

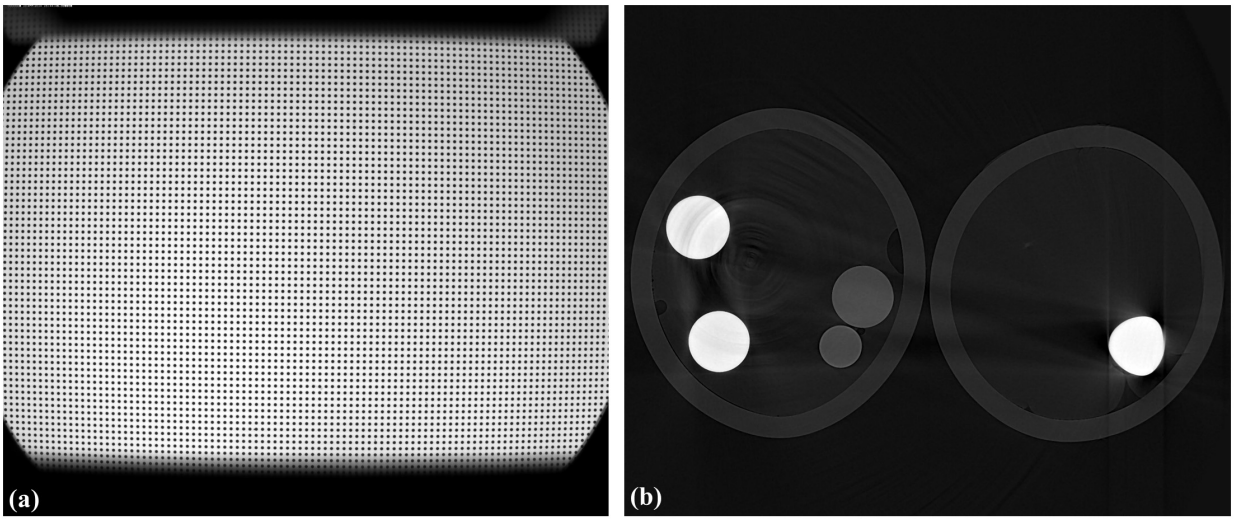


Fig. 1.1.19: Artifacts caused the lens-distortion problem. (a) Distorted image of a grid pattern. (b) Artifacts in a reconstructed image.

Fig. 1.1.20: Problems due to the limited dynamic range of a camera. (a) Underexposed area. (b) Overexposed area.

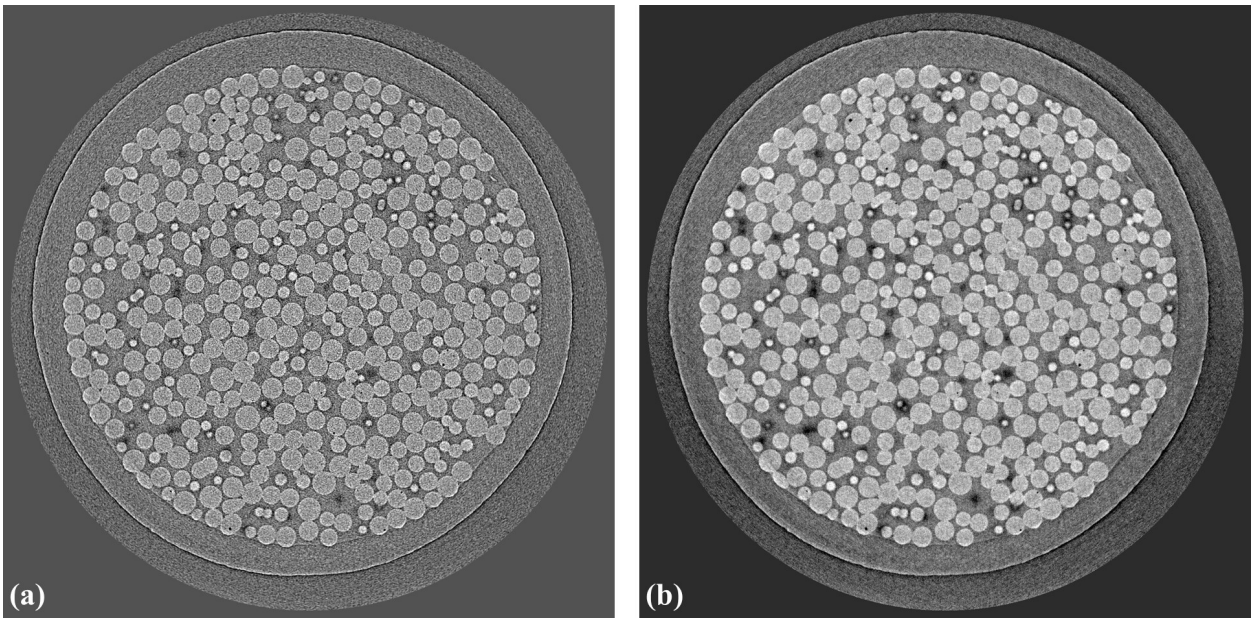


Fig. 1.1.21: Comparison of two reconstruction methods. (a) FBP. (b) SIRT.

1.1.4 Basic workflow for processing tomographic data

Read/write data

The first step is to know how to load data to workspace. Different communities use different file formats: hdf/nxs, mrc, txrm, xrm, tif, dicom, raw,... and they need specific Python libraries to work with. The common format used by synchrotron/neutron community is hdf/nxs. To load a dataset from a hdf/nxs file we need to know the key or path to the data. This can be done using Hdfviewer or Algotom's function as shown in *section 1.2*. For a tomographic hdf file, some basic metadata we need to know:

- Keys to projections images, flat-field images, and dark-field images.
- Key to rotation angles corresponding to projection images. This information may be not needed if the data was acquired in the ange range of [0; 180-degree].
- If the data is from a helical scan, extra information such as pixel size, pitch, and translation positions is needed.
- Information which is used by specific data processing methods such as pixel size, sample-detector distance, or X-ray energy.

Fig. 1.1.22 shows the content of a hdf file where users can find key to datasets used for tomographic reconstruction.

To load these data using Algotom's functions:

```
import algotom.io.loadersaver as losa

file = "C:/data/tomo_00064.h5"
proj_img = losa.load_hdf(file, key_path="exchange/data")  # This is an hdf␣
↪object, no data being loaded yet.
flat_img = losa.load_hdf(file, key_path="exchange/data_white")
dark_img = losa.load_hdf(file, key_path="exchange/data_dark")
angles = losa.load_hdf(file, key_path="exchange/theta")
```

Another tomographic hdf/nxs file acquired at beamline I12, Diamond Light Source (DLS) where the file structure and keys are different to the one before. In this data, dark-field images and flat-field images are at the same dataset as projection images where there is a dataset named "image_key" used to distinguish these images.

We can load the data, extract a projection image, and save it to tif.

```
import algotom.io.loadersaver as losa

file = "E:/Tomo_data/68067.nxs"
data_img = losa.load_hdf(file, key_path="entry1/tomo_entry/data/data") # This␣
↪is an hdf object.
# Extract a subset of data and save to tif
print(data_img.shape) # 1861 x 2160 x2560
losa.save_image("E:/output/image_00061.tif", data_img[61])
```

There are many Algotom's functions in the IO module to handle different tasks such as converting tif images to hdf, displaying the hdf tree, or saving output to a hdf file.

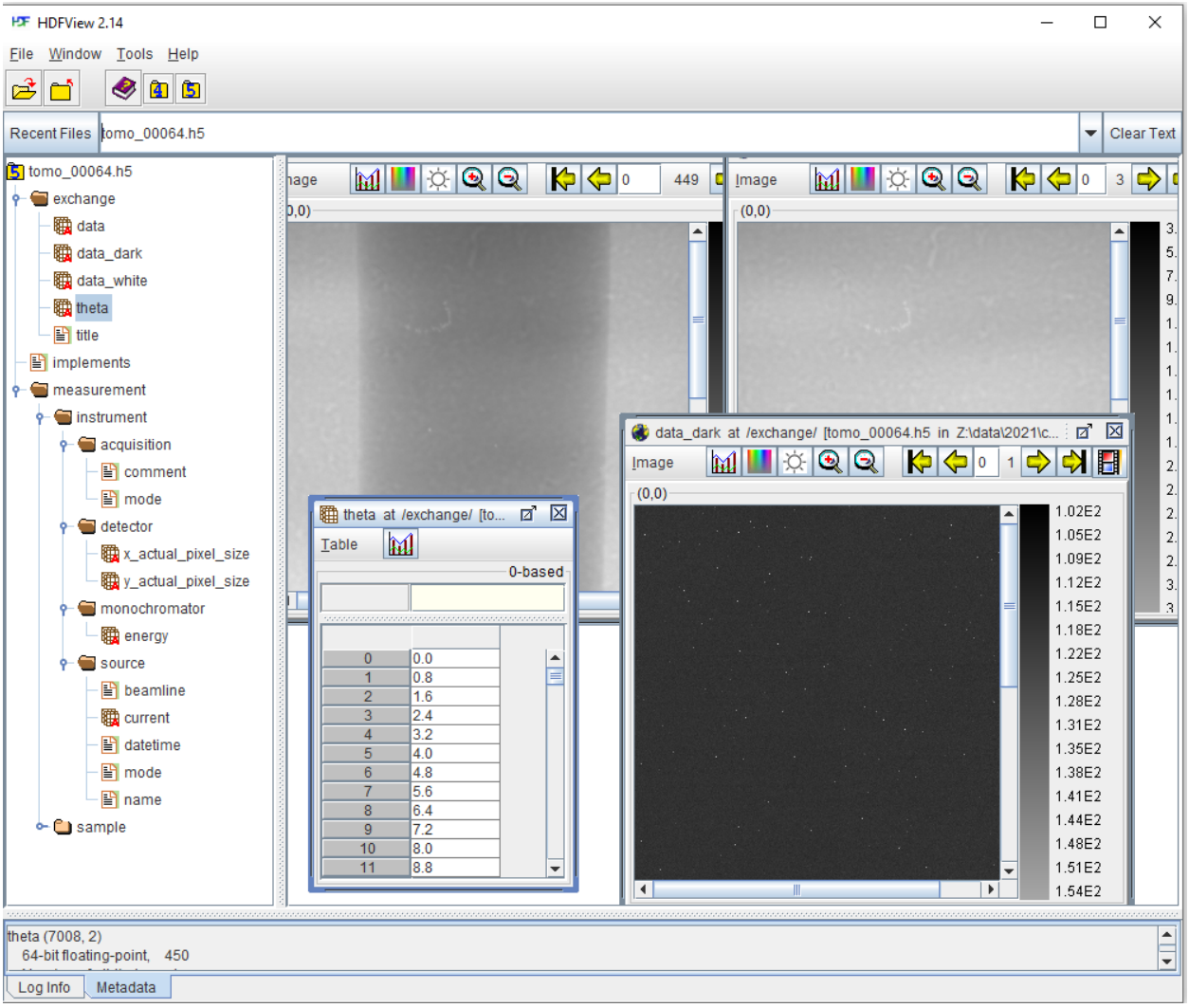


Fig. 1.1.22: Datasets of a tomographic hdf file.

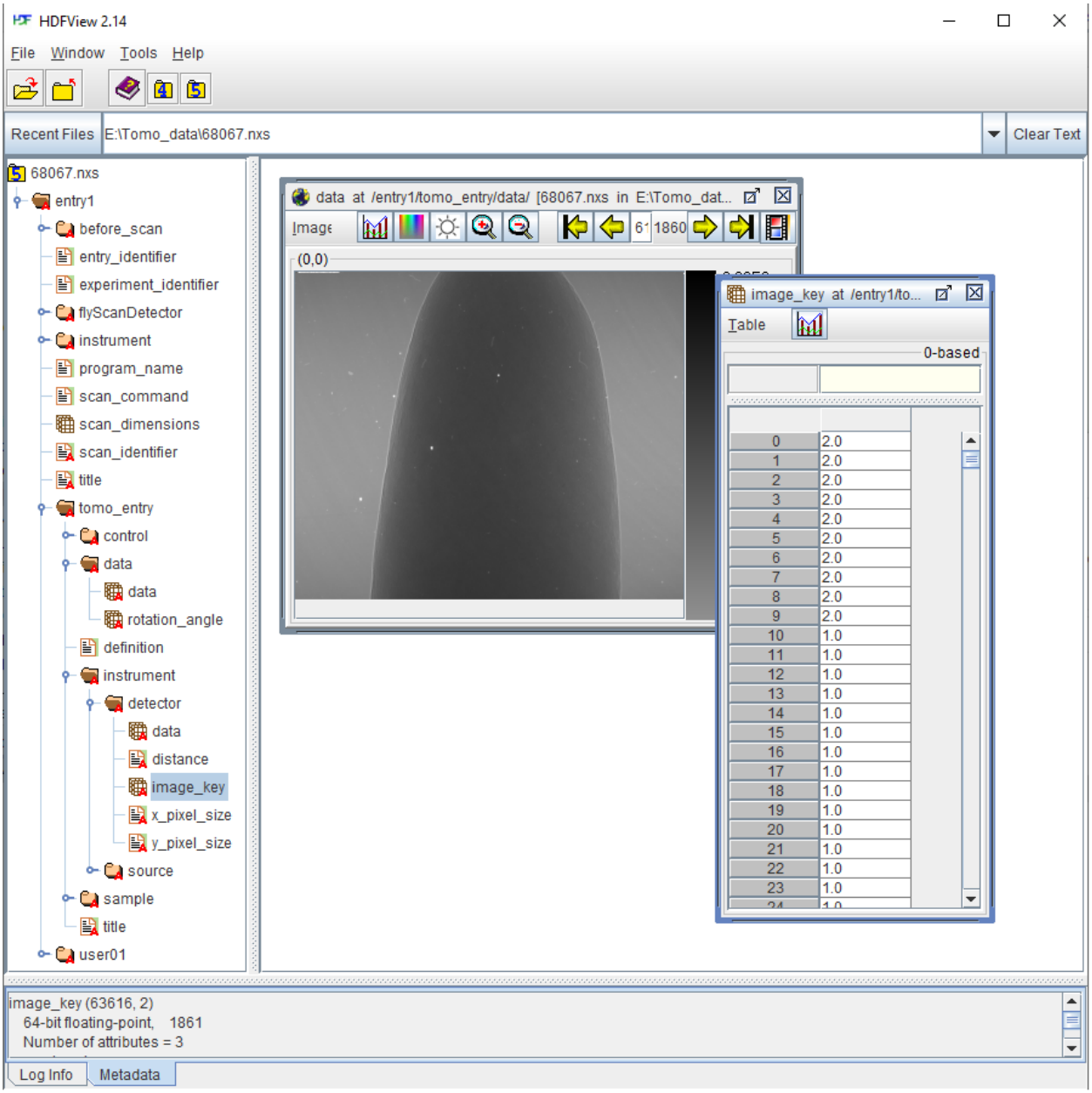


Fig. 1.1.23: Datasets of a tomographic hdf file acquired at DLS. Image-key value of 2 is for dark-field, 1 is for flat-field, and 0 is for projection image.

Flat-field correction

The flat-field correction process is based on the Beer-Lambert's law

$$\frac{I}{I_0} = \int e^{-\alpha(x,y,z)dx}$$

in practice, it is done using the following formula

$$\frac{P_\theta - D}{F - D}$$

where P_θ is a projection image of a sample at a rotation angle of θ, D is a dark-field image (camera's dark noise) taken with a photon source off, and F is a flat-field image taken without the sample. This can be done using Algotom as follows; data used in this demonstration can be download from here

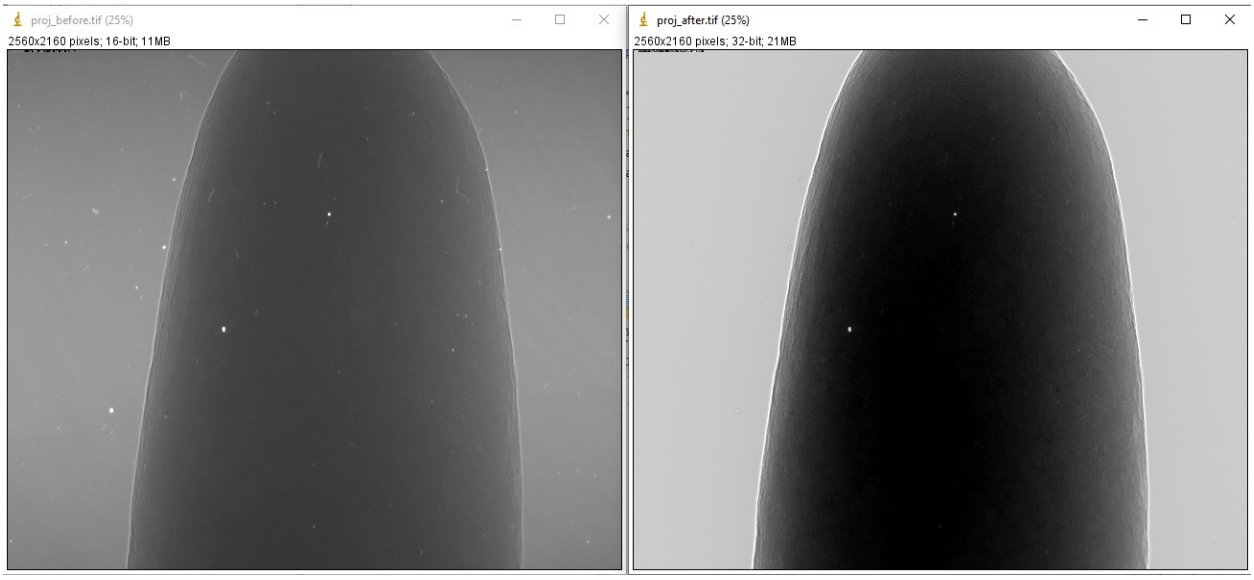
```
import numpy as np
import algotom.io.loadersaver as losa

file = "E:/Tomo_data/68067.nxs"
data_img = losa.load_hdf(file, key_path="entry1/tomo_entry/data/data") # This␣
↪is an hdf object.
# Get image key
ikey = losa.load_hdf(file, key_path="entry1/tomo_entry/instrument/detector/
↪image_key")
ikey = np.squeeze(np.asarray(ikey[:])) # Load data and convert to numpy 1d-
↪array.
# Use image_key to load flat-field images and average them
dark_field = np.mean(np.asarray(data_img[np.squeeze(np.where(ikey == 2.0)), :,␣
↪:]), axis=0)
flat_field = np.mean(np.asarray(data_img[np.squeeze(np.where(ikey == 1.0)), :,␣
↪:]), axis=0)
# Get indices of projection images
proj_idx = np.squeeze(np.where(ikey == 0))
# Apply flat-field correction to the first projection image.
proj_img = data_img[proj_idx[0]]
flat_dark = flat_field - dark_field
nmean = np.mean(flat_dark)
flat_dark[flat_dark == 0.0] = nmean  # Handle zero division
proj_norm = (proj_img - dark_field) / flat_dark
# Save images
losa.save_image("E:/output/proj_before.tif", proj_img)
losa.save_image("E:/output/proj_after.tif", proj_norm)
```

Running the code gives the output images

We can apply the process to a sinogram.

```
# Generate sinogram at the middle of an image height
(depth, height, width) = data_img.shape
sino_idx = height // 2
start = proj_idx[0]
stop = proj_idx[-1] + 1
sinogram = data_img[start:stop, sino_idx, :]
```

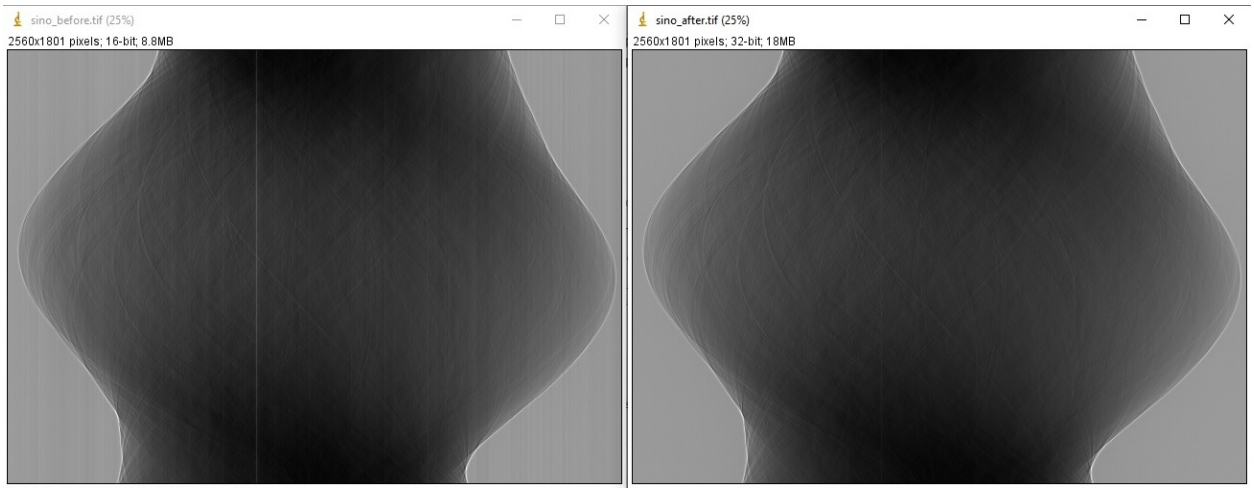
(continues on next page)



(continued from previous page)

```
# Apply flat-field correction the sinogram
sino_norm = (sinogram - dark_field[sino_idx]) / flat_dark[sino_idx]
# Save images
losa.save_image("E:/output/sino_before.tif", sinogram)
losa.save_image("E:/output/sino_after.tif", sino_norm)
```

which results in



Zinger removal

Zingers are prominent bright dots in images caused by scattered X-rays hitting the detector system CCD or CMOS chip (Fig. 1.1.24 (a,b)). They produce line artifacts across a reconstructed image (Fig. 1.1.24 (c)).

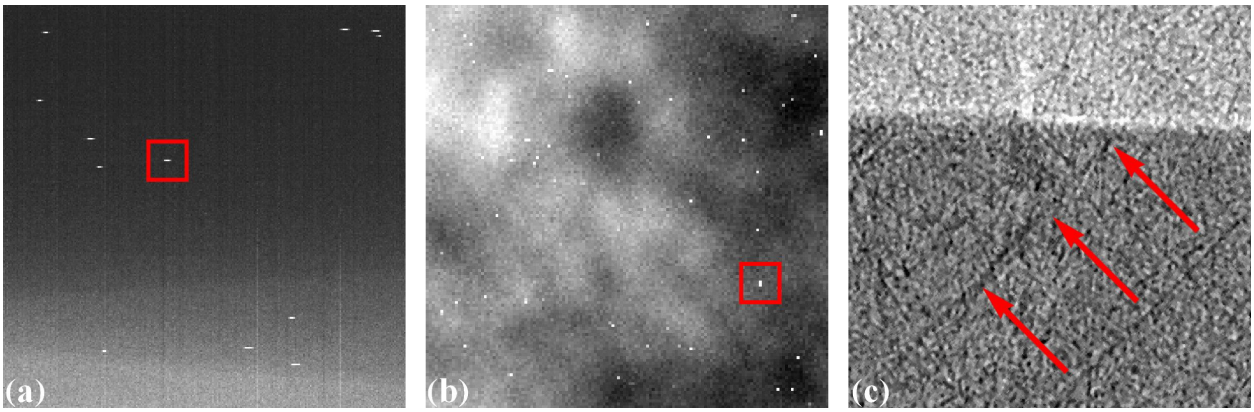


Fig. 1.1.24: Artifacts caused by zingers. (a) Zingers in the sinogram space. (b) Zingers in the projection space. (c) Line artifacts caused by the zingers.

Zingers are easily removed by using a method in Algotom

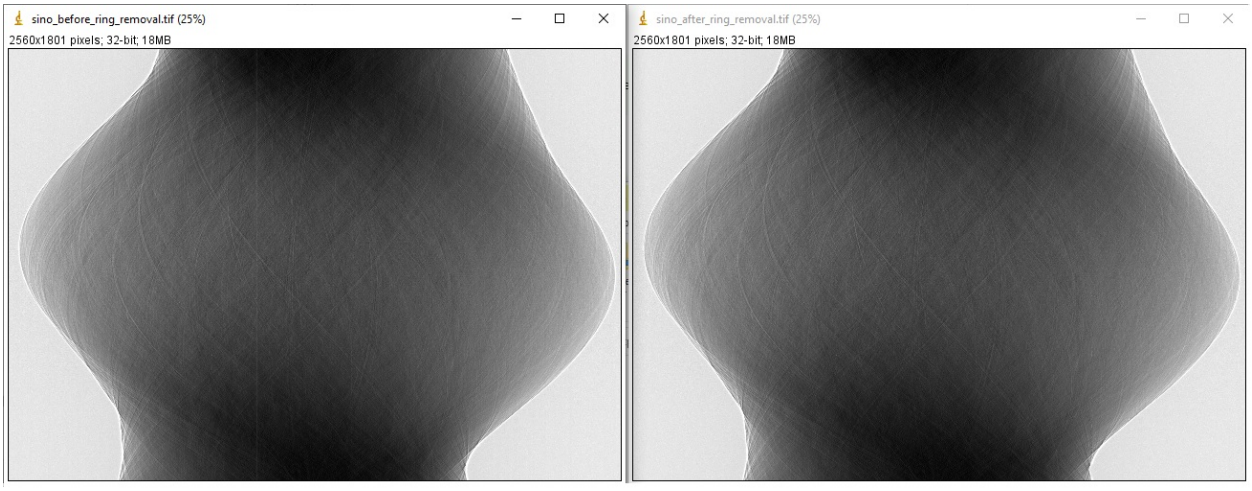
```
import algotom.prep.removal as rem

sino_rem1 = rem.remove_zinger(sino_norm, 0.005, size=2)
```

Ring artifact removal

Causes of ring artifacts and methods for removing them [R19] have been documented in detailed here. There are many methods to choose from in Algotom. However the combination of methods has been proven to be the most effective way to clean most of ring artifact types. Note that in the sinogram space, ring artifacts appear as stripe artifacts. Example of how to use the methods

```
sino_rem2 = rem.remove_all_stripe(sino_rem1, 3.1, 51, 21)
losa.save_image("E:/output/sino_before_ring_removal.tif", sino_rem1)
losa.save_image("E:/output/sino_after_ring_removal.tif", sino_rem2)
```

resulting in



Center-of-rotation determination

There are a few methods to determine the center-of-rotation. The demonstrated method [R21] below uses a 180-degree sinogram for calculation.

```
import algotom.prep.calculation as calc

center = calc.find_center_vo(sino_rem2, width // 2 - 50, width // 2 + 50)
print(center) # >> 1275.25
```

Denoising or contrast enhancement

There is a method for enhancing the contrast of an image, known as the Paganin filter which is commonly used at synchrotron facilities. Algotom implements a simplified version of this filter, named the Fresnel filter as it is based on the Fresnel propagator. There is a widespread misunderstanding in the community that the resulting image of the Paganin filter is a phase-contrast image. It is not. Because the filter acts as a low-pass filter, it reduces noise and the dynamic range of an image. This helps to enhance the contrast between low-contrast features which can be confused if this enhancement comes from the phase effect. Detailed demonstration for the argument is at here.

Note that a denoising filter or smoothing filter should not be used before the above pre-processing methods (zinger removal, ring artifact removal, center calculation). Blurring an image will impact the performance of these methods.

```
sino_filt1 = filt.fresnel_filter(sino_rem2, 200)
sino_filt2 = filt.fresnel_filter(sino_rem2, 1000)
losa.save_image("E:/output/sino_denoising_strength_200.tif", sino_filt1)
losa.save_image("E:/output/sino_denoising_strength_1000.tif", sino_filt2)
```

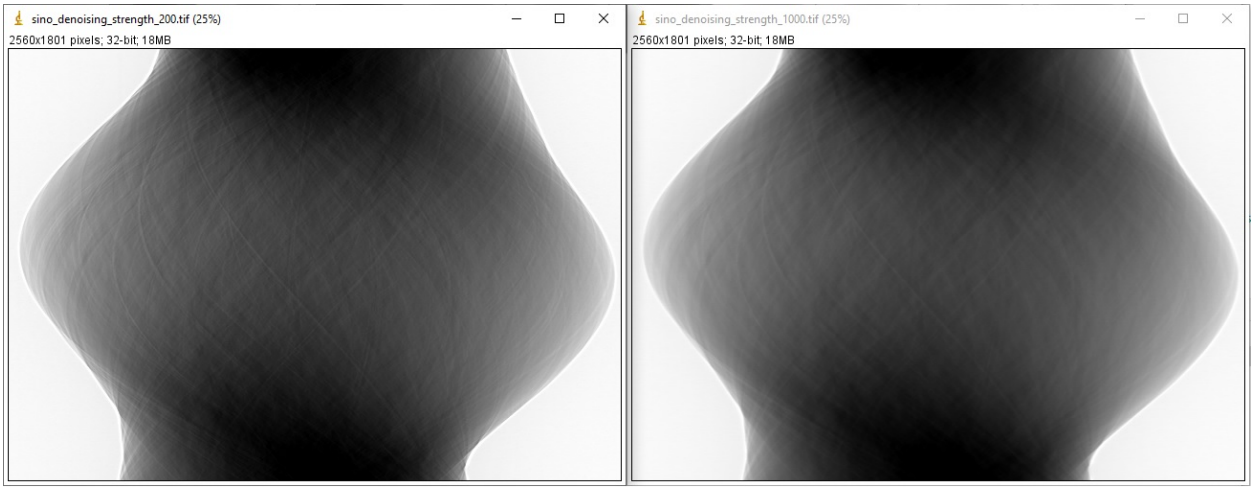


Fig. 1.1.25: Results of using the Fresnel filter. (a) Ratio = 200. (b) Ratio = 1000.

Image reconstruction

There are many choices for reconstruction methods and open-source software. In the current version (<=1.1), Algotom implements two FFT-based methods which is fast enough for a 2k x 2k x 2k dataset. Algotom also provides wrappers for other reconstruction methods available in Tomopy (gridrec) and Astra Toolbox (FBP, SIRT, SART, CGLS,...).

Examples of comparing reconstructed images before and after artifacts removal:

```
import algotom.rec.reconstruction as rec

# No need to pass angles if it's a 180-degree sinogram
rec_img1 = rec.dfi_reconstruction(sino_norm, center, angles=None)
rec_img2 = rec.dfi_reconstruction(sino_rem2, center, angles=None)
losa.save_image("E:/output/rec_with_artifacts.tif", rec_img1)
losa.save_image("E:/output/rec_artifacts_removed.tif", rec_img2)
```

Examples of comparing reconstructed images after using the Fresnel filter with different strengths:

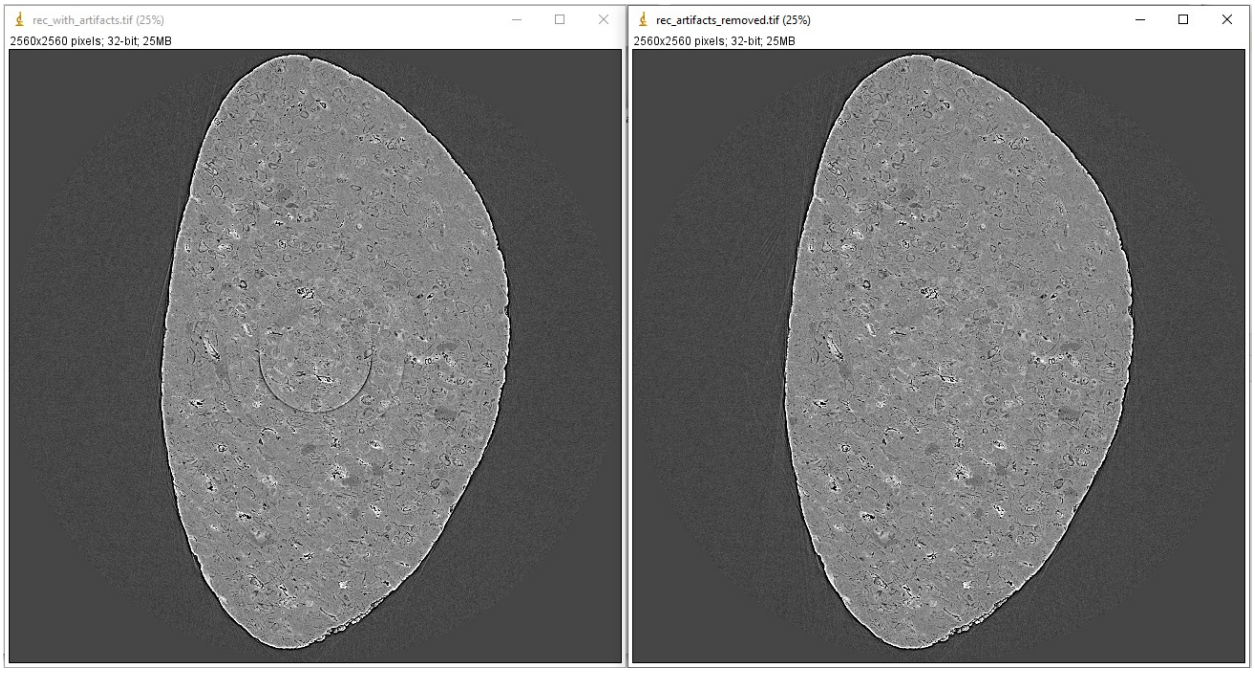
```
rec_img3 = rec.dfi_reconstruction(sino_filt1, center, angles=None)
rec_img4 = rec.dfi_reconstruction(sino_filt2, center, angles=None)
losa.save_image("E:/output/rec_filt1.tif", rec_img3)
losa.save_image("E:/output/rec_filt2.tif", rec_img4)
```

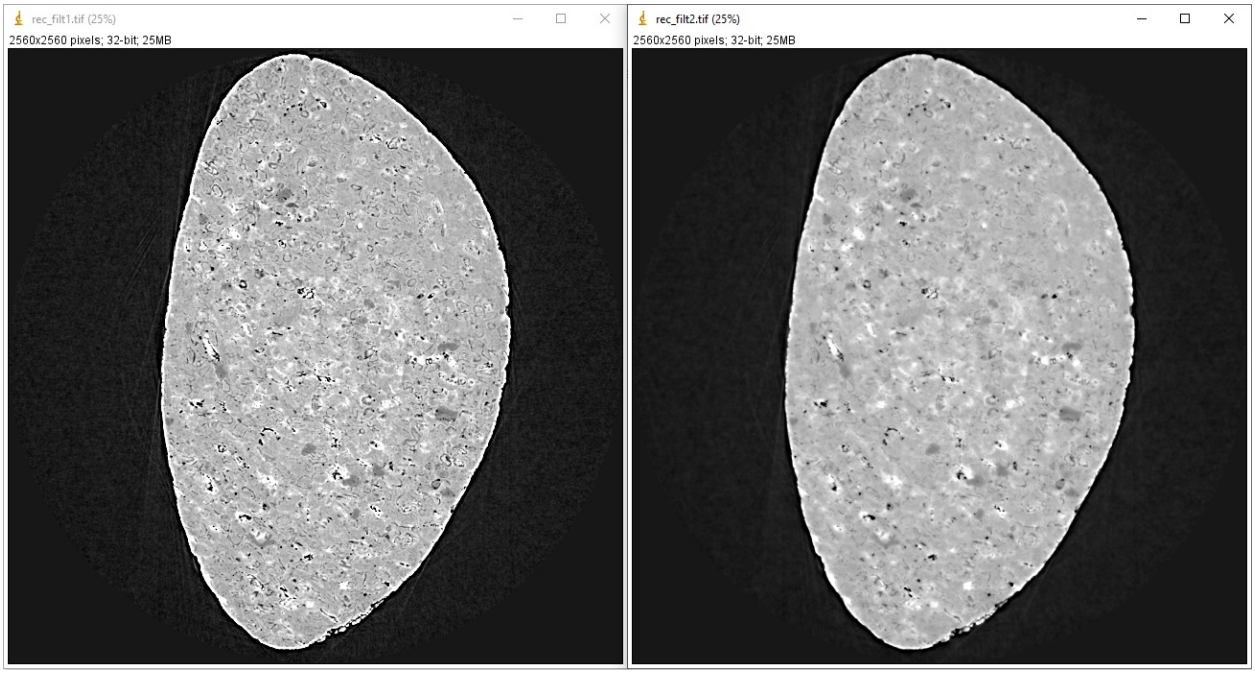
Other data processing steps

Distortion correction

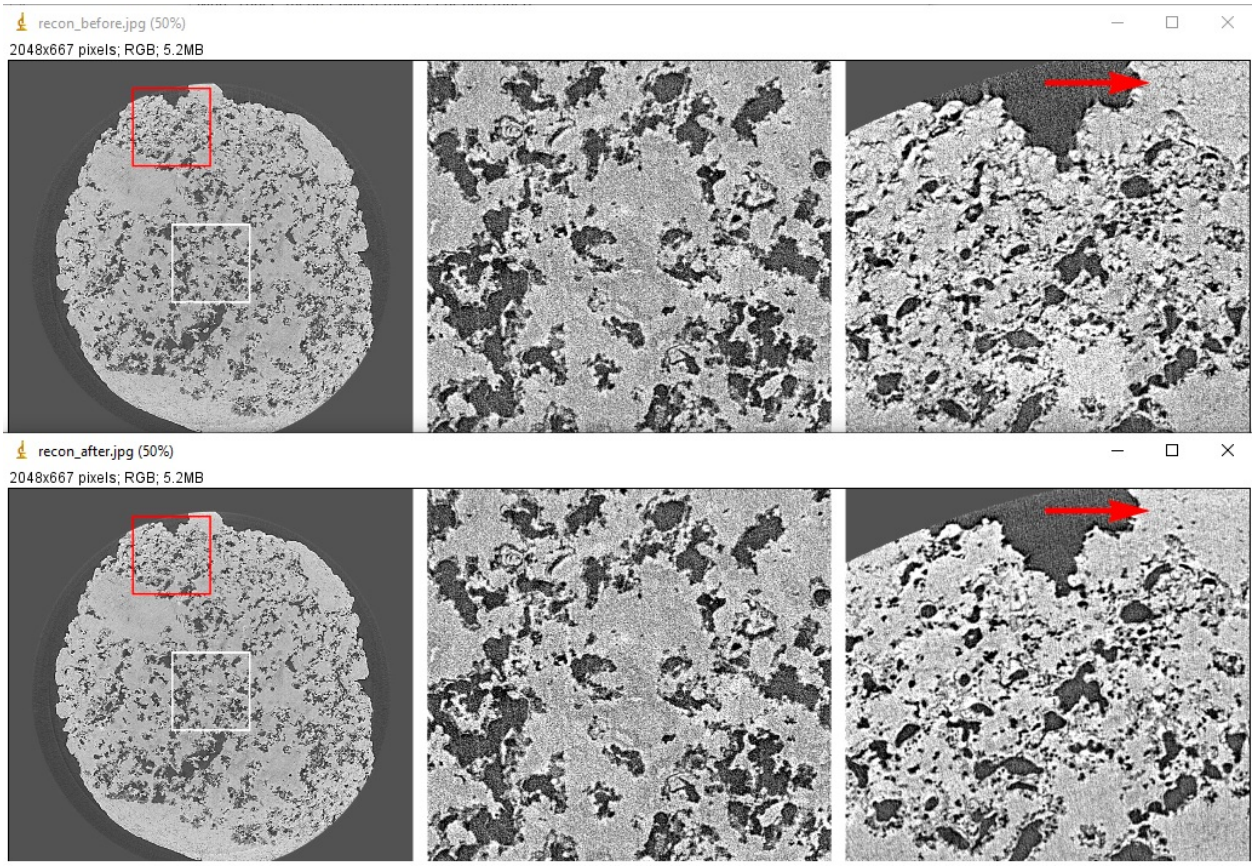
If a detecting system suffers from the lens-distortion problem, the working routine is as follows:

- Acquire a grid-pattern image.
- Calculate distortion coefficients [R18] using the Discorpy package. The output is a text file.
- Use the calculated coefficients for correction.


rec_with_artifacts.tif (25%)
2560x2560 pixels; 32-bit; 25MB
rec_artifacts_removed.tif (25%)
2560x2560 pixels; 32-bit; 25MB


rec_filt1.tif (25%)
2560x2560 pixels; 32-bit; 25MB
rec_filt2.tif (25%)
2560x2560 pixels; 32-bit; 25MB

```
import numpy as np
import algotom.io.loadersaver as losa
import algotom.prep.correction as corr
import algotom.prep.removal as remo
import algotom.prep.calculation as calc
import algotom.prep.filtering as filt
import algotom.rec.reconstruction as reco

# Paths to data. Download at: https://doi.org/10.5281/zenodo.3339629
proj_path = "E:/data/tomographic_projections.hdf"
flat_path = "E:/data/flats.hdf"
dark_path = "E:/data/darks.hdf"
coef_path = "E:/data/coefficients_bw.txt"
key_path = "/entry/data/data"

# Where to save the outputs
output_base = "E:/output/"

# Load data of projection images as an hdf object
proj_data = losa.load_hdf(proj_path, key_path)
(depth, height, width) = proj_data.shape

# Load flat-field images and dark-field images, average each of them
flat_field = np.mean(losa.load_hdf(flat_path, key_path)[:], axis=0)
dark_field = np.mean(losa.load_hdf(dark_path, key_path)[:], axis=0)
# Load distortion coefficients
xcenter, ycenter, list_fact = losa.load_distortion_coefficient(coef_path)
# Apply distortion correction to dark- and flat-field image.
flat_discor = corr.unwarp_projection(flat_field, xcenter, ycenter, list_
↪fact)
dark_discor = corr.unwarp_projection(dark_field, xcenter, ycenter, list_
↪fact)

# Generate a sinogram with distortion correction.
index = 800
sinogram = corr.unwarp_sinogram(proj_data, index, xcenter, ycenter, list_
↪fact)
sinogram = corr.flat_field_correction(sinogram, flat_discor[index], dark_
↪discor[index])
sinogram = remo.remove_all_stripe(sinogram, 3.0, 51, 17)
center = calc.find_center_vo(sinogram, width // 2 - 50, width // 2 + 50)
# Reconstruct image from the sinogram
rec_img = reco.dfi_reconstruction(sinogram, center, angles=None, apply_
↪log=True)
losa.save_image(output_base + "/rec_00800.tif", rec_img)
```



Sinogram stitching for a half-acquisition scan

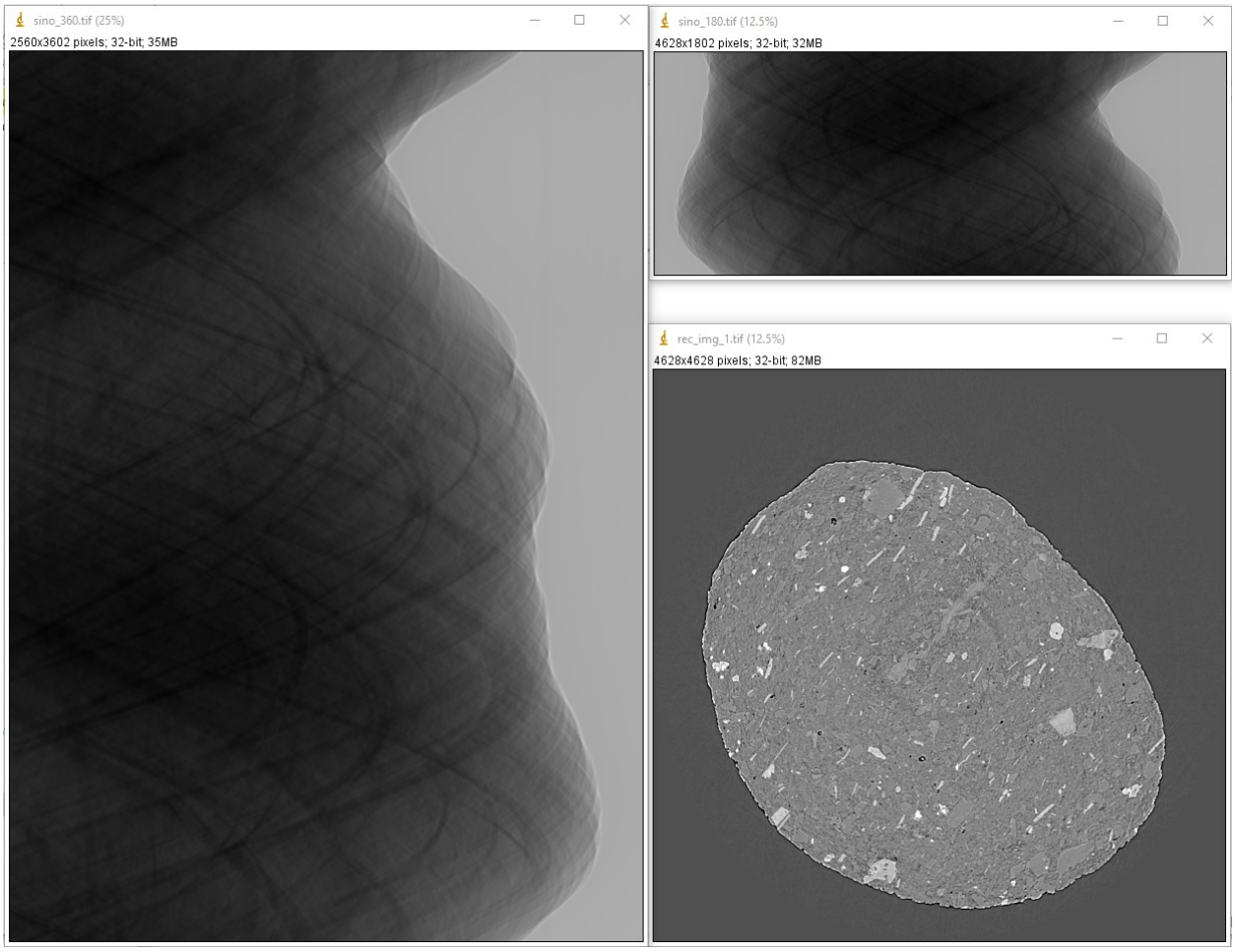
Half-acquisition scanning technique are being used more often at synchrotron facilities. It is a simple technique to double the field-of-view (FOV) of a tomography system by shifting the rotation axis to a side of the FOV then acquiring data in the angle range of [0, 360-degree]. To process the data, a 360-degree sinogram is converted to an equivalent 180-degree sinogram by stitching two halves of the 360-degree sinogram, before reconstruction. For stitching, we need to know either the center-of-rotation, or the overlap-area and overlap-side between two halves of the sinogram. Algotom provides methods [C1] for automatically finding these parameters.

```
import numpy as np
import algotom.io.loadersaver as losa
import algotom.prep.correction as corr
import algotom.prep.removal as remo
import algotom.prep.calculation as calc
import algotom.prep.conversion as conv
import algotom.rec.reconstruction as reco

input_base = "E:/data/"
output_base = "E:/output/"

# Data at: https://doi.org/10.5281/zenodo.4386983
proj_path = input_base + "/scan_00008/projections_00000.hdf"
flat_path = input_base + "/scan_00009/flats_00000.hdf"
dark_path = input_base + "/scan_00009/darks_00000.hdf"
```

(continues on next page)

(continued from previous page)

```
meta_path = input_base + "/scan_00008/scan_00008.nxs"
key_path = "/entry/data/data"
angle_key = "/entry1/tomo_entry/data/rotation_angle"

data = losa.load_hdf(proj_path, key_path)
(depth, height, width) = data.shape
angles = np.squeeze(np.asarray(losa.load_hdf(meta_path, angle_key)[:]))
# Load dark-field images and flat-field images, averaging each result.
flat_field = np.mean(losa.load_hdf(flat_path, key_path)[:], axis=0)
dark_field = np.mean(losa.load_hdf(dark_path, key_path)[:], axis=0)

# Generate a sinogram and perform flat-field correction.
index = height // 2
sino_360 = corr.flat_field_correction(data[:, index, :], flat_field[index],␣
↪dark_field[index])

# Calculate the center-of-rotation, the overlap-side and overlap-area used for␣
↪stitching
(center0, overlap, side, _) = calc.find_center_360(sino_360, 100)

# Remove zingers
sino_360 = remo.remove_zinger(sino_360, 0.08)
# Remove ring artifacts
sino_360 = remo.remove_all_stripe(sino_360, 3, 51, 17)
# Convert the 360-degree sinogram to the 180-degree sinogram.
sino_180, center1 = conv.convert_sinogram_360_to_180(sino_360, center0)
losa.save_image(output_base + "/sino_360.tif", sino_360)
losa.save_image(output_base + "/sino_180.tif", sino_180)

# Perform reconstruction
rec_img = reco.dfi_reconstruction(sino_180, center1, apply_log=True)
losa.save_image(output_base + "/rec_img_1.tif", rec_img)

# 2nd way: extend the 360-degree sinogram. It's useful for tomography fly-scans
# where the two halves of a 360-degree sinogram are mismatch due to the angle
# step is not divisible.
(sino_ext, center2) = conv.extend_sinogram(sino_360, center0)
# Perform reconstruction
# Using fbp-method for angle range > 180 degree
img_rec = reco.fbp_reconstruction(sino_ext, center2, angles=angles * np.pi /␣
↪180.0,
                                  apply_log=False, gpu=True)
losa.save_image(output_base + "/rec_img_2.tif", img_rec)
```


sino_360.tif (25%)
2560x3602 pixels; 32-bit; 35MB
sino_180.tif (12.5%)
4628x1802 pixels; 32-bit; 32MB
rec_img_1.tif (12.5%)
4628x4628 pixels; 32-bit; 82MB

1.1.5 Parallel processing in Python

Having a multicore CPU, certainly we want to make use of it for parallel processing. This is easily done using the Joblib library. Explanation of the functions is as follow

```
from joblib import Parallel, delayed

# Note the use of parentheses
results = Parallel(n_jobs=8, prefer="threads")(delayed(func_name)(func_para1,␣
↪func_para2) for i in range(i_start, i_stop, i_step))
```

The first part of the code, `Parallel(n_jobs=8, prefer="threads")` , is to select the number of cores and a back-end method for parallelization. The second part of the code, `(delayed()() for ...)` has 3 sub-sections: the name of a function, its parameters, and the loop. We can also use nested loops

```
results = Parallel(n_jobs=8, prefer="threads")(delayed(func_name)(func_para1,␣
↪func_para2) for i in range(i_start, i_stop, i_step) \
                                                                              ␣
↪          for j in range(j_start, j_stop, j_step))
```

Note that `results` is a list of the outputs of the function used. The order of the items in the list corresponding to how the loops are defined. The following examples will make things more clear.

- Example to show the output order of nested loops:

```
from joblib import Parallel, delayed

def print_order(i, j):
    print("i = {0}; j = {1} \n".format(i, j))
    return i, j

results = Parallel(n_jobs=4, prefer="threads")(delayed(print_order)(i, j) for i in␣
↪range(0, 2, 1) \
                                                                    for j in␣
↪range(2, 4, 1))
print("Output = ", results)
```

```
>>>
i = 0; j = 2
i = 0; j = 3
i = 1; j = 2
i = 1; j = 3
Output =  [(0, 2), (0, 3), (1, 2), (1, 3)]
```

- Example to show how to apply a smoothing filter to multiple images in parallel

```
import timeit
import multiprocessing as mp
import numpy as np
import scipy.ndimage as ndi
from joblib import Parallel, delayed

# Select number of cpu cores
ncore = 16
```

(continues on next page)

(continued from previous page)

```
if ncore > mp.cpu_count():
    ncore = mp.cpu_count()

# Create data for testing
height, width = 3000, 5000
image = np.zeros((height, width), dtype=np.float32)
image[1000:2000, 1500:3500] = 1.0
n_slice = 16
data = np.moveaxis(np.asarray([i * image for i in range(n_slice)]), 0, 1)
print(data.shape) # >>> (3000, 16, 5000)

# Using sequential computing for comparison
t0 = timeit.default_timer()
results = []
for i in range(n_slice):
    mat = ndi.gaussian_filter(data[:, i, :], (3, 5), 0)
    results.append(mat)
t1 = timeit.default_timer()
print("Time cost for sequential computing: ", t1 - t0) # >>> 8.831482099999999

# Using parallel computing
t0 = timeit.default_timer()
results = Parallel(n_jobs=16, prefer="threads")(delayed(ndi.gaussian_filter)(data[:,
↪ i, :], (3, 5), 0) for i in range(n_slice))
t1 = timeit.default_timer()
print("Time cost for parallel computing: ", t1 - t0)   # >>> 0.8372323000000002

# As the output is a list we have to convert it to a numpy array
# and reshape to get back the original shape
results = np.asarray(results)
print(results.shape)  # >>> (16, 3000, 5000)
results = np.moveaxis(results, 0, 1)
print(results.shape)  # >>> (3000, 16, 5000)
```

There are several options for choosing the backend methods. Depending on the problem and how input data are used, their performance can be significantly different. In the above example, the "threads" option gives the best performance. Note that we can't use the above approaches for parallel reading or writing data from/to a hdf file. There is a different way of doing these.

- Users can also refer to how Algotom uses Joblib for different use-cases as shown here, here, or here.

1.1.6 Alignment for a parallel-beam tomography system

Due to the parallelism of the penetrating X-rays, 2D projections can be divided into independent 1D projection rows. Collecting 1D projections at a specific row angularly forms a sinogram, which is used to reconstruct a 2D slice of an object (Fig. 1.1.26). To ensure the independence of 1D projections at each row, it is crucial to maintain the rotation axis parallel to the imaging plane and perpendicular to each image row. This requirement is known as tomography alignment. In a synchrotron-based tomography system, the high configurability in using different optics magnifications and/or sample-detector distances often causes the rotation axis being misaligned with the imaging plane. As the result, alignment adjustments are necessary for different tomography setups.

The misalignment of a tomography system is identified by measuring the tilt and roll angle of the rotation axis relative to the imaging plane. This is achieved by scanning a point-like object, such as a sphere offset from the rotation axis,

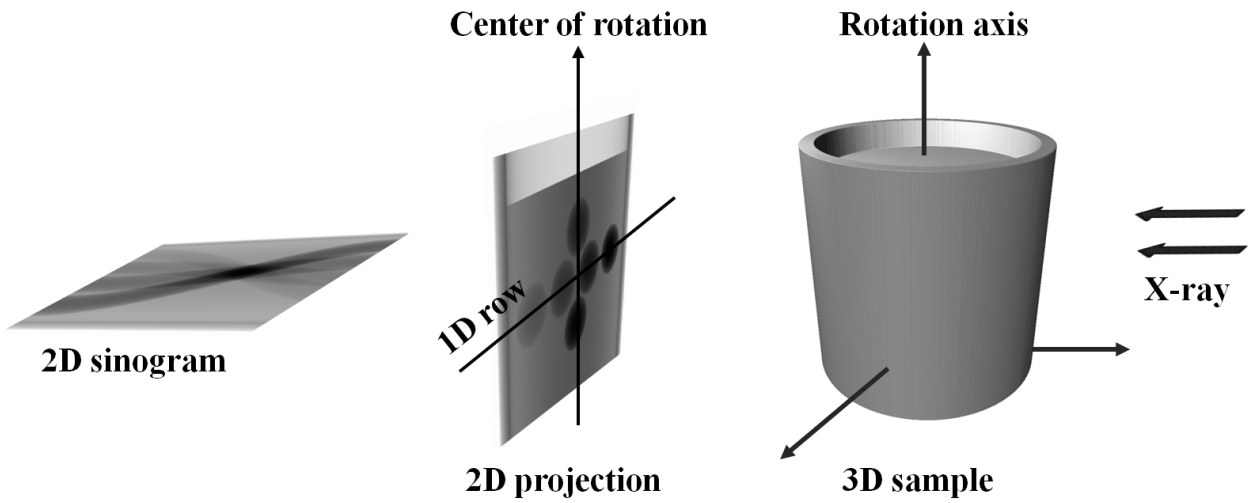


Fig. 1.1.26: Schematic of parallel-beam X-ray tomography.

through a full rotation and tracking the trajectory of its center of mass, as illustrated in Fig. 1.1.27. A similar approach can be employed using a needle by tracking the top of it through a full rotation.

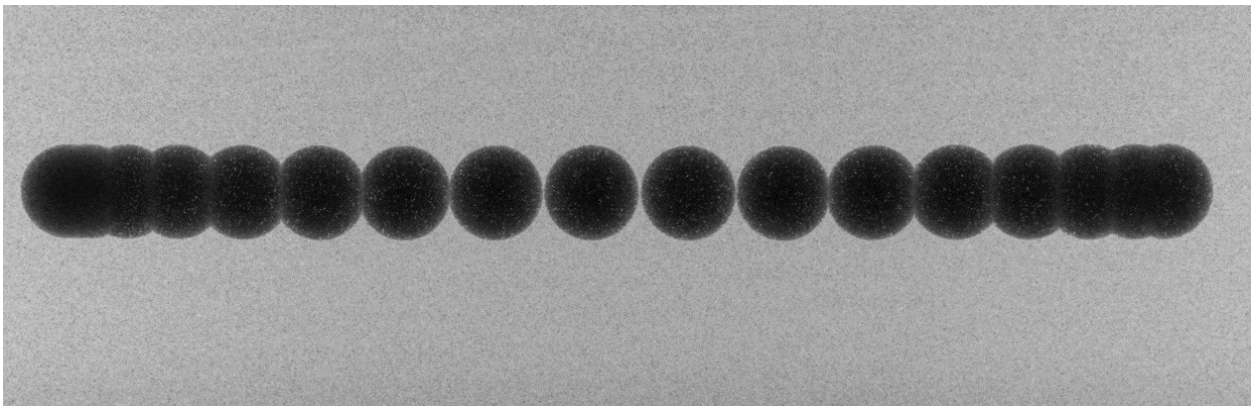


Fig. 1.1.27: Overlay of projections of a sphere during a circular scan.

In a well-aligned system, the range of y-coordinates of points remains below 1 pixel, as depicted in Fig. 1.1.28. If the system is misaligned, the y-coordinates of points will appear as an ellipse; where the roll angle corresponds to the angle of the major axis, and the tilt is related to the ratio between the minor and major axes of the ellipse.

This section demonstrates how to use methods available in Algotom to calculate the tilt and roll angle of the rotation axis from projections of a sphere scanned over the range of [0, 360] degrees.

- Load the raw data and the corresponding flat-field images:

```
import numpy as np
import scipy.ndimage as ndi
import matplotlib.pyplot as plt
import algotom.io.loadersaver as losa
import algotom.util.calibration as calib

proj_path = "/tomo/data/scan_00001/"
flat_path = "/tomo/data/scan_00002/"
```

(continues on next page)

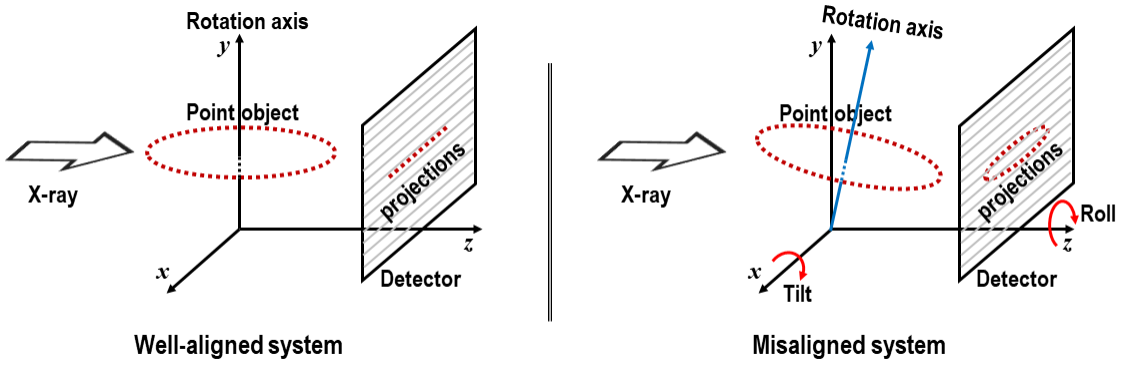


Fig. 1.1.28: Demonstration of a well-aligned tomography system and a misaligned one

(continued from previous page)

```
# If inputs are tif files
proj_files = losa.find_file(proj_path + "/*.tif*")
flat_files = losa.find_file(flat_path + "/*.tif*")
proj_data = np.asarray([losa.load_image(file) for file in proj_files])
flat = np.mean(np.asarray([losa.load_image(file) for file in flat_files]),
               axis=0)

# # If inputs are hdf files
# hdf_key = "entry/data/data"  # Change to the correct key.
# proj_data = losa.load_hdf(proj_path, hdf_key)
# (depth, height, width) = proj_data.shape
# flat = np.mean(np.asarray(losa.load_hdf(flat_path, hdf_key)), axis=0)
flat[flat == 0.0] = np.mean(flat)

have_flat = True
fit_ellipse = True  # Use an ellipse-fit method
ratio = 1.0  # To adjust the threshold for binarization

crop_left = 10
crop_right = 10
crop_top = 1000
crop_bottom = 1000

figsize = (15, 7)
(depth, height, width) = proj_data.shape
left = crop_left
right = width - crop_right
top = crop_top
bottom = height - crop_bottom
width_cr = right - left
height_cr = bottom - top
```

- For each projection, multiple preprocessing steps are applied to segment the sphere and determine its center of mass. These steps include flat-field correction, background removal, binarization, and the removal of non-spherical objects, as follows:

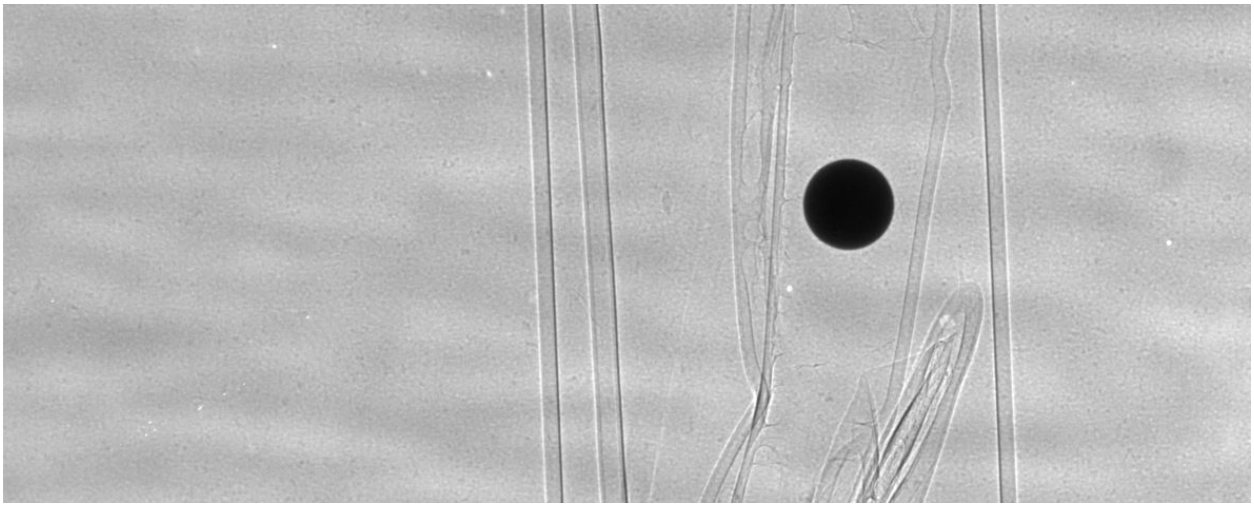


Fig. 1.1.29: Projection of the sphere

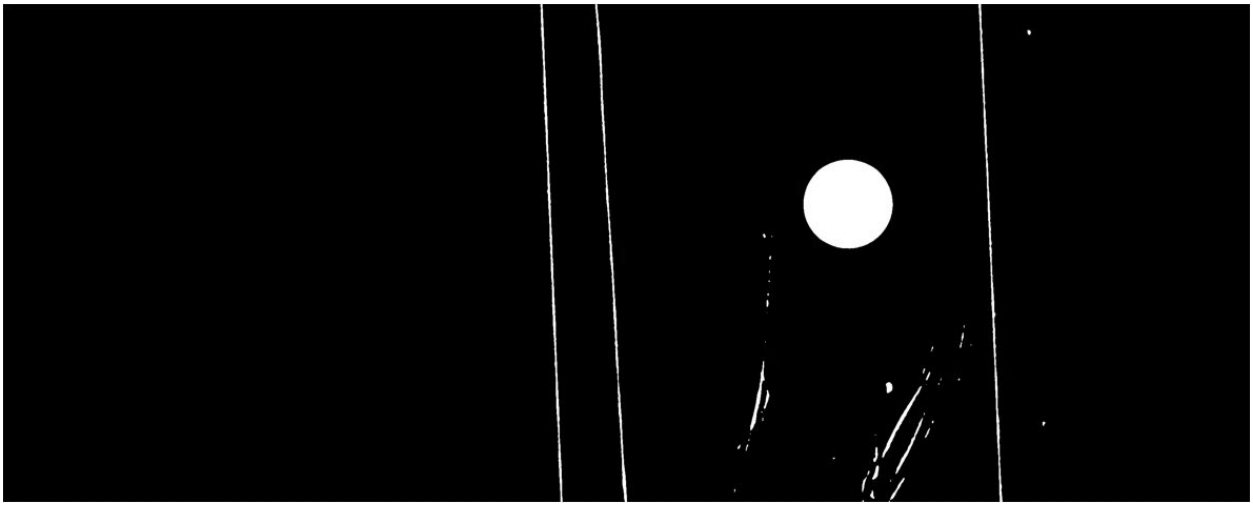


Fig. 1.1.30: Binarized image

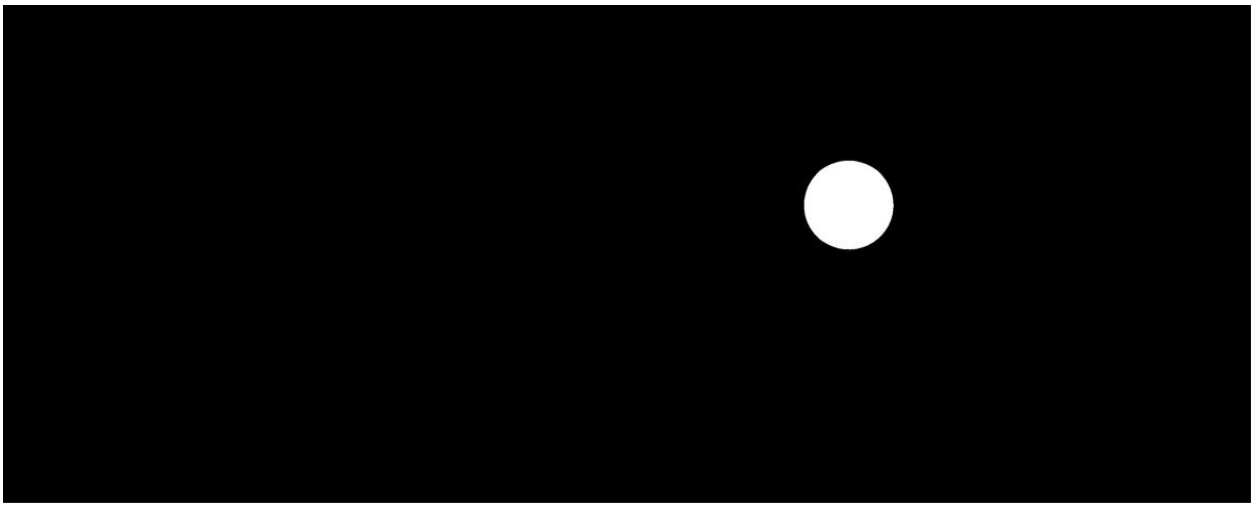


Fig. 1.1.31: Segmented sphere

```
x_centers = []
y_centers = []
img_list = []
print("\n==========================================")
print("Extract the sphere and get its center-of-mass\n")

for i, img in enumerate(proj_data):
    # Crop image and perform flat-field correction
    if have_flat:
        mat = img[top: bottom, left:right] / flat[top: bottom, left:right]
    else:
        mat = img[top: bottom, left:right]
    # Denoise
    mat = ndi.gaussian_filter(mat, 2)
    # Normalize the background.
    # Optional, should be used if there's no flat-field.
    mat = calib.normalize_background_based_fft(mat, 5)
    threshold = calib.calculate_threshold(mat, bgr='bright')
    # Binarize the image
    mat_bin0 = calib.binarize_image(mat, threshold=ratio * threshold, bgr='bright')
    sphere_size = calib.get_dot_size(mat_bin0, size_opt="max")
    # Keep the sphere only
    mat_bin = calib.select_dot_based_size(mat_bin0, sphere_size)
    nmean = np.sum(mat_bin)
    if nmean == 0.0:
        print("\
↪n*********************************************************************")
        print("Adjust threshold or crop the FOV to remove objects larger than the␣
↪sphere!")
        print("Current threshold used: {}".format(threshold))
        print(
↪"*********************************************************************")
        plt.figure(figsize=figsize)
```

(continues on next page)

(continued from previous page)

```
        plt.imshow(mat_bin0, cmap="gray")
        plt.show()
        raise ValueError("No binary object selected!")
    (y_cen, x_cen) = ndi.center_of_mass(mat_bin)
    x_centers.append(x_cen)
    y_centers.append(height_cr - y_cen)
    img_list.append(mat)
    print("  ---> Done image: {}".format(i))
x = np.float32(x_centers)
y = np.float32(y_centers)
img_list = np.asarray(img_list)
img_overlay = np.min(img_list, axis=0)
```

- The coordinates of the center of mass of the sphere are used to calculate the till and roll either using an ellipse-fit method or a linear-fit method.

```
# ============================================================================

def fit_points_to_ellipse(x, y):
    if len(x) != len(y):
        raise ValueError("x and y must have the same length!!!")
    A = np.array([x ** 2, x * y, y ** 2, x, y, np.ones_like(x)]).T
    vh = np.linalg.svd(A, full_matrices=False)[-1]
    a0, b0, c0, d0, e0, f0 = vh.T[:, -1]
    denom = b0 ** 2 - 4 * a0 * c0
    msg = "Can't fit to an ellipse!!!"
    if denom == 0:
        raise ValueError(msg)
    xc = (2 * c0 * d0 - b0 * e0) / denom
    yc = (2 * a0 * e0 - b0 * d0) / denom
    roll_angle = np.rad2deg(
        np.arctan2(c0 - a0 - np.sqrt((a0 - c0) ** 2 + b0 ** 2), b0))
    if roll_angle > 90.0:
        roll_angle = - (180 - roll_angle)
    if roll_angle < -90.0:
        roll_angle = (180 + roll_angle)
    a_term = 2 * (a0 * e0 ** 2 + c0 * d0 ** 2 - b0 * d0 * e0 + denom * f0) * (
            a0 + c0 + np.sqrt((a0 - c0) ** 2 + b0 ** 2))
    if a_term < 0.0:
        raise ValueError(msg)
    a_major = -2 * np.sqrt(a_term) / denom
    b_term = 2 * (a0 * e0 ** 2 + c0 * d0 ** 2 - b0 * d0 * e0 + denom * f0) * (
            a0 + c0 - np.sqrt((a0 - c0) ** 2 + b0 ** 2))
    if b_term < 0.0:
        raise ValueError(msg)
    b_minor = -2 * np.sqrt(b_term) / denom
    if a_major < b_minor:
        a_major, b_minor = b_minor, a_major
        if roll_angle < 0.0:
            roll_angle = 90 + roll_angle
        else:
            roll_angle = -90 + roll_angle
```

(continues on next page)

(continued from previous page)

```
    return roll_angle, a_major, b_minor, xc, yc

# ============================================================================
# Calculate the tilt and roll using an ellipse-fit or a linear-fit method

if fit_ellipse is True:
    (a, b) = np.polyfit(x, y, 1)[:2]
    dist_list = np.abs(a * x - y + b) / np.sqrt(a ** 2 + 1)
    dist_list = ndi.gaussian_filter1d(dist_list, 2)
    if np.max(dist_list) < 1.0:
        fit_ellipse = False
        print("\nDistances of points to a fitted line is small, "
              "Use a linear-fit method instead!\n")

if fit_ellipse is True:
    try:
        result = fit_points_to_ellipse(x, y)
        roll_angle, major_axis, minor_axis, xc, yc = result
        tilt_angle = np.rad2deg(np.arctan2(minor_axis, major_axis))
    except ValueError:
        # If can't fit to an ellipse, using a linear-fit method instead
        fit_ellipse = False
        print("\nCan't fit points to an ellipse, using a linear-fit method instead!\
↪n")

if fit_ellipse is False:
    (a, b) = np.polyfit(x, y, 1)[:2]
    dist_list = np.abs(a * x - y + b) / np.sqrt(a ** 2 + 1)
    appr_major = np.max(np.asarray([np.sqrt((x[i] - x[j]) ** 2 +
                                            (y[i] - y[j]) ** 2)
                                    for i in range(len(x))
                                    for j in range(i + 1, len(x))]))
    dist_list = ndi.gaussian_filter1d(dist_list, 2)
    appr_minor = 2.0 * np.max(dist_list)
    tilt_angle = np.rad2deg(np.arctan2(appr_minor, appr_major))
    roll_angle = np.rad2deg(np.arctan(a))

print("================================================")
print("Roll angle: {} degree".format(roll_angle))
print("Tilt angle: {} degree".format(tilt_angle))
print("================================================\n")
```

- Show the results:

```
# Show the results
plt.figure(1, figsize=figsize)
plt.imshow(img_overlay, cmap="gray", extent=(0, width_cr, 0, height_cr))
plt.tight_layout(rect=[0, 0, 1, 1])

plt.figure(0, figsize=figsize)
plt.plot(x, y, marker="o", color="blue")
plt.title(
```

(continues on next page)

(continued from previous page)

```
    "Roll : {0:2.4f}; Tilt : {1:2.4f} (degree)".format(roll_angle, tilt_angle))
if fit_ellipse is True:
    # Use parametric form for plotting the ellipse
    angle = np.radians(roll_angle)
    theta = np.linspace(0, 2 * np.pi, 100)
    x_fit = (xc + 0.5 * major_axis * np.cos(theta) * np.cos(
        angle) - 0.5 * minor_axis * np.sin(theta) * np.sin(angle))
    y_fit = (yc + 0.5 * major_axis * np.cos(theta) * np.sin(
        angle) + 0.5 * minor_axis * np.sin(theta) * np.cos(angle))
    plt.plot(x_fit, y_fit, color="red")
else:
    plt.plot(x, a * x + b, color="red")
plt.xlabel("x")
plt.ylabel("y")
plt.tight_layout()
plt.show()
```

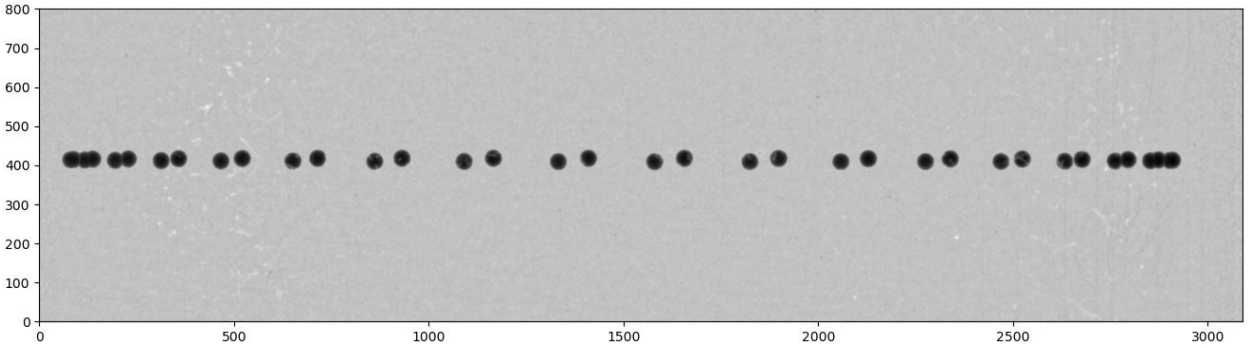


Fig. 1.1.32: Overlay of projections of a sphere for checking.

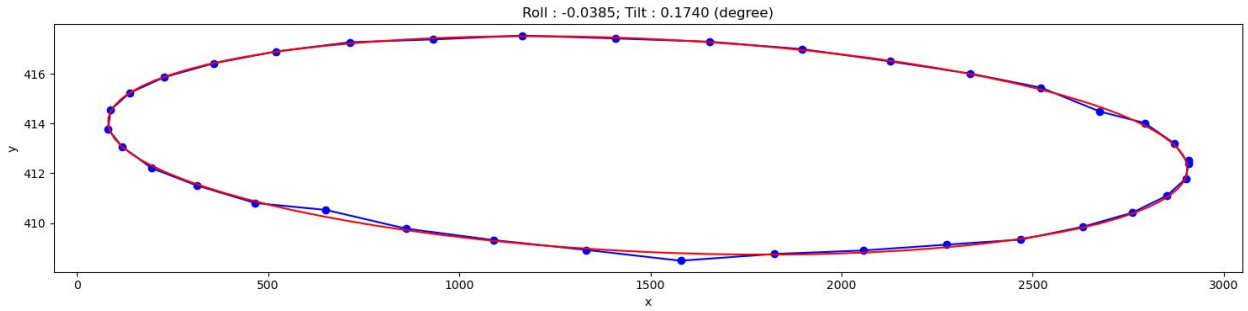


Fig. 1.1.33: Showing the result of finding the tilt and roll.

From the given results, we can adjust the rotation axis or the detector system accordingly. Note that the calculated angles are based only on input images, so the sign of the angles does not reflect the true geometry of a tomography system. Using information such as the direction of rotation when scanning spheres and/or camera orientation, we can correctly identify the sign of these angles. After the adjustment, calculation results should be as follows:

The above routine performs very well in practice. However, if the projection images are of low quality due to blobs on the scintillator or optics system, an additional cleaning step for image processing (using some functions in the scikit-image library) can be included as follows:

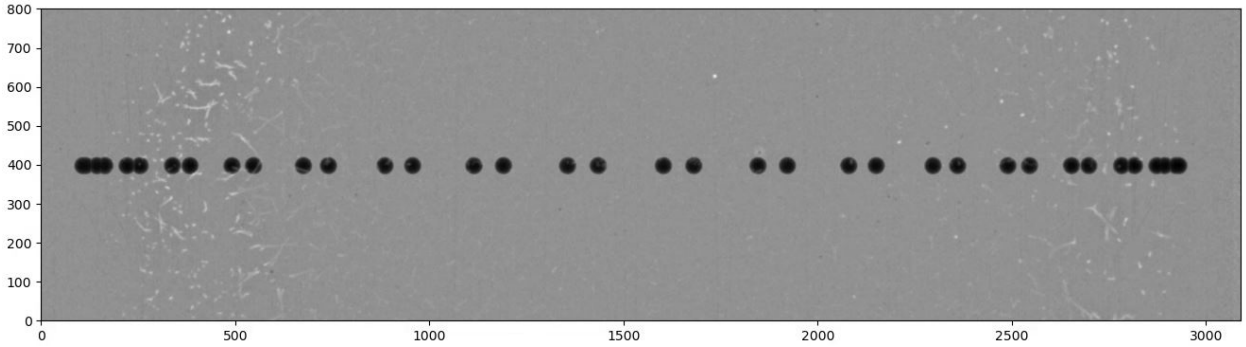


Fig. 1.1.34: Overlay of projections of a sphere after alignment.

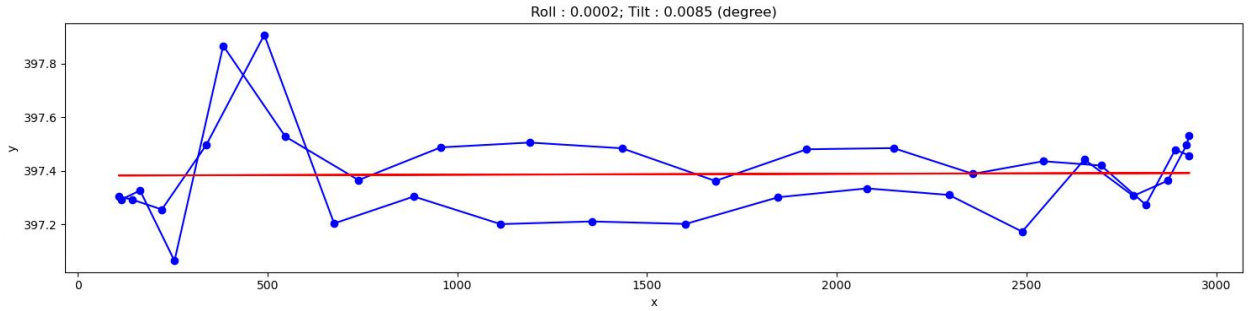


Fig. 1.1.35: Result of finding the tilt and roll after alignment.

```
from skimage import measure, segmentation

def remove_non_round_objects(binary_image, ratio_threshold=0.9):
    """
    To clean binary image and remove non-round objects
    """
    binary_image = segmentation.clear_border(binary_image)
    binary_image = ndi.binary_fill_holes(binary_image)
    label_image = measure.label(binary_image)
    properties = measure.regionprops(label_image)
    mask = np.zeros_like(binary_image, dtype=bool)
    # Filter objects based on the axis ratio
    for prop in properties:
        if prop.major_axis_length > 0:
            axis_ratio = prop.minor_axis_length / prop.major_axis_length
            if axis_ratio > ratio_threshold:
                mask[label_image == prop.label] = True
    # Apply mask to keep only round objects
    filtered_image = np.logical_and(binary_image, mask)
    return filtered_image

# ...
# Binarize the image
mat_bin0 = calib.binarize_image(mat, threshold=ratio * threshold, bgr='bright')
# Clean the image
```

(continues on next page)

(continued from previous page)

```
mat_bin0 = remove_non_round_objects(mat_bin0)
sphere_size = calib.get_dot_size(mat_bin0, size_opt="max")
# Keep the sphere only
# ...
```

The complete script and its **commandline user interface (CLI) version** are available here. If users prefer an interactive way of assessing tomographic alignment as shown below, the ImageJ macros can be downloaded from here.

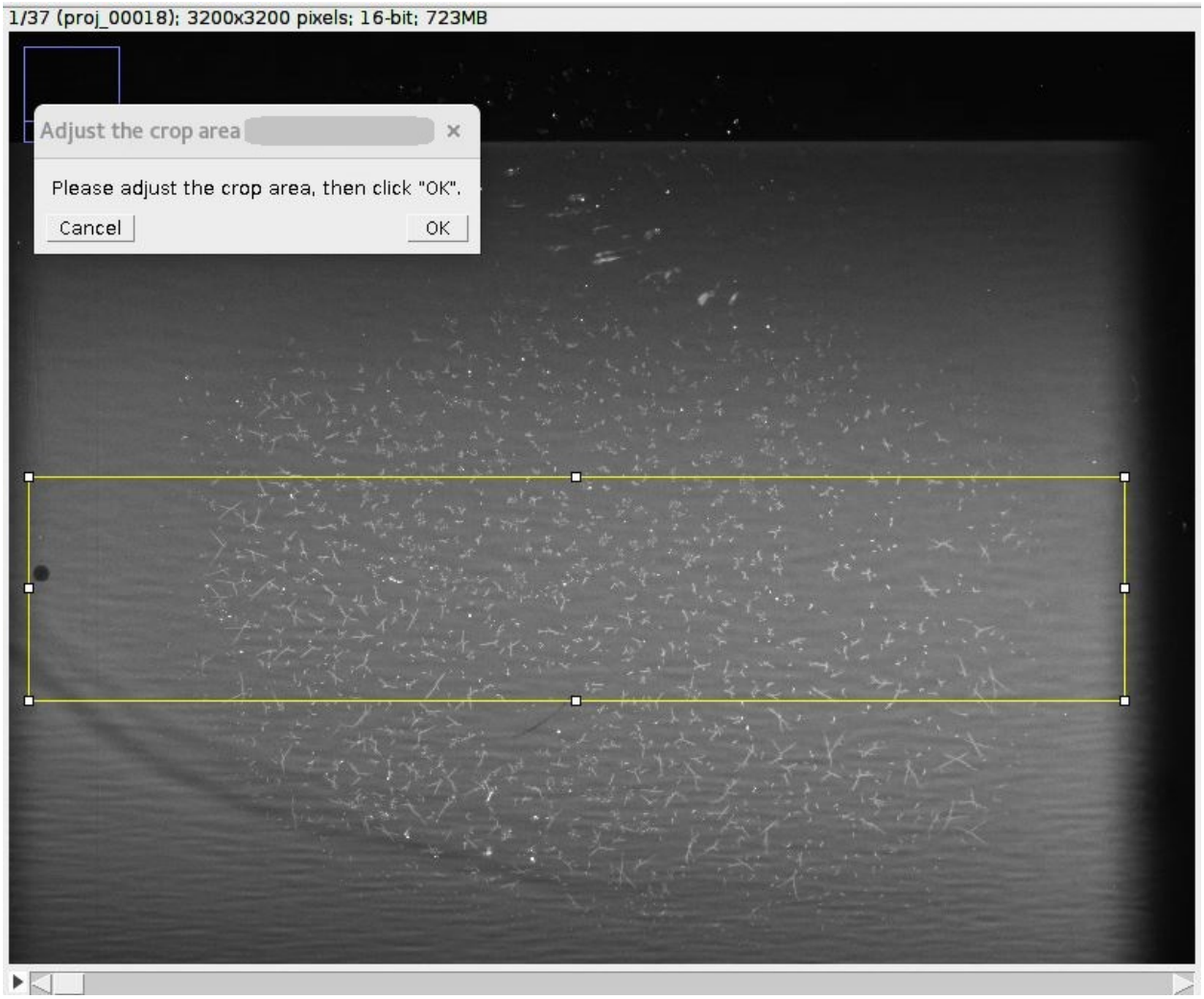


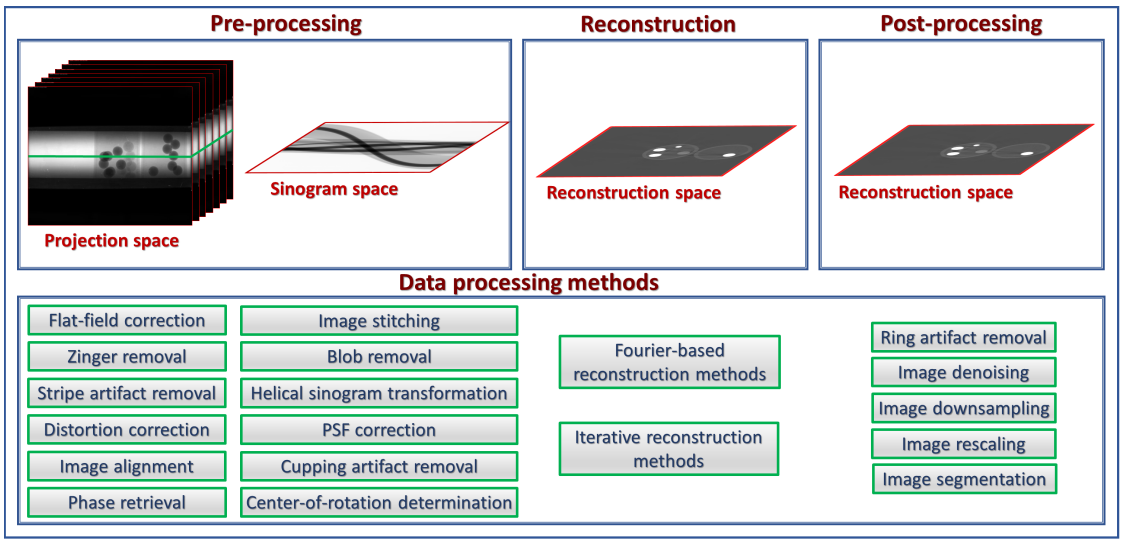
Fig. 1.1.36: Interactive approach for tomography alignment using ImageJ macro.

1.2 Features

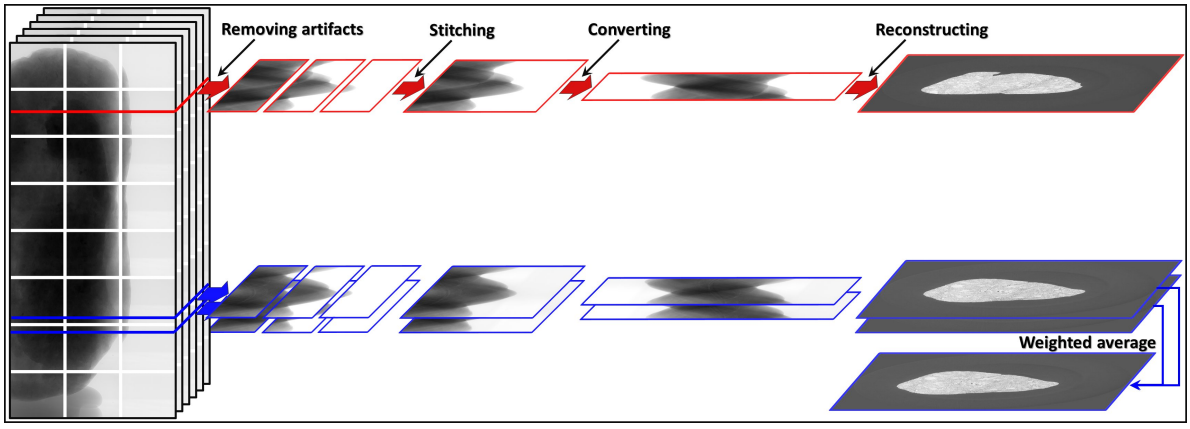
1.2.1 Capabilities

Algotom is a lightweight package. The software is built on top of a few core Python libraries to ensure its ease-of-installation. Methods distributed in Algotom have been developed and tested at synchrotron beamlines where massive datasets are produced. This factor drives the methods developed to be easy-to-use, robust, and practical. **Algotom can be used on a normal computer to process large tomographic data**. Some featuring methods in Algotom are as follows:

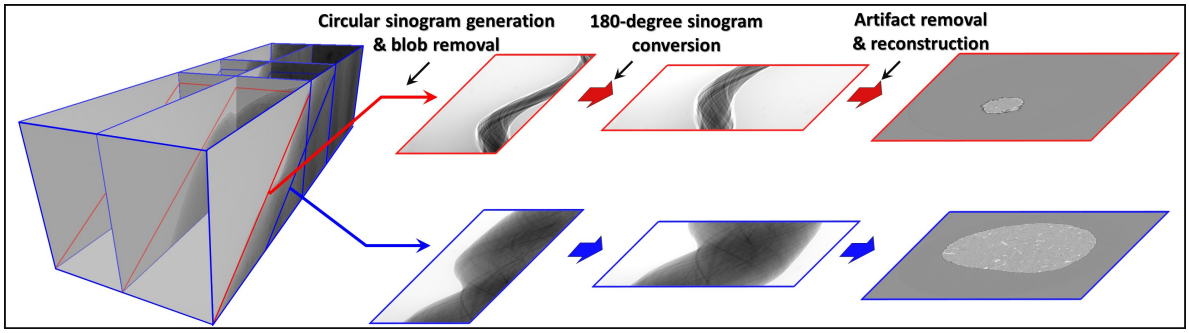
- Methods in a full data processing pipeline: reading-writing data, pre-processing, tomographic reconstruction, and post-processing.



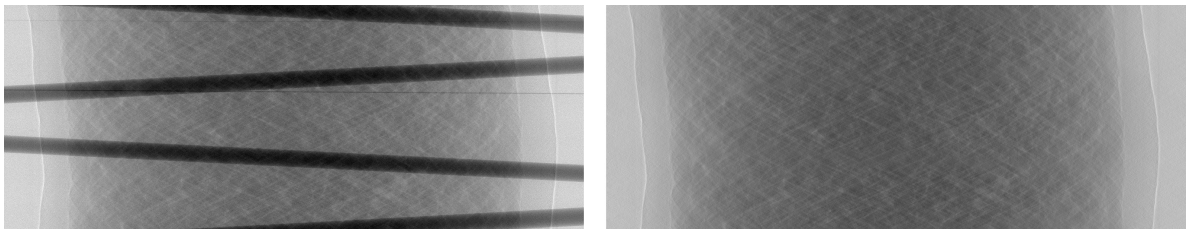
- Methods for processing grid scans (or tiled scans) with the offset rotation-axis to multiply double the field-of-view (FOV) of a parallel-beam tomography system.



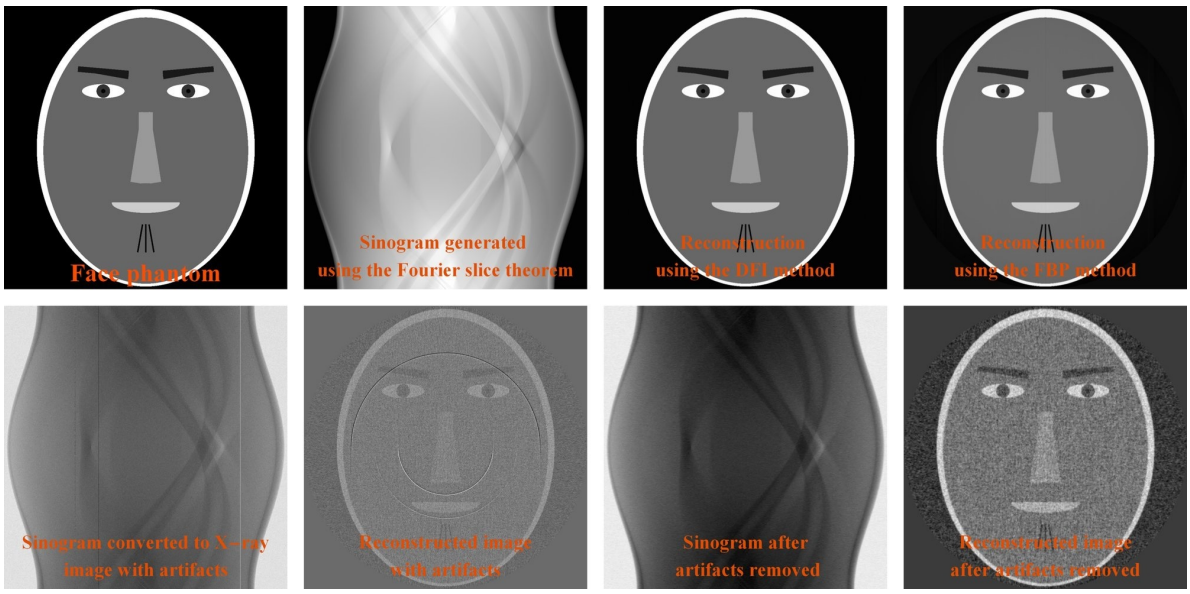
- Methods for processing helical scans (with/without the offset rotation-axis).



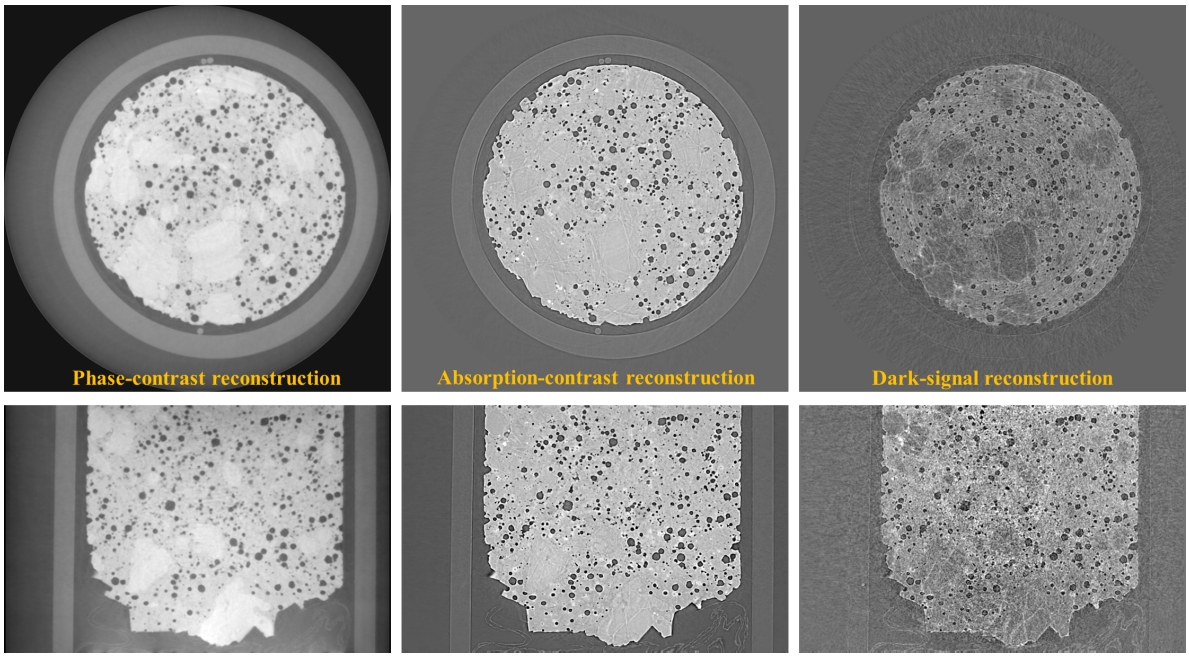
- Methods for determining the center-of-rotation (COR) and auto-stitching images in *half-acquisition scans* (360-degree acquisition with the offset COR).
- Some practical methods developed and implemented for the package: zinger removal, tilted sinogram generation, sinogram distortion correction, beam hardening correction, DFI (direct Fourier inversion) reconstruction, FBP reconstruction, and double-wedge filter for removing sample parts larger than the FOV in a sinogram.



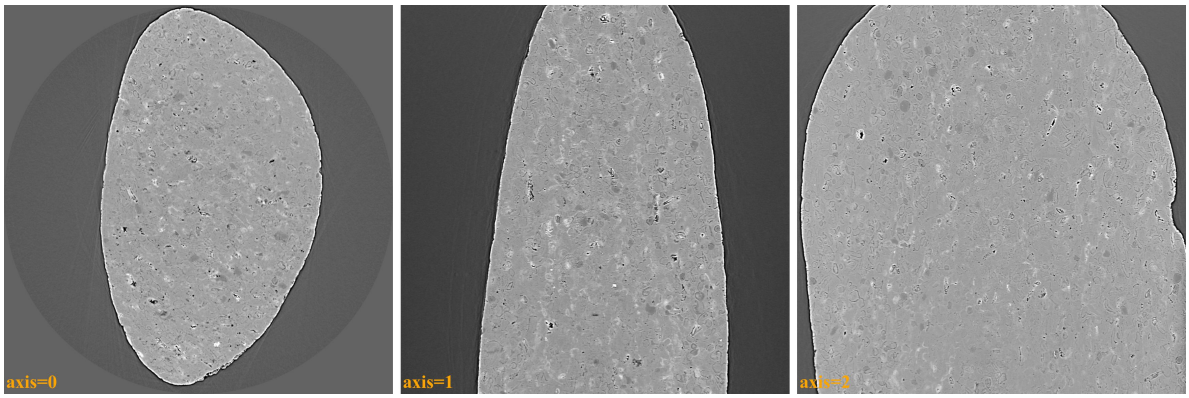
- Utility methods for customizing *ring/stripe artifact removal methods* and parallelizing computational work.
- Calibration methods for determining pixel-size in helical scans.
- Methods for generating simulation data: phantom creation, sinogram calculation based on the Fourier slice theorem, and artifact generation.



- Methods for phase-contrast imaging: phase unwrapping, *speckle-based phase retrieval*, image correlation, and image alignment.



- Methods for downsampling, rescaling, and reslicing (+rotating, cropping) 3D reconstructed image without large memory usage.

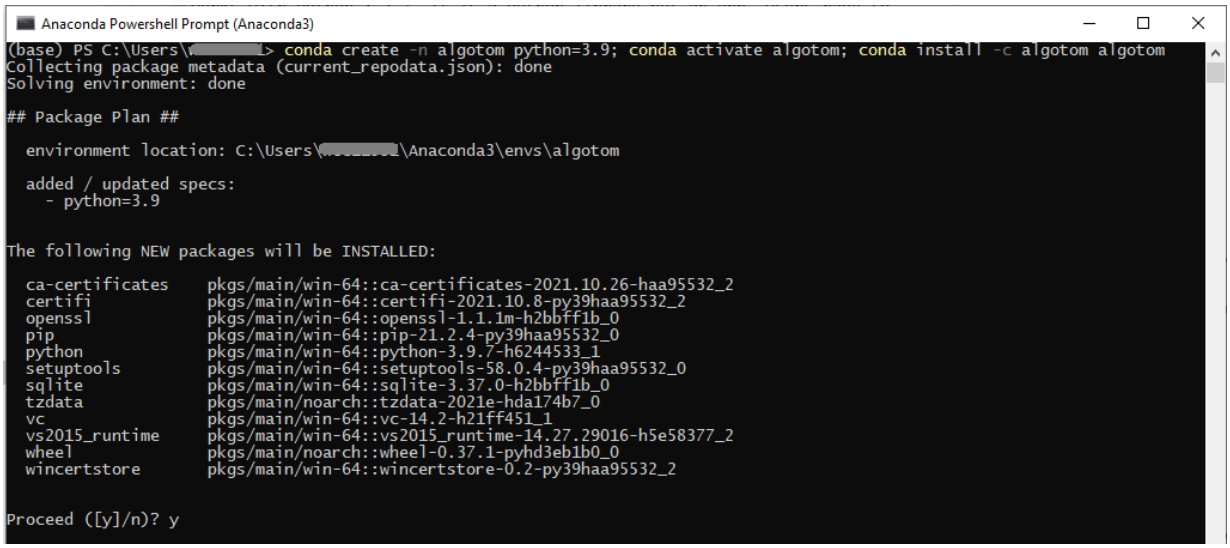


1.2.2 Development principles

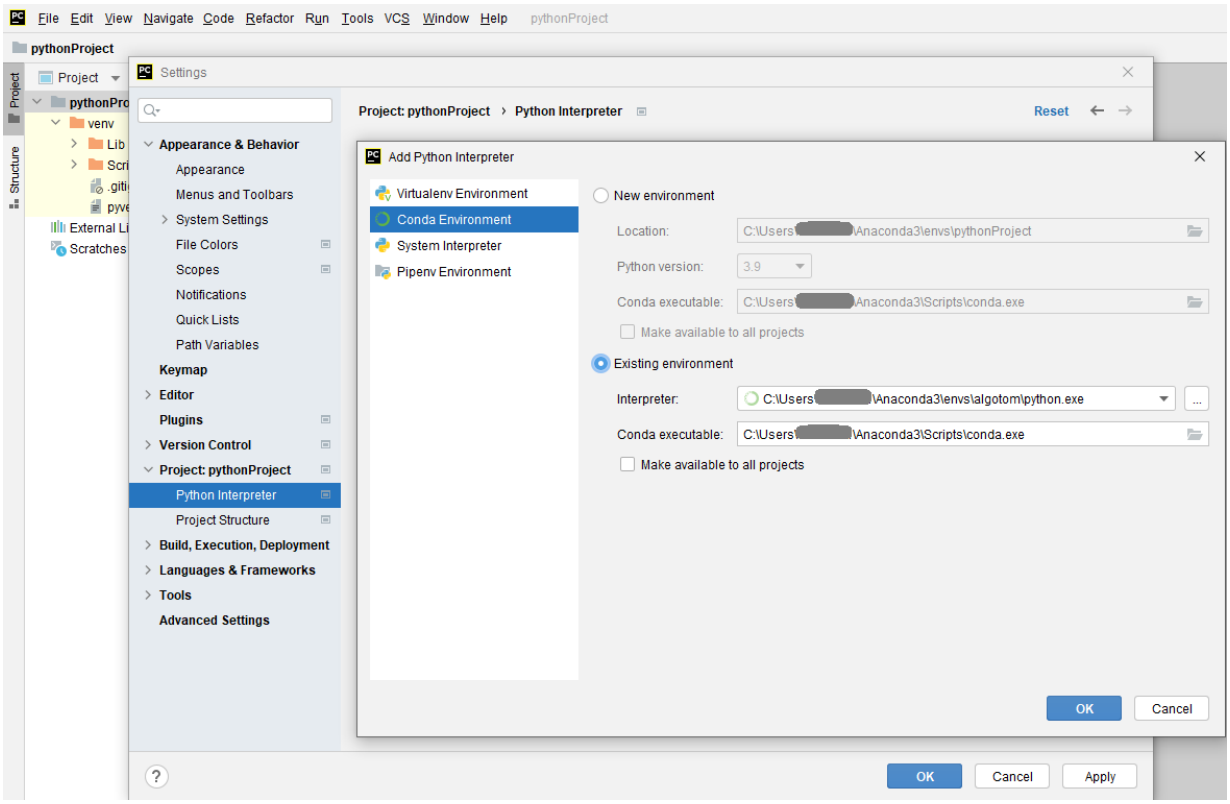
- While Algotom offers a comprehensive range of tools for tomographic data processing covering raw-data reading, pre-processing, reconstruction, post-processing, and data saving; its development primarily focuses on pre-processing techniques. This distinction makes it a prominent feature among other tomographic software.
- To ensure that the software can work across platforms and is easy-to-install; dependencies are minimized, and only well-maintained Python libraries are used.
- To achieve high-performance computing and leverage GPU utilization while ensuring ease of understanding, usage, and software maintenance, Numba is used instead of Cupy or PyCuda.
- Methods are structured into modules and functions rather than classes to enhance usability, debugging, and maintenance.
- Algotom is highly practical as it can run on computers with or without a GPU, multicore CPUs; and accommodates both small and large memory capacities.

1.3 Installation

Algotom is installable across operating systems (Windows, Linux, Mac) and works with Python >=3.7. It is a Python library not an app. Users have to write Python codes to process their data. For beginners, a quick way to get started with Python programming is to install Anaconda, then follow instructions here. There are many IDE software can be used to write and run Python codes e.g Spyder, Pydev, Pycharm (Community), or Visual Studio Code. After installing these software, users need to configure Python interpreter by pointing to the installed location of Anaconda. Each software has instructions of how to do that. There is a list of standard Python libraries shipped with Anaconda, known as the *base* environment. To install a Python package out of the list, it's a good practice that users should create a separate environment from the base. This tutorial gives an overview about Python environment. Instructions of how to create a new environment and how to install new packages are here and here. The following image shows the screenshot of how to use Anaconda Powershell Prompt to create a new environment and install Algotom.



Note that the IDE software needs to be reconfigured to point to the new environment as shown below.



If users don't want to install Anaconda which is quite heavy due to the base environment shipped with it, Miniconda is enough to customize Python environment.

1.3.1 Using conda

Install Miniconda as instructed above, then:

Open Linux terminal or Miniconda/Anaconda Powershell prompt and run the following commands:

If install to an existing environment:

```
conda install -c conda-forge algotom
```

or:

```
conda install -c algotom algotom
```

If install to a new environment:

```
conda create -n algotom python>=3.7
conda activate algotom
conda install -c conda-forge algotom
```

1.3.2 Using pip

Install Miniconda as instructed above, then

Open Linux terminal or Miniconda/Anaconda Powershell prompt and run the following commands:

If install to an existing environment:

```
pip install algotom
```

If install to a new environment:

```
conda create -n algotom python>=3.7
conda activate algotom
pip install algotom
```

1.3.3 From source

Clone Algotom from Github repository:

```
git clone https://github.com/algotom/algotom.git algotom
```

Download and install Miniconda software, then:

Open Linux terminal or Miniconda/Anaconda Powershell prompt and run the following commands:

```
conda create -n algotom python>=3.7
conda activate algotom
cd algotom
python setup.py install
```

1.3.4 Notes

To use GPU-enabled functions, users have to make sure that their computers have a NVIDIA GPU and must install CUDA Toolkit. Installing the latest version of CUDA Toolkit (or Python) is not recommended as scientific software often takes time to update.

To compromise between ease-of-installation and performance, GPU-enabled reconstruction functions in Algotom use Numba. Users can use other reconstruction methods; which are optimized for speed such as the gridding reconstruction method in Tomopy or GPU-enabled methods in Astra Toolbox; using Algotom's wrappers. Making sure that Tomopy and Astra Toolbox are installed before use. Referring to the websites of these packages to know how to install or acknowledge if you use them.

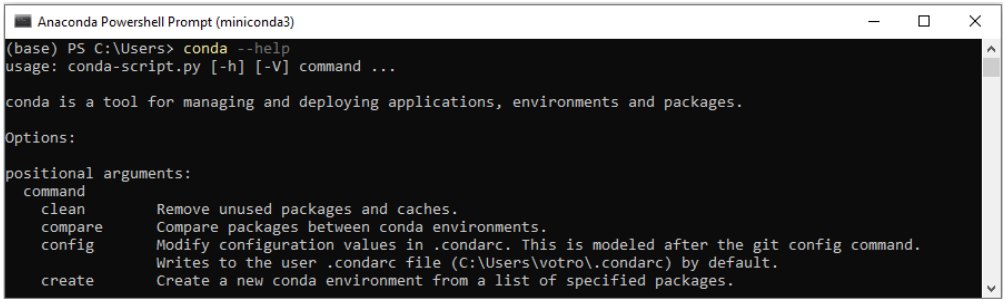
1.4 Demonstrations

1.4.1 Setting up a Python workspace

This section demonstrates step-by-step how to install Python libraries, software, and tools; i.e. setting up a workspace for coding; on WinOS to write Python codes and process tomographic data. There are many ways to set up a Python workspace. However, we only show approaches which are easy-to-follow and less troublesome for beginners.

1. Install Conda, a package manager, to install Python libraries

Download Miniconda from here and install it. After that, run Anaconda Powershell Prompt. This Powershell is a command-line interface where users can run commands to install/manage Python environments and packages.



There is a list of commands in Conda, but we just need a few of them. The first command is to create a new environment. An environment is a collection of Python packages. We should create different environments for different usages (such as to process tomographic data, write sphinx documentation, or develop a specific Python software...) to avoid the conflict between Python libraries. The following command will create an environment named *myspace*

```
conda create -n myspace
```

Then we must activate this environment before installing Python packages into it.

```
conda activate myspace
```

Name of the activated environment with be shown in the command line as below

First things first, we install Python. Here we specify Python 3.10 (or 3.11), not the latest one, as the Python ecosystem taking time to keep up.

```
conda install python=3.10
```

Then we install tomographic packages. A Python package can be distributed through its own channel, the conda-forge channel (a huge collection of Python packages), Pypi, or users can download the source codes and install themselves using *setup.py*. The order of priority should be: conda-forge, own channel, Pypi, then source codes. Let install the Algotom package first using the instruction shown on its documentation page.

```
conda install -c conda-forge algotom
```

Because Algotom relies on dependencies, e.g. Numpy, Numba, Scipy, H5py,... they are also installed at the same time. The Python environment and its packages are at *C:/Users/user_ID/miniconda3/envs/myspace*. Other *conda* commands are often used:

- *conda list* : list packages installed in an activated environment.
- *conda uninstall <package>* : to uninstall a package.
- *conda deactivate* : to deactivate a current environment
- *conda remove -n myspace –all* : delete an environment.
- *conda info -e* : list environments created.

2. Install tomography-related, image-processing packages

There are a few of tomography packages which users should install along with Algotom: Astra Toolbox and Tomopy

```
conda install -c astra-toolbox astra-toolbox=2.1.0

conda install -c conda-forge tomopy
```

For packages using Nvidia GPUs, making sure to install the CUDA toolkit as well. A popular visualization package, Matplotlib, is important to check or save results of a workflow.

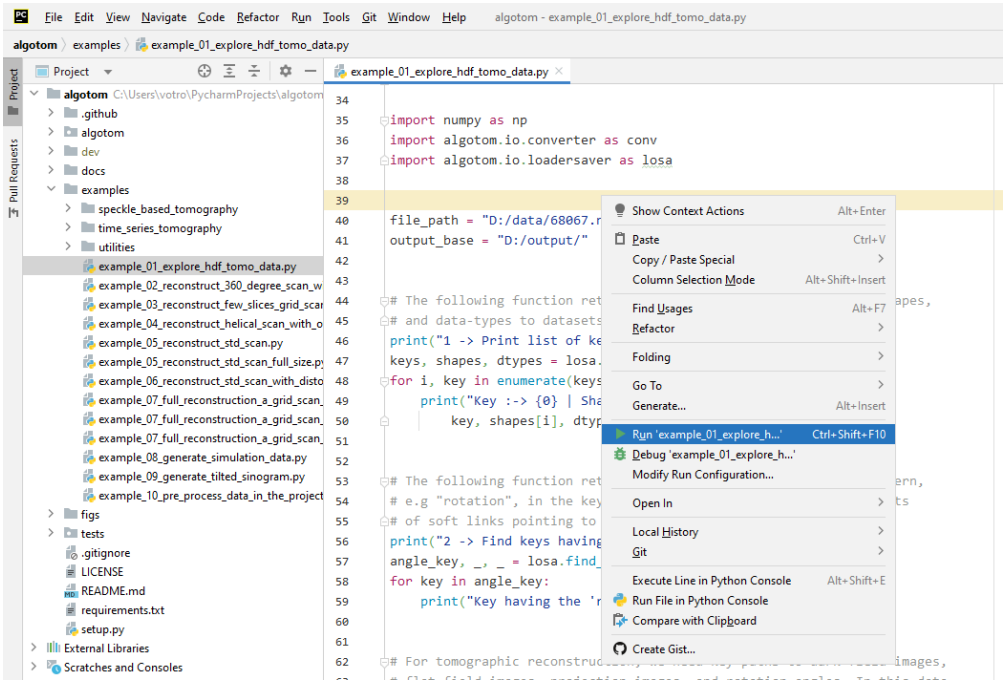
```
conda install -c conda-forge matplotlib
```

If users need to calculate distortion coefficients of a lens-based detector of a tomography system, using Discorpy

```
conda install -c conda-forge discorpy
```

3. Install Pycharm for writing and debugging Python codes

Pycharm is one of the most favorite IDE software for Python programming. It has many features which make it easy for coding such as syntax highlight, auto-completion, auto-format, auto-suggestion, typo check, version control, or change history. Pycharm (Community edition) is free software. After installing, users needs to configure the Python interpreter (File->Settings->Project->Python interpreter-> Add ->Conda environment) pointing to the created conda environment, *C:/Users/user_ID/miniconda3/envs/myspace*, as demonstrated in *section 1.1*. It's very easy to create a python file, write codes, and run them as shown below.



4. Write and run codes interactively using Jupyter Notebook (optional)

Using Python scripts is efficient and practical for processing multiple datasets. However, if users want to work with data interactively to define a workflow, Jupyter Notebook is a good choice.

Install Jupyter in the activated environment

```
conda install -c conda-forge jupyter
```

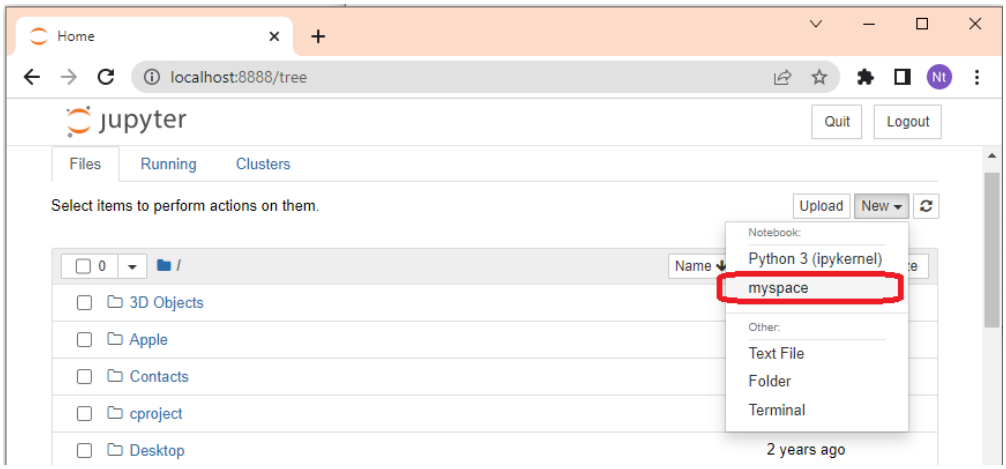
Run the following command to enable the current environment in notebook (only need for the first time setup). Note to change the name of the environment if users use a different name.

```
ipython kernel install --user --name="myspace"
```

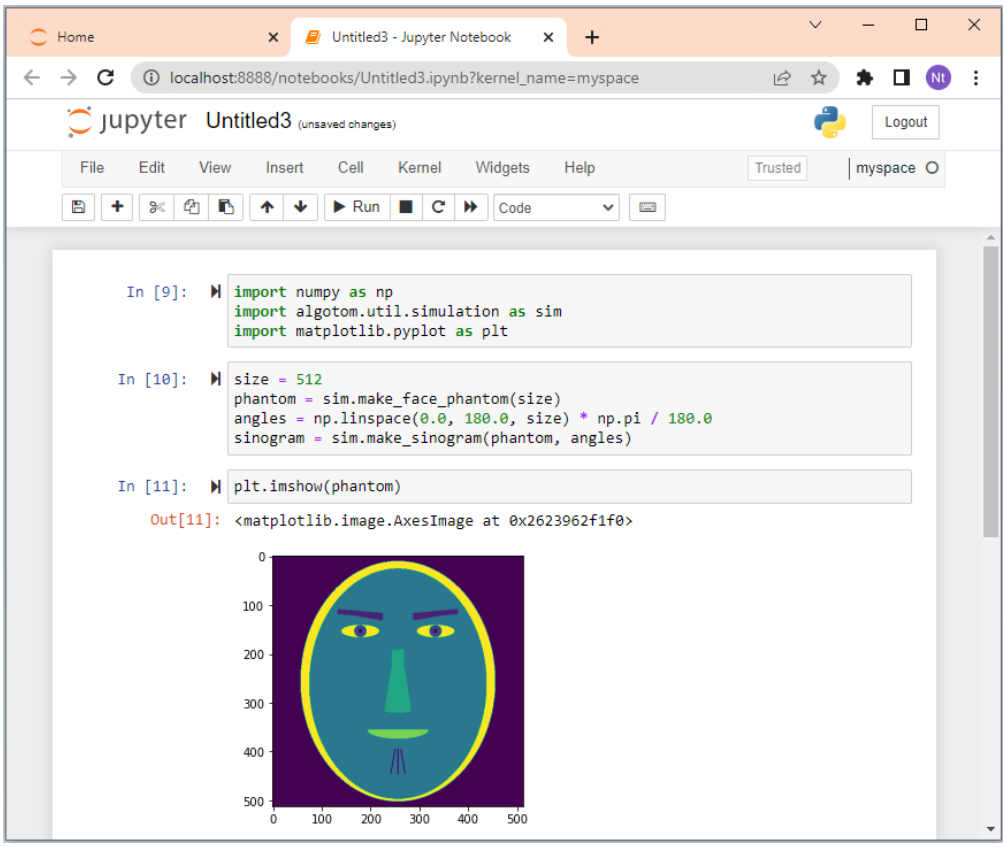
Then run Jupyter notebook by

```
jupyter notebook
```

Select the kernel as shown below



It will create a new tab for inputting codes



Note that the working folder (drive) of the notebook is where we run the command *jupyter notebook* from. If users want to work at a different drive, e.g. D:, they must navigate to that drive before running the notebook. (FYI, Press Ctrl+C to terminate a current running notebook from the Powershell Prompt)

```
cd D:
jupyter notebook
```

For who would like to use JupyterLab instead of Jupyter Notebook

Similar as above but for JupyterLab

```
conda install -c conda-forge jupyterlab
```

Run the following command only for the first time setup. Note to change the name of the environment if users use a different name.

```
ipython kernel install --user --name="myspace"
```

Then run JupyterLab by

```
jupyter lab
```

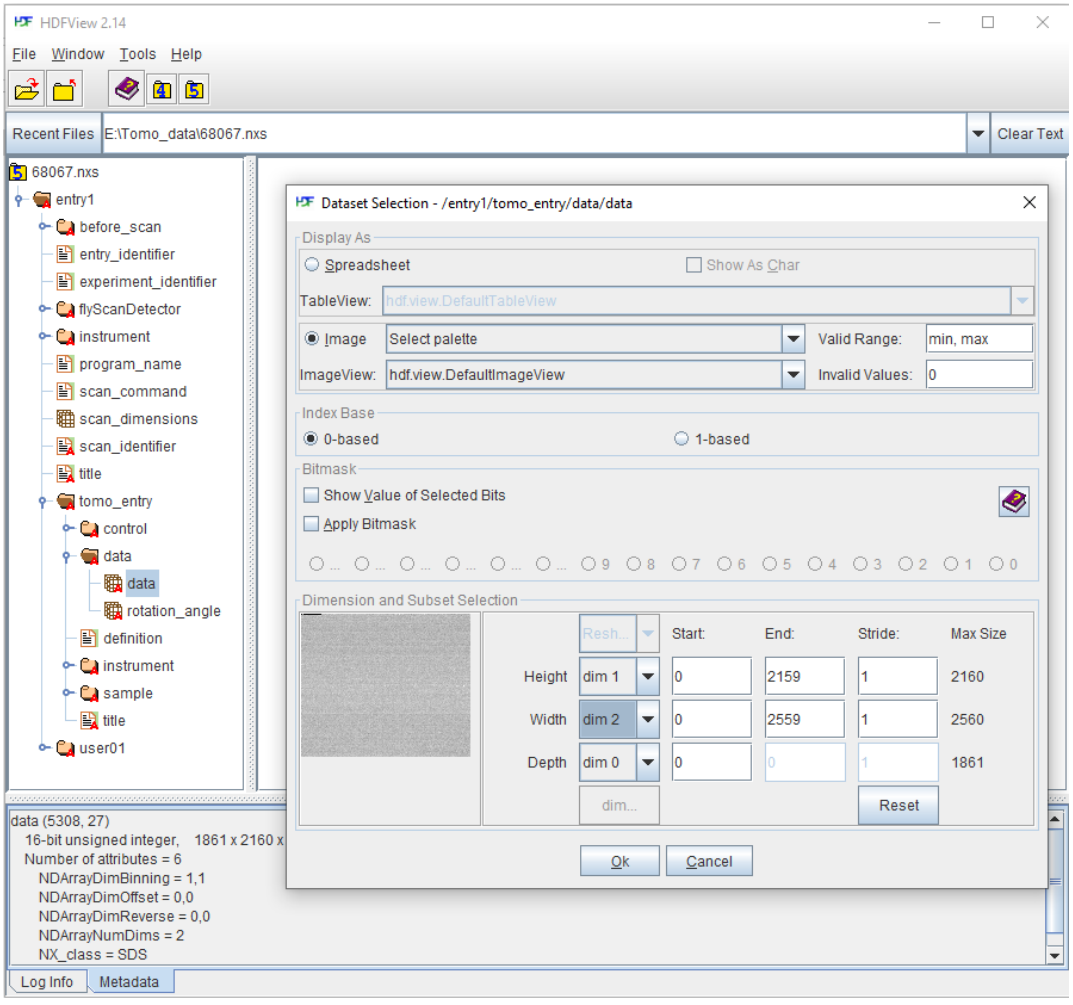
1.4.2 Exploring raw data and making use of the input-output module

The following sections show how to handle different types of raw data before they can be used for processing and reconstruction.

Nxs/hdf files

A nxs/hdf file can contain multiple datasets and data-types. Generally speaking, it likes a folder with many sub-folders and files inside (i.e. hierarchical format). To get data from a hdf file we need to know the path to the data. For example, we want to know the path to projection-images of this tomographic data. The data have two files: a hdf file which contains images recorded by a detector and a nxs file which contains the metadata of the experiment. The hdf file was linked to the nxs file at the time they were created, so we only need to work with the nxs file.

- Using Hdfview (version 2.14 is easy to install) we can find the path to image data is *"/entry1/tomo_entry/data/data"*. To display an image in that dataset: right click on "data" -> select "Open as" -> select "dim1" for "Height", select "dim2" for "Width" -> click "OK".



A metadata we need to know is rotation angles corresponding to the acquired images. The path to this data is *"/entry1/tomo_entry/data/rotation_angle"*. There are three types of images in a tomographic dataset: images with sample (projection), images without sample (flat-field or white field), and images taken with a photon source off (dark-field). In the data used for this demonstration, there's a metadata in *"/entry1/instrument/image_key/image_key"* used to indicate the type of an image: 0 <-> projection; 1 <-> flat-field;

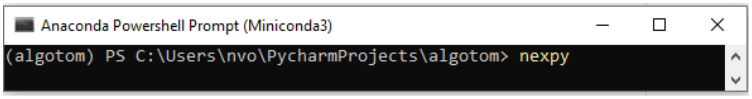
2 <-> dark-field.

Different tomography facilities name above datasets differently. Some names rotation angles as "theta_angle". Some record flat-field and dark-field images as separate datasets (Fig. 1.1.22). There has been an effort to unify these terms for synchrotron-based tomography community. This will be very userful for end-users where they can use the same codes for processing data acquired at different facilities.

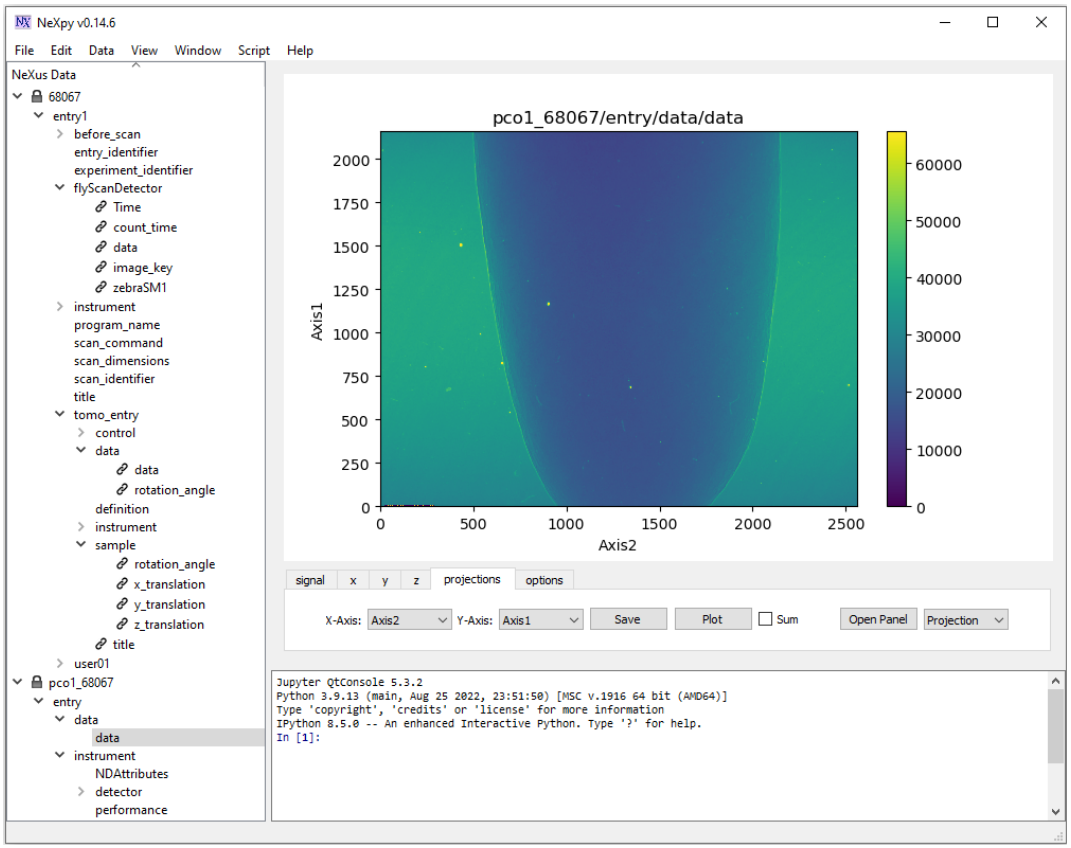
- Other way of exploring nxs/hdf files is to use NeXpy. Users need to install NeXpy in an activated *environment*.

```
conda install -c conda-forge nexpy
```

and run from that environment



NeXpy provides more options to explore data. Noting that image in NeXpy is displayed with the origin at the bottom left. This is different to Hdfview (Fig. 1.1.23).



Other python-based GUI software can be used are: Broh5 or Vitables.

- Users also can use functions in the *input-output module* of Algotom to explore data. For example, to display the hierarchical structure of a hdf file:

```
import algotom.io.loadersaver as losa

file_path = "E:/Tomo_data/68067.nxs"
losa.get_hdf_tree(file_path)
```

```
Output:
entry1
    |
    |-- before_scan
    |   |
    |   |-- cam1
    |   |   |
    |   |   |-- cam1_roll (1,)
    |   |   |-- cam1_x (1,)
    |   |   |-- cam1_z (1,)
    |   |-- dcm1_cap_1
    |   |   |-- dcm1_cap_1 (1,)
```

To find datasets having the pattern of "data" in their paths:

```
keys, shapes, types = losa.find_hdf_key(file_path, "data")
for i in range(len(keys)):
    print(i," Key: {0} | Shape: {1} | Type: {2} ".format(keys[i], shapes[i],
↪ types[i]))
```

```
Output:
0  Key: entry1/flyScanDetector/data | Shape: (1861, 2160, 2560) | Type:␣
↪uint16
1  Key: entry1/instrument/flyScanDetector/data | Shape: (1861, 2160, 2560)␣
↪| Type: uint16
2  Key: entry1/tomo_entry/data | Shape: None | Type: None
3  Key: entry1/tomo_entry/control/data | Shape: (1,) | Type: float64
4  Key: entry1/tomo_entry/data/data | Shape: (1861, 2160, 2560) | Type:␣
↪uint16
5  Key: entry1/tomo_entry/data/rotation_angle | Shape: (1861,) | Type:␣
↪float64
6  Key: entry1/tomo_entry/instrument/detector/data | Shape: (1861, 2160,␣
↪2560) | Type: uint16
```

After knowing the path (key) to a dataset containing images we can extract an image and save it as tif. A convenient feature of methods for saving data in Algotom is that if the output folder doesn't exist it will be created.

```
image_data = losa.load_hdf(file_path, "entry1/tomo_entry/data/data")
losa.save_image("E:/output/image_00100.tif", image_data[100])
```

We also can extract multiple images from a hdf file and save them to tiff using a single command

```
import algotom.io.converter as conv

# Extract images with the indices of (start, stop, step) along axis 0
conv.extract_tif_from_hdf(file_path, "E:/output/some_proj/", "entry1/tomo_
↪entry/data/data",
                          index=(0, -1, 100), axis=0, crop=(0, 0, 0, 0),␣
↪prefix='proj')
```

Tiff files

In some tomography systems, raw data are saved as tiff images. As shown in *section 2*, processing methods for tomographic data work either on projection space or sinogram space, or on both. Because of that, we have to switch between spaces, i.e. slicing 3D data along different axis. This cannot be done efficiently if using the tiff format. In such case, users can convert tiff images to the hdf format first before processing them with options to add metadata.

```
input_folder = "E:/raw_tif/" # Folder with tiff files inside. Note that the␣
↪names of the
                             # tiff files must be corresponding to the␣
↪increasing order of angles
output_file = "E:/convert_hdf/tomo_data.hdf"
num_angle = len(losa.file_file(input_folder + "/*tif*"))
angles = np.linspace(0.0, 180.0, num_angle)
conv.convert_tif_to_hdf(input_folder, output_file, key_path='entry/data',
                        crop=(0, 0, 0, 0), pattern=None,
                        options={"entry/angles": angles, "entry/energy_keV":␣
↪20})
```

In some cases, we may want to load a stack of tiff images and average them such as flat-field images or dark-field images. This can be done in different ways

```
input_folder = "E:/flat_field/"
# 1st way
flat_field = np.mean(losa.get_tif_stack(input_folder, idx=None, crop=(0, 0, 0,␣
↪0)), axis=0)
# 2nd way. The method was written for speckle-tracking tomography but can be␣
↪used here
flat_field = losa.get_image_stack(None, input_folder, average=True, crop=(0, 0,
↪ 0, 0))
# 3rd way
list_file = losa.find_file(input_folder + "/*tif*")
flat_field = np.mean(np.asarray([losa.load_image(file) for file in list_file]),
↪ axis=0)
```

Mrc files

Mrc format is a standard format in electron tomography. To load this data, users need to install the Mrcfile library

```
conda install -c conda-forge mrcfile
```

and check the documentation page to know how to extract data and metadata from this format. For large files, we use memory-mapped mode to read only part of data needed as shown below.

```
import mrcfile
import algotom.io.loadersaver as losa

mrc = mrcfile.mmap("E:/etomo/tomo.mrc", mode='r+')
output_base = "E:/output"
(depth, height, width) = mrc.data.shape
for i in range(0, depth, 10):
    name = "0000" + str(i)
    losa.save_image(output_base + "/img_" + name[-5:] + ".tif", mrc.data[i])
```

Methods in Algotom assume that the rotation axis of a tomographic data is parallel to the columns of an image. Users may need to rotate images loaded from a mrc file because the rotation axis is often parallel to image-rows instead.

Other file formats

For other file formats such as xrm, txrm, fits, ... users can use the DXchange library to load data

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

and refer the documentation page for more details.

1.4.3 Methods and tools for removing ring artifacts

Algotom provides improved implementations of many methods for removing ring artifacts; which were published previously by the same author in Sarepy; to be easier to use and customize. More than that, there are many tools for users to design their own removal methods.

Note that ring artifacts in a reconstructed image are corresponding to stripe artifacts in the sinogram image or the polar-transformed image. Most of ring removal methods are actually stripe removal methods under the surface.

Improvements

- Users can select different smoothing filters available in Scipy or in *Algotom utility module* for removing stripes by passing keyword arguments as dict type:

```
import algotom.io.loadersaver as losa
import algotom.prep.removal as rem
sinogram = losa.load_image("D:/data/sinogram.tif")
# Sorting-based methods use the median filter by default, users can select
# another filter as below.
sinogram1 = rem.remove_stripe_based_sorting(sinogram, option={"method": "gaussian_
↪filter",
                                                              "para1": (1, 21)})
```

- The sorting-based technique, which is simple but effective to remove partial stripes and avoid void-center artifacts, is an option for other ring removal methods.

```
sinogram2 = rem.remove_stripe_based_filtering(sinogram, 3, sort=True)
sinogram3 = rem.remove_stripe_based_regularization(sinogram, 0.005, sort=True)
```

Tools for designing ring removal methods

The cleaning capability with least side-effect of a ring removal method relies on a smoothing filter or an interpolation technique which the method employs. Other supporting techniques for revealing stripe artifacts such as sorting, filtering, fitting, wavelet decomposition, polar transformation, or forward projection are commonly used. Algotom provides these supporting tools for users to incorporate with their own smoothing filters or interpolation techniques.

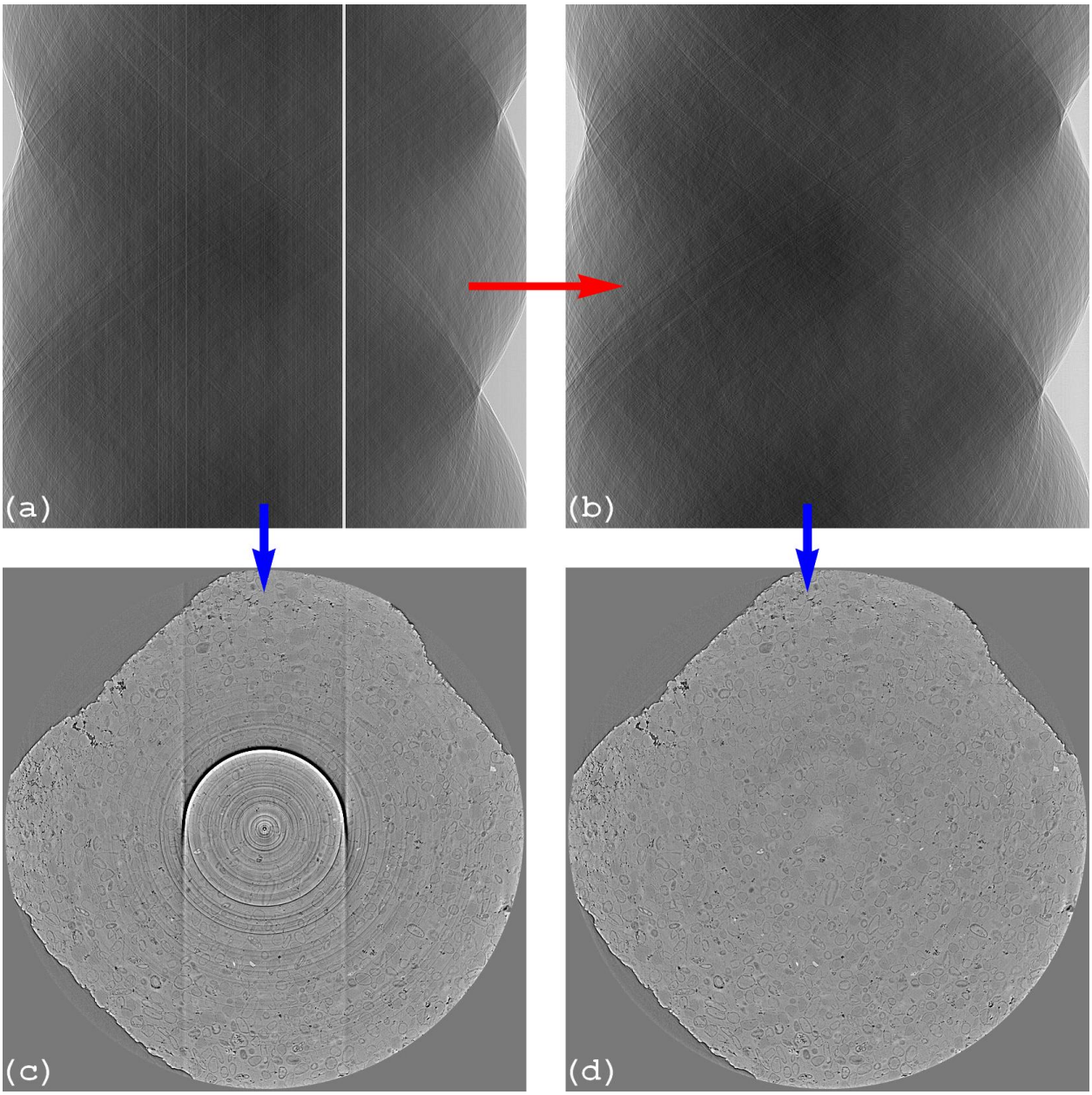


Fig. 1.4.1: Ring removal methods working on sinogram image, known as pre-processing methods. (a) Sinogram before correction. (b) Sinogram after correction. (c) Reconstructed image from sinogram (a). (d) Reconstructed image from sinogram (b).

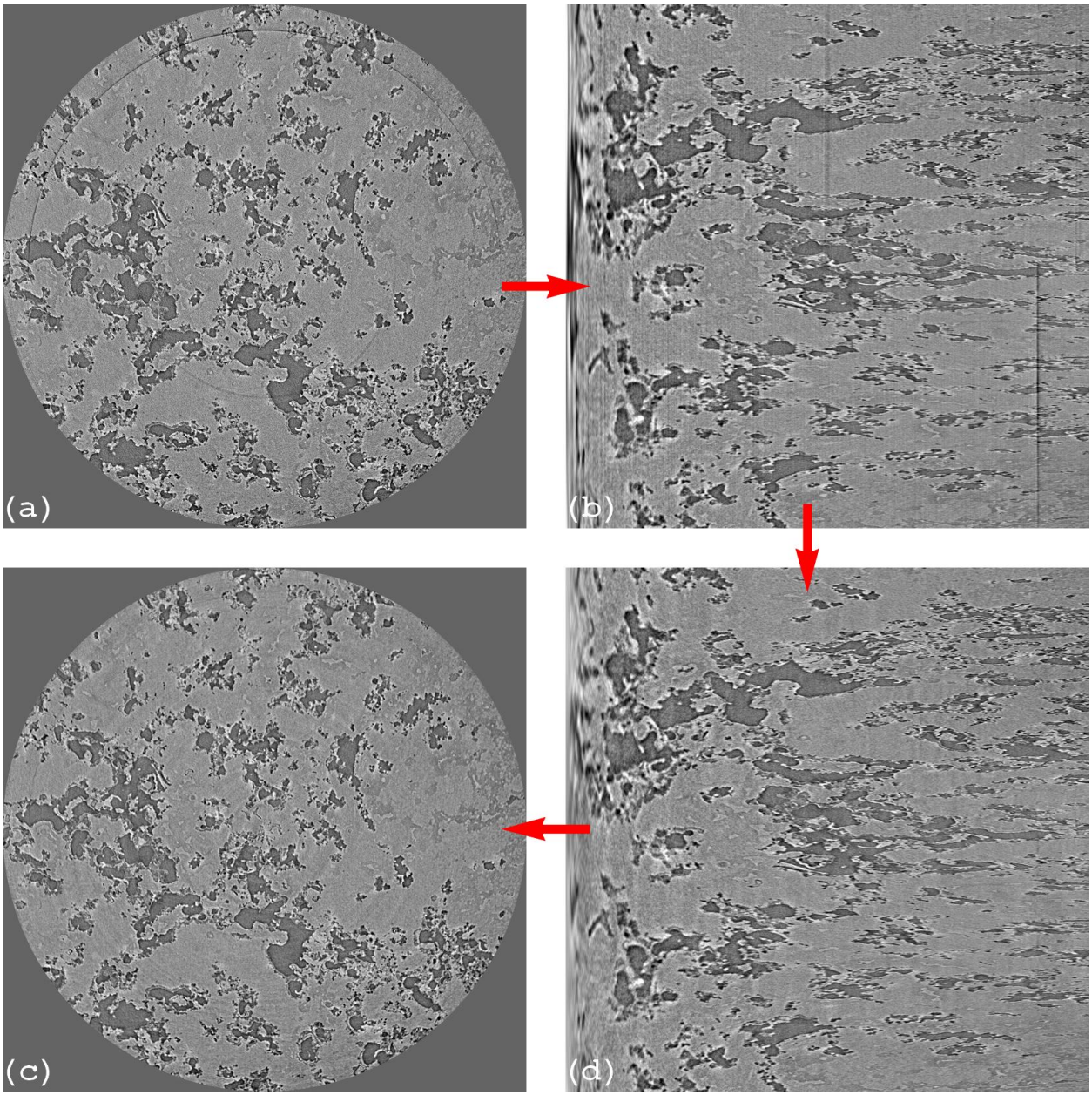


Fig. 1.4.2: Ring removal methods working on polar-transformed image, known as post-processing methods. (a) Reconstructed image before correction. (b) Polar transformation of image (a). (d) Stripe artifacts removed from image (b). (c) Cartesian transformation of image (d).

Back-and-forth sorting

The technique (algorithm 3 in [R19]) couples an image with an index array for sorting the image backward and forward along an axis. Users can combine the sorting forward method, a customized filter, and the sorting backward method as follows

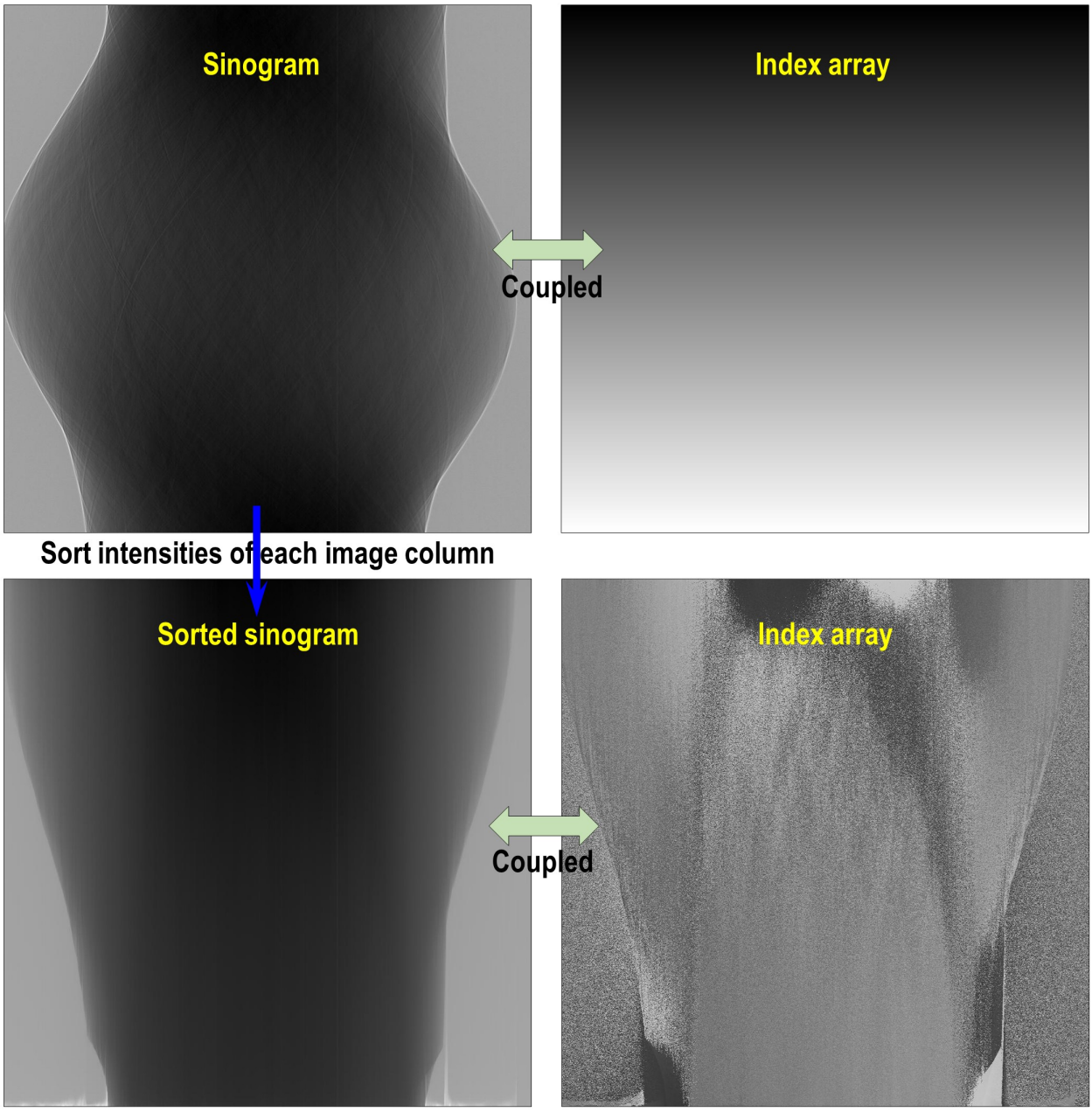


Fig. 1.4.3: Demonstration of the forward sorting.

```
import algotom.util.utility as util
import scipy.ndimage as ndi

# Sort forward
```

(continues on next page)

(continued from previous page)

```
sino_sort, mat_index = util.sort_forward(sinogram, axis=0)
# Use a customized smoothing filter here
sino_sort = apply_customized_filter(sino_sort, parameters)
# Sort backward
sino_corr = util.sort_backward(sino_sort, mat_index, axis=0)
```

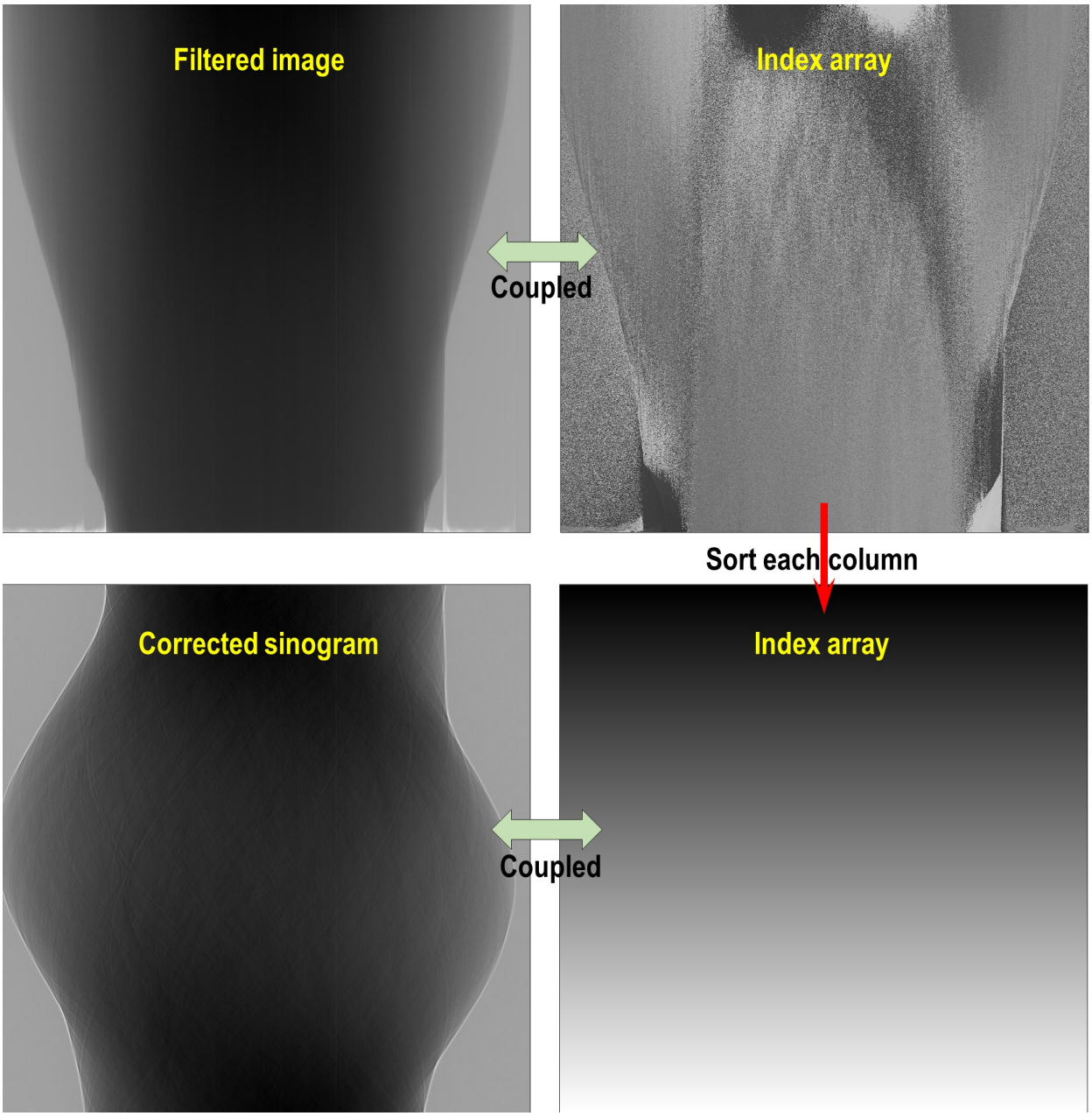


Fig. 1.4.4: Demonstration of the backward sorting.

Separation of frequency components

The technique can help to reveal stripe artifacts by separating frequency components of each image-column using a 1D window available in Scipy. Example of how to use the technique:

```
# Separate a sinogram image
sino_smooth, sino_sharp = util.separate_frequency_component(sinogram, axis=0,
                                                            window={"name":
↪"gaussian",
                                                                    "sigma": 5}
↪)
# Use a customized smoothing filter here
sino_smooth_filtered = apply_customized_filter(sino_smooth, parameters)
# Add back
sino_corr = sino_smooth_filtered + sino_sharp
```

Polynomial fitting along an axis

The technique can help to reveal low contrast stripes easily by applying a polynomial fit along each image-column.

```
sino_fit = util.generate_fitted_image(sinogram, 3, axis=0, num_chunk=1)
# Use a customized smoothing filter here
sino_smooth = apply_customized_filter(sino_fit, parameters)
# Get back the sinogram
sino_corr = (sinogram / sino_fit) * sino_smooth
```

Wavelet decomposition and reconstruction

Functions for wavelet decomposition, wavelet reconstruction, and applying a smoothing filter to specific levels of directional image-details are provided. The following codes decompose a sinogram to level 2. As can be seen in Fig. 1.4.7 stripe artifacts are visible in vertical details of results. One can apply a smoothing filter to remove these stripes then apply a wavelet reconstruction to get the resulting sinogram.

```
outputs = util.apply_wavelet_decomposition(sinogram, "db9", level=2)
[mat_2, (cH_level_2, cV_level_2, cD_level_2), (cH_level_1, cV_level_1, cD_
↪level_1)] = outputs
# Save results of vertical details
# losa.save_image("D:/output/cV_level_2.tif", cV_level_2)
# losa.save_image("D:/output/cV_level_1.tif", cV_level_1)

# Apply the gaussian filter to each level of vertical details
outputs = util.apply_filter_to_wavelet_component(outputs, level=None, order=1,
                                                 method="gaussian_filter",␣
↪para=[(1, 11)])
# Optional: remove stripes on the approximation image (mat_2 above)
outputs[0] = rem.remove_stripe_based_sorting(outputs[0], 11)
# Apply the wavelet reconstruction
sino_corr = util.apply_wavelet_reconstruction(outputs, "db9")
```

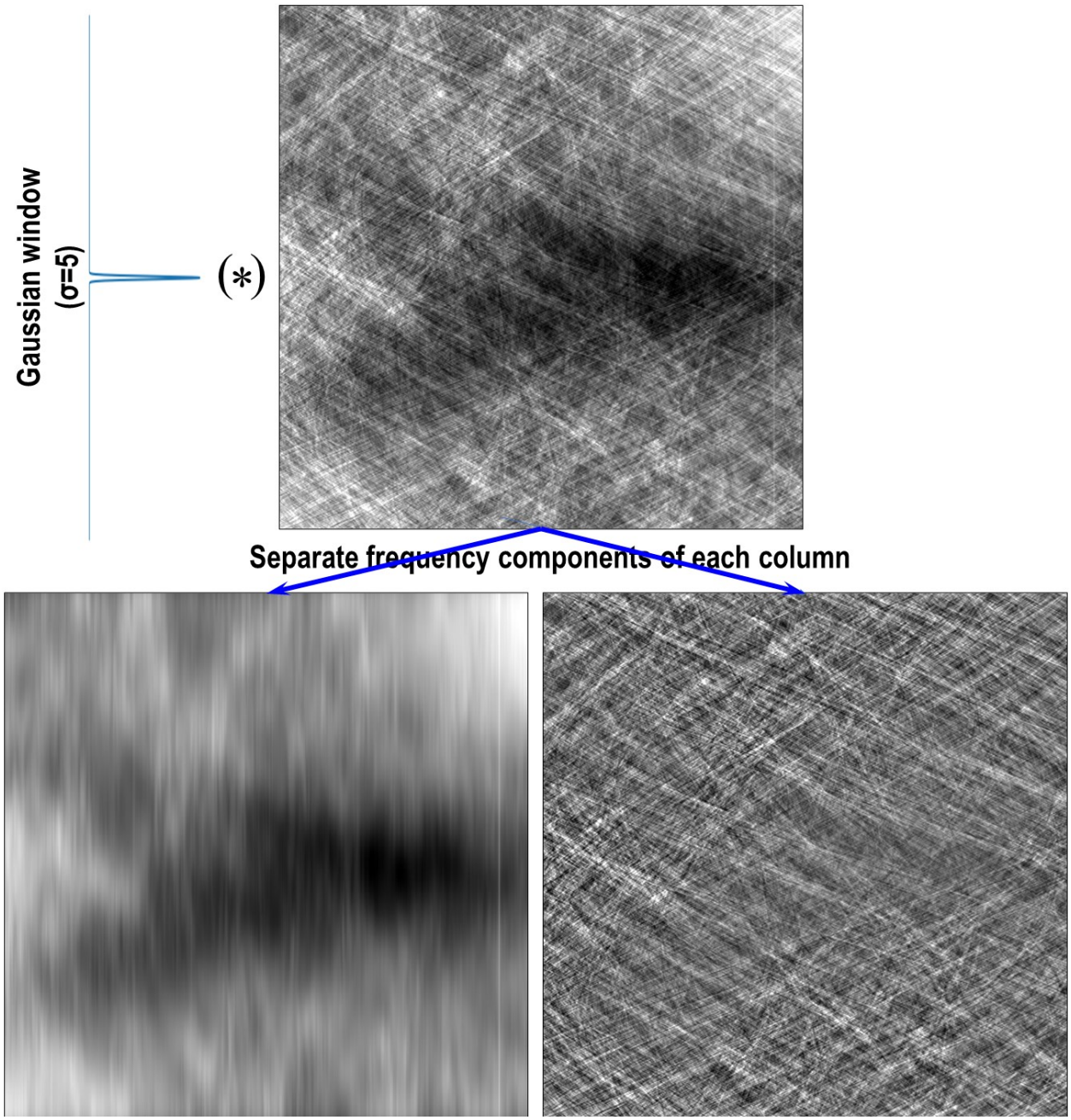


Fig. 1.4.5: Demonstration of how to separate frequency components of a sinogram along each column.

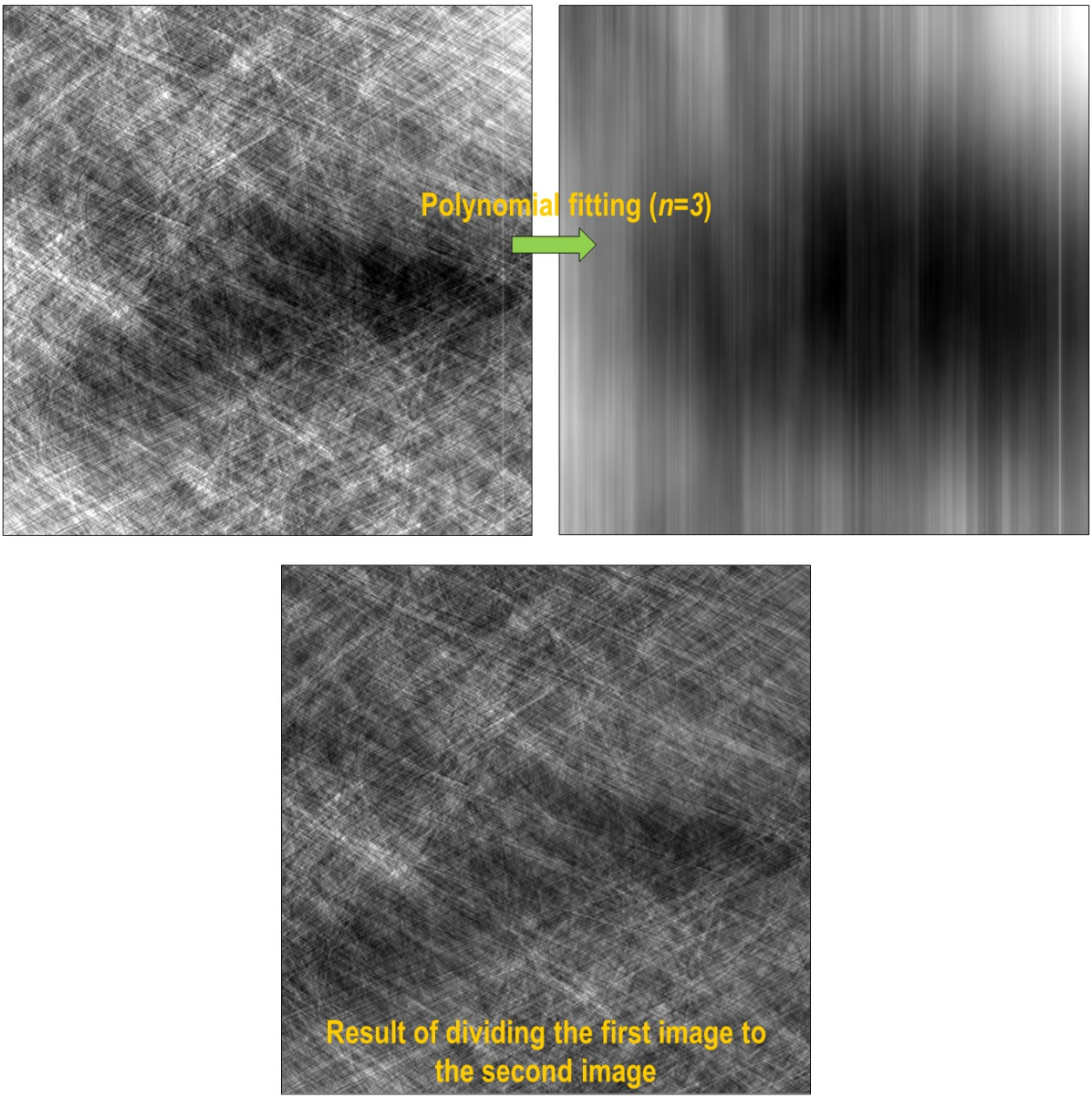


Fig. 1.4.6: Demonstration of how to apply a polynomial fitting along each column of a sinogram.

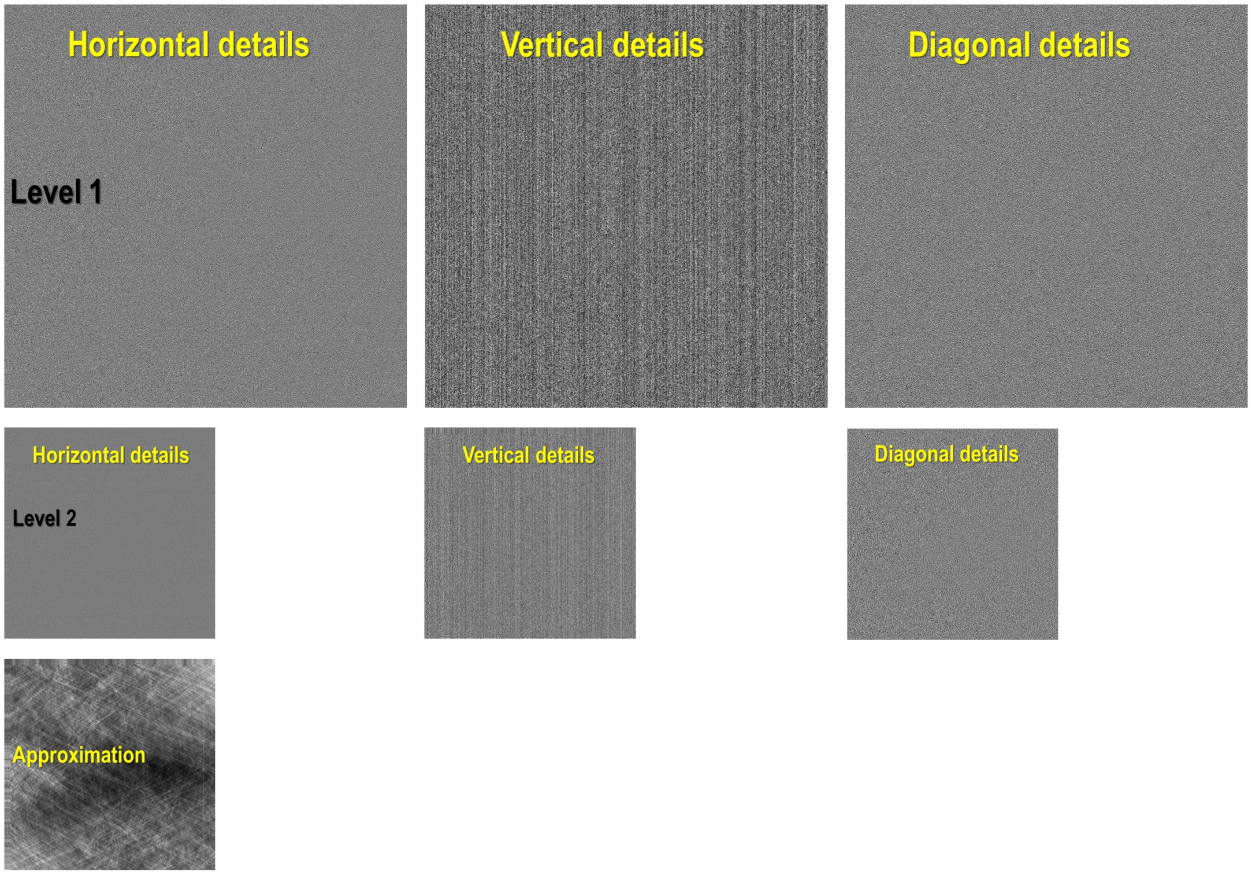


Fig. 1.4.7: Demonstration of the wavelet decomposition.

Stripe interpolation

Users can design a customized stripe-detection method, then pass the result (as a 1D binary array) to the following function to remove stripes by interpolation.

```
sino_corr = util.interpolate_inside_stripe(sinogram, list_mask, kind="linear")
```

Transformation between Cartesian and polar coordinate system

This is a well-known technique to remove ring artifacts from a reconstructed image as shown in Fig. 1.4.2.

```
img_rec = losa.load_image("D:/data/reconstructed_image.tif")
# Transform the reconstructed image into polar coordinates
img_polar = util.transform_slice_forward(img_rec)

# Use a customized smoothing filter here
img_corr = apply_customized_filter(img_polar, parameters)

# Transform the resulting image into Cartesian coordinates
img_carte = util.transform_slice_backward(img_corr)
```

Transformation between sinogram space and reconstruction space

Algotom provides a re-projection method to convert a reconstructed image to the sinogram image. As using directly the Fourier slice theorem it's fast compared to ray-tracing-based methods or image-rotation-based methods.

```
import numpy as np
import algotom.util.simulation as sim
import algotom.rec.reconstruction as rec

rec_img = losa.load_image("D:/data/reconstructed_image.tif")
(height, width) = rec_img.shape
angles = np.deg2rad(np.linspace(0.0, 180.0, height))

# Re-project the reconstructed image
sino_calc = sim.make_sinogram(rec_img, angles=angles)

# Use a customized stripe-removal method
sino_corr = apply_customized_filter(sino_calc, parameters)

# Reconstruct
img_rec = rec.dfi_reconstruction(sino_corr, (width - 1) / 2, apply_log=False)
```

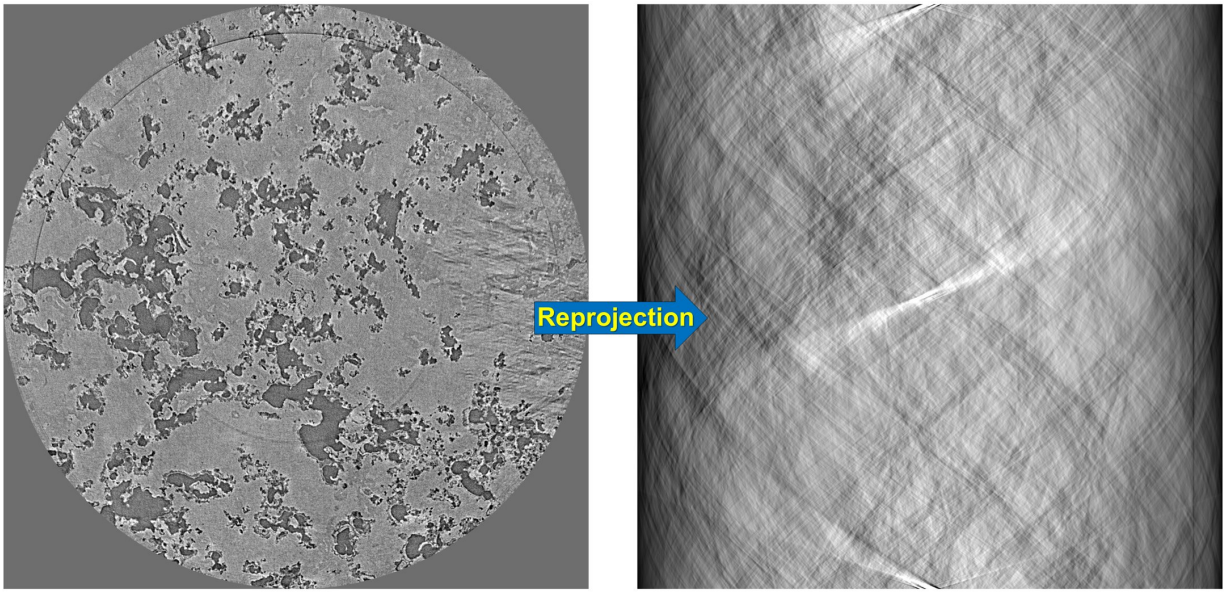


Fig. 1.4.8: Demonstration of how to re-project a reconstructed image.

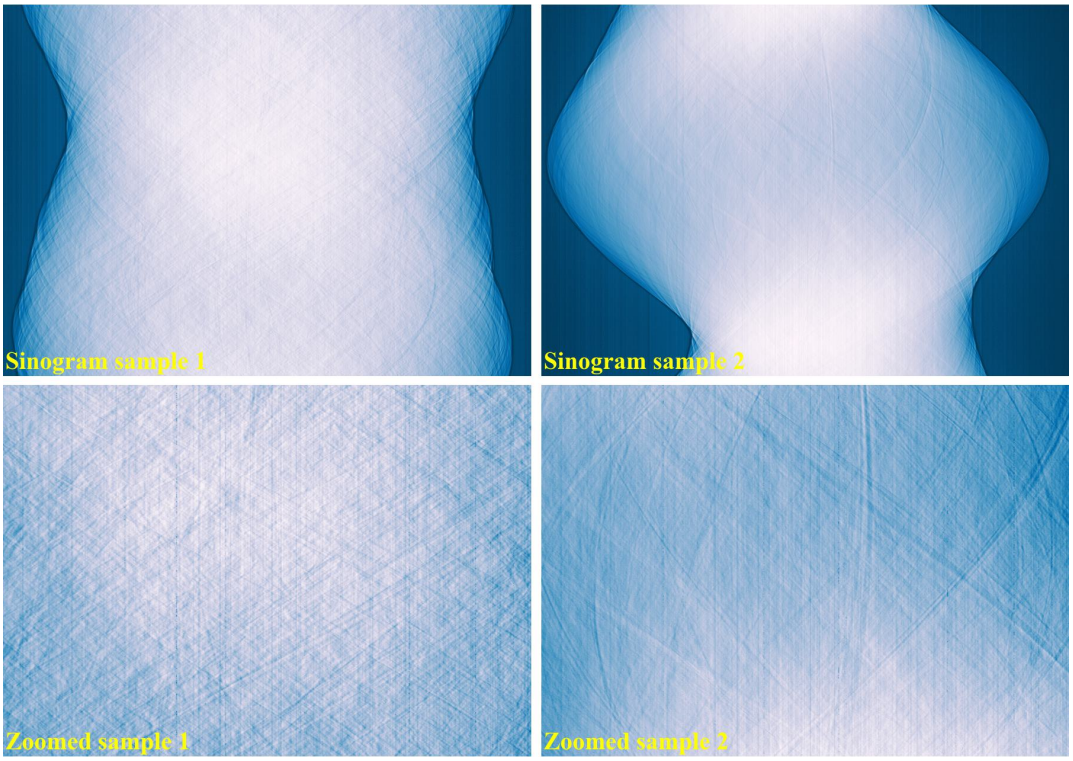
1.4.4 Comparison of ring removal methods on challenging sinograms

Ring artifact is the most pervasive type of artifacts in tomographic imaging. Numerous approaches for removing this artifact have been published over the years. In [R19], the author proposed many algorithms and a combination of them (algorithm 6, 5, 4, and 3) to remove most types of ring artifacts. This combined method, called **algo-6543** for short, is easy-to-use and very effective. It has been implemented in Python, Matlab, and available in several tomographic Python packages. To know more about causes of ring artifacts, types of ring artifacts, and details of removal algorithms out of the original paper; users can check out the documentation page here. This section demonstrates the performance of the algo-6543 method and sorting-based methods in comparison with other methods on challenging sinograms. These data are available here and free to use. They are very useful for testing ring removal methods.

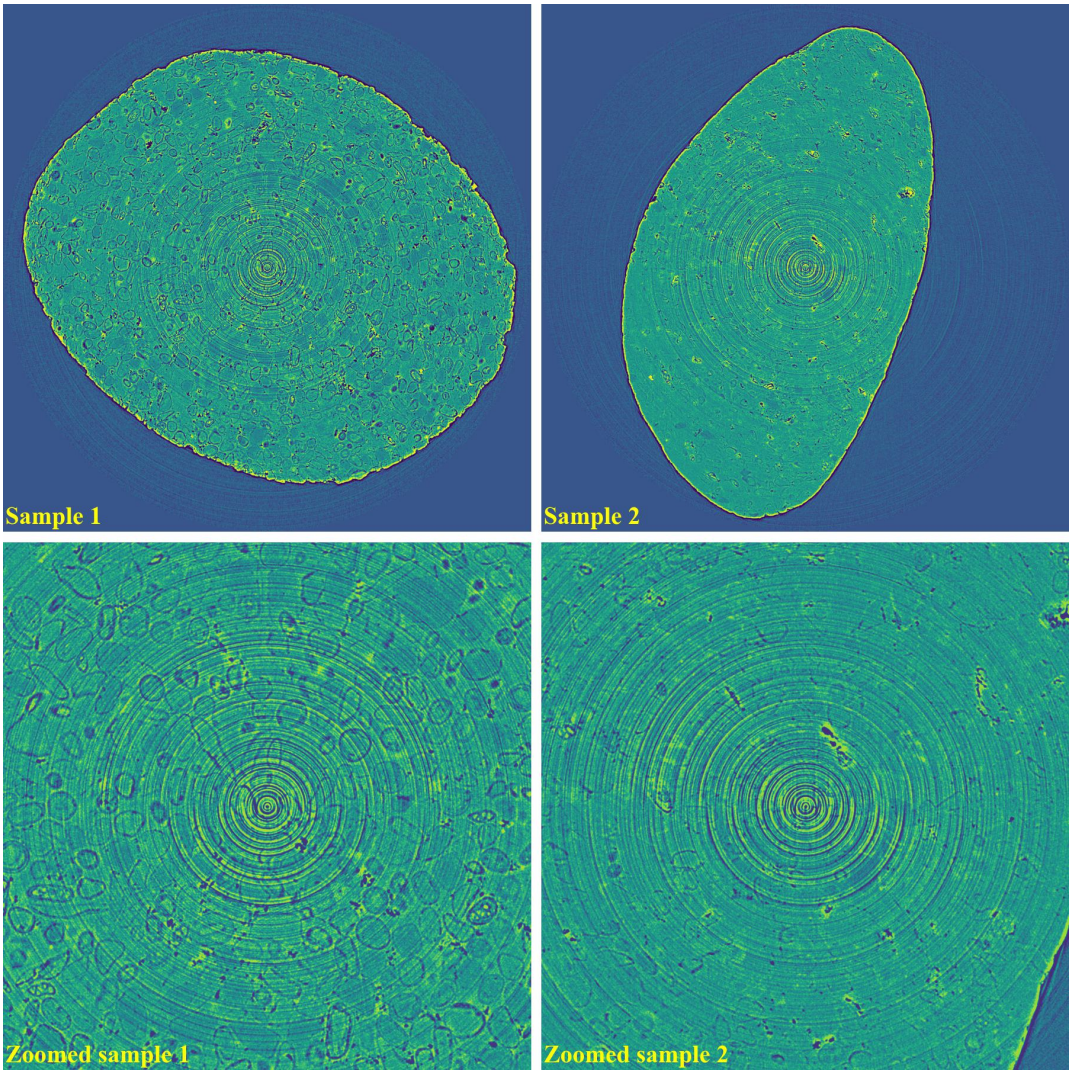
Same sample-type and slice but different in shape

The following images show sinograms and reconstructed images of two limestone rocks with different shapes before and after ring removal methods are applied.

- Sinograms at the same detector-row:



- Reconstructed images without using a ring removal method:



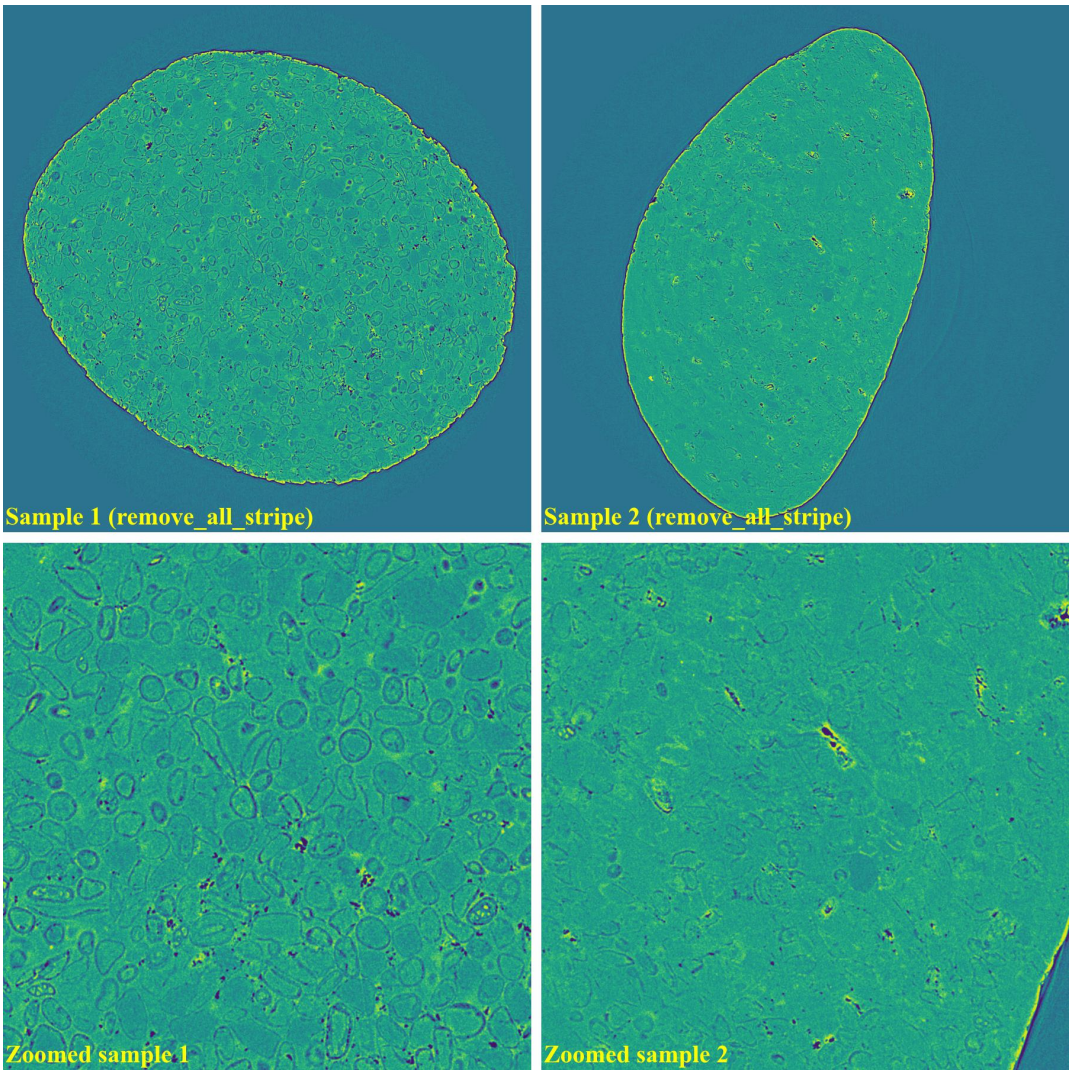
- If using the combination of methods:

```
import algotom.io.loadersaver as losa
import algotom.prep.calculation as calc
import algotom.prep.removal as rem
import algotom.rec.reconstruction as rec

input_base = "E:/data/"
output_base = "E:/rings_removed/remove_all_stripe/"

sinogram1 = losa.load_image(input_base + "/same_type_same_slice_different_
↪shape_sample1.tif")
sinogram2 = losa.load_image(input_base + "/same_type_same_slice_different_
↪shape_sample2.tif")
center1 = calc.find_center_vo(sinogram1)
center2 = calc.find_center_vo(sinogram2)

sinogram1 = rem.remove_all_stripe(sinogram1, snr=3.0, la_size=51, sm_
↪size=21)
```

(continues on next page)

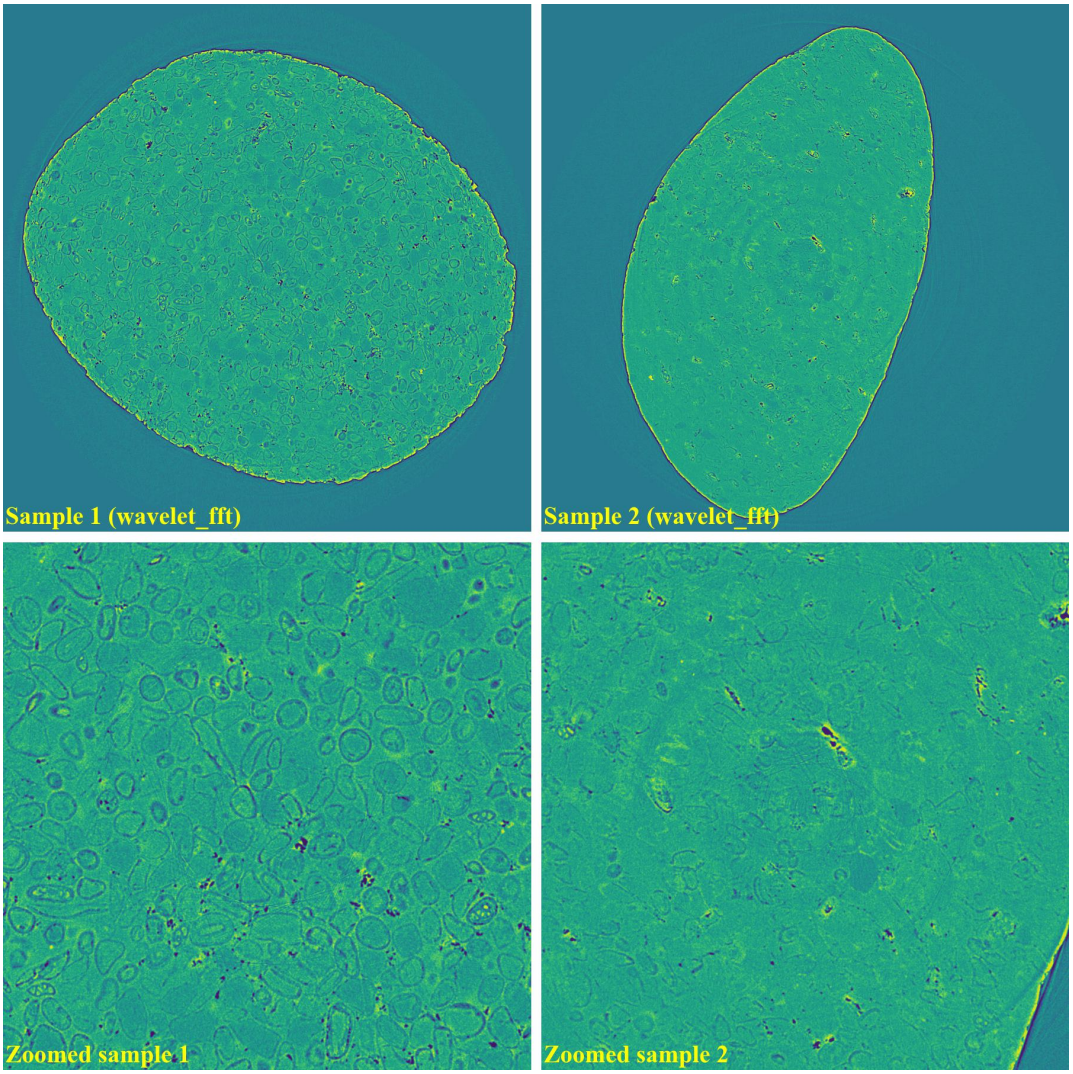
(continued from previous page)

```
sinogram2 = rem.remove_all_stripe(sinogram2, snr=3.0, la_size=51, sm_
↪size=21)

img_rec1 = rec.dfi_reconstruction(sinogram1, center1)
img_rec2 = rec.dfi_reconstruction(sinogram2, center2)
losa.save_image(output_base + "/rec_sample1.tif", img_rec1)
losa.save_image(output_base + "/rec_sample2.tif", img_rec2)
```



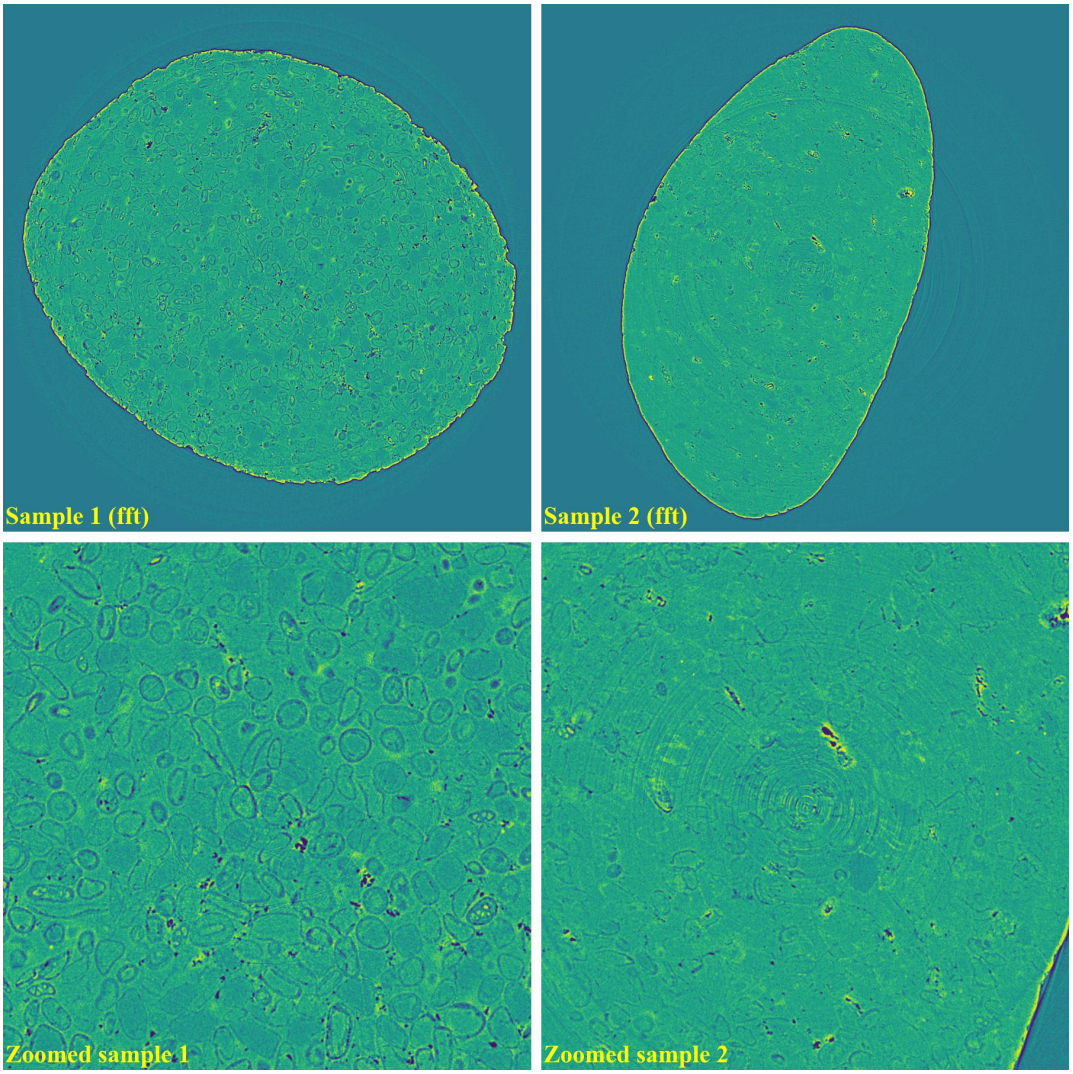
- If using the wavelet-fft-based method:

```
sinogram1 = rem.remove_stripe_based_wavelet_fft(sinogram1, level=5, size=2,␣
↪wavelet_name="db10")
sinogram2 = rem.remove_stripe_based_wavelet_fft(sinogram2, level=5, size=2,␣
↪wavelet_name="db10")
```



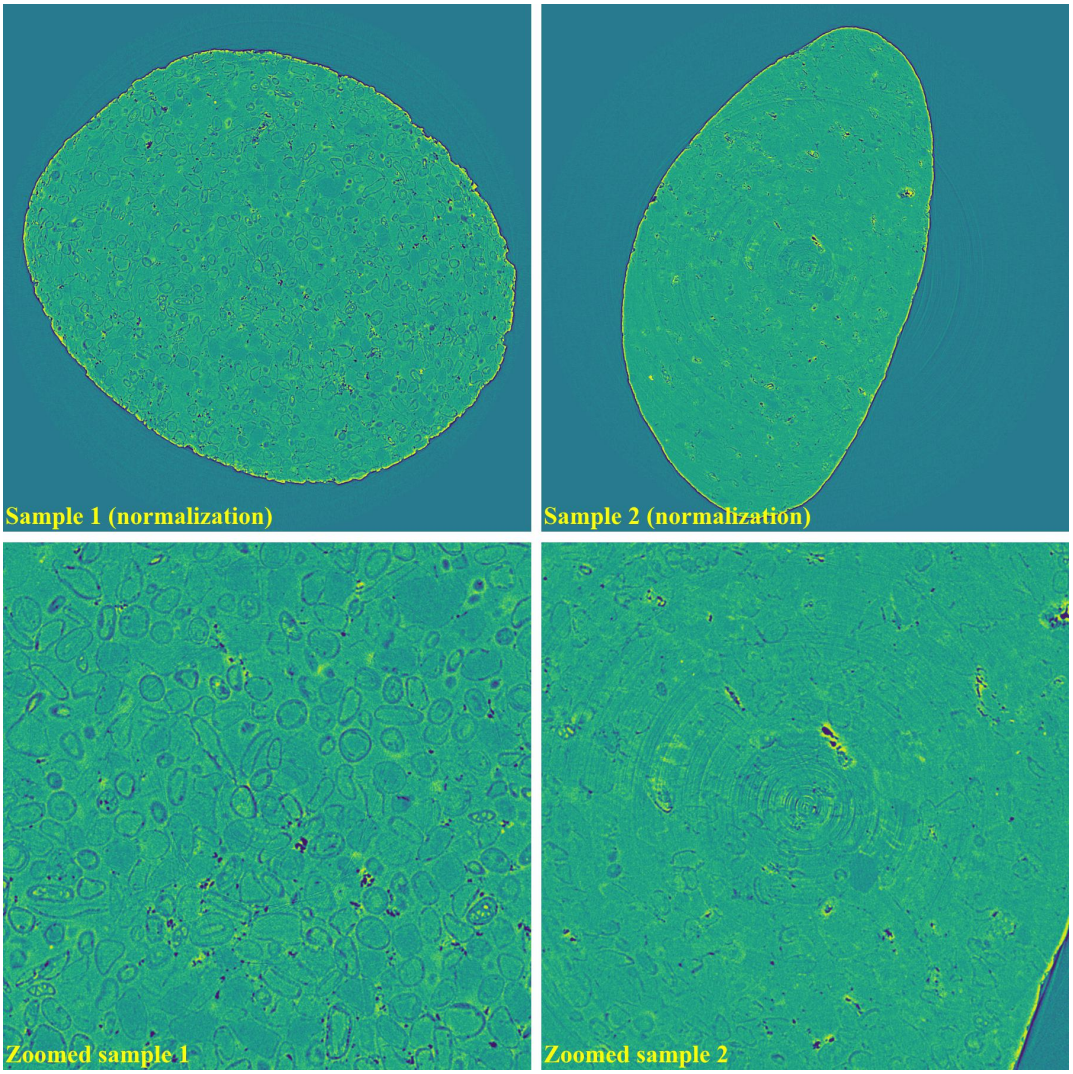
- If using the fft-based method:

```
sinogram1 = rem.remove_stripe_based_fft(sinogram1, u=20, n=10, v=0)
sinogram2 = rem.remove_stripe_based_fft(sinogram2, u=20, n=10, v=0)
```



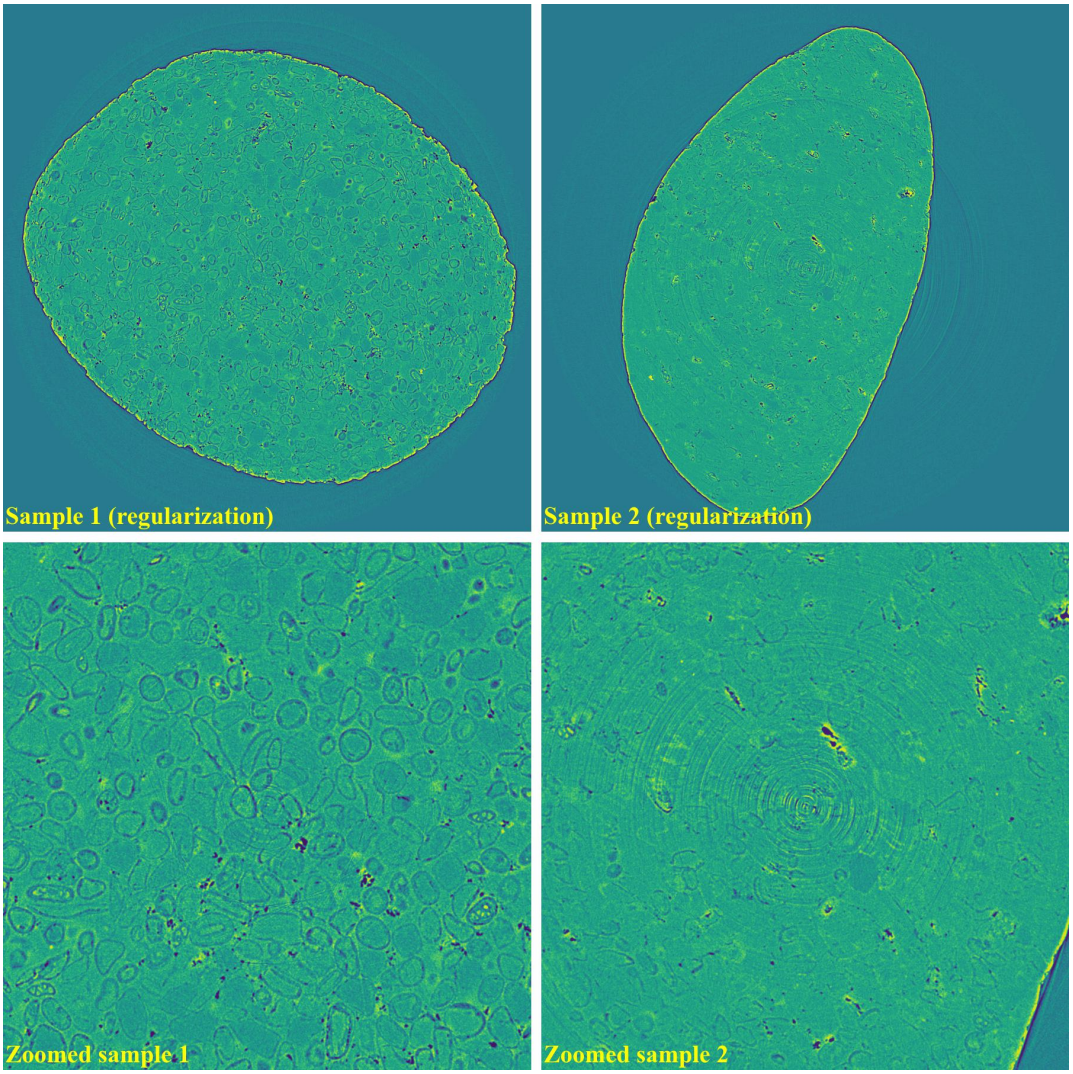
- If using the normalization-based method:

```
sinogram1 = rem.remove_stripe_based_normalization(sinogram1, 11)
sinogram2 = rem.remove_stripe_based_normalization(sinogram2, 11)
```



- If using the regularization-based method:

```
sinogram1 = rem.remove_stripe_based_regularization(sinogram1, alpha=0.0005,␣
↪apply_log=True)
sinogram2 = rem.remove_stripe_based_regularization(sinogram2, alpha=0.0005,␣
↪apply_log=True)
```

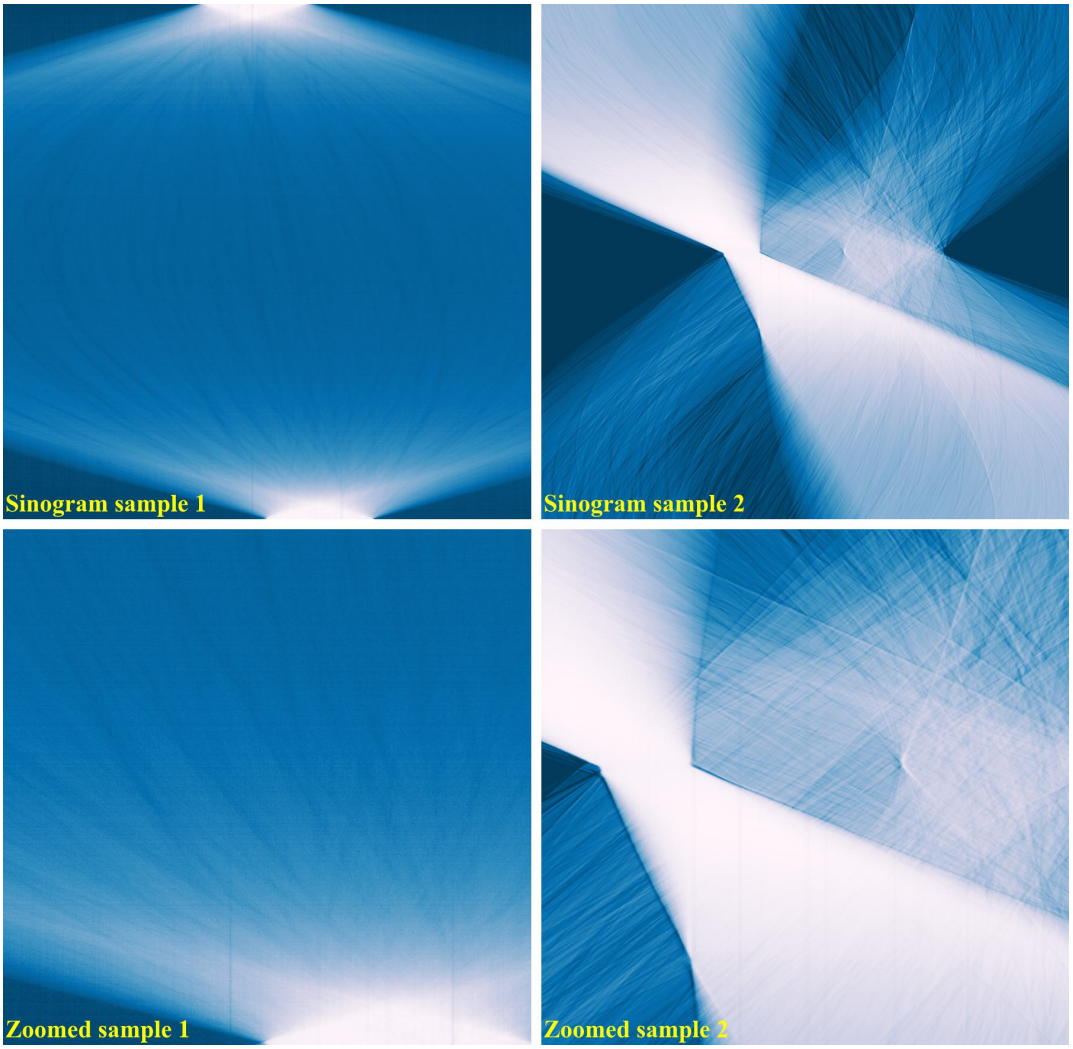


As demonstrated, using the algo-6543 method gives the best results with least side-effect artifacts. For other methods, it's impossible to use the same parameters for different samples or slices.

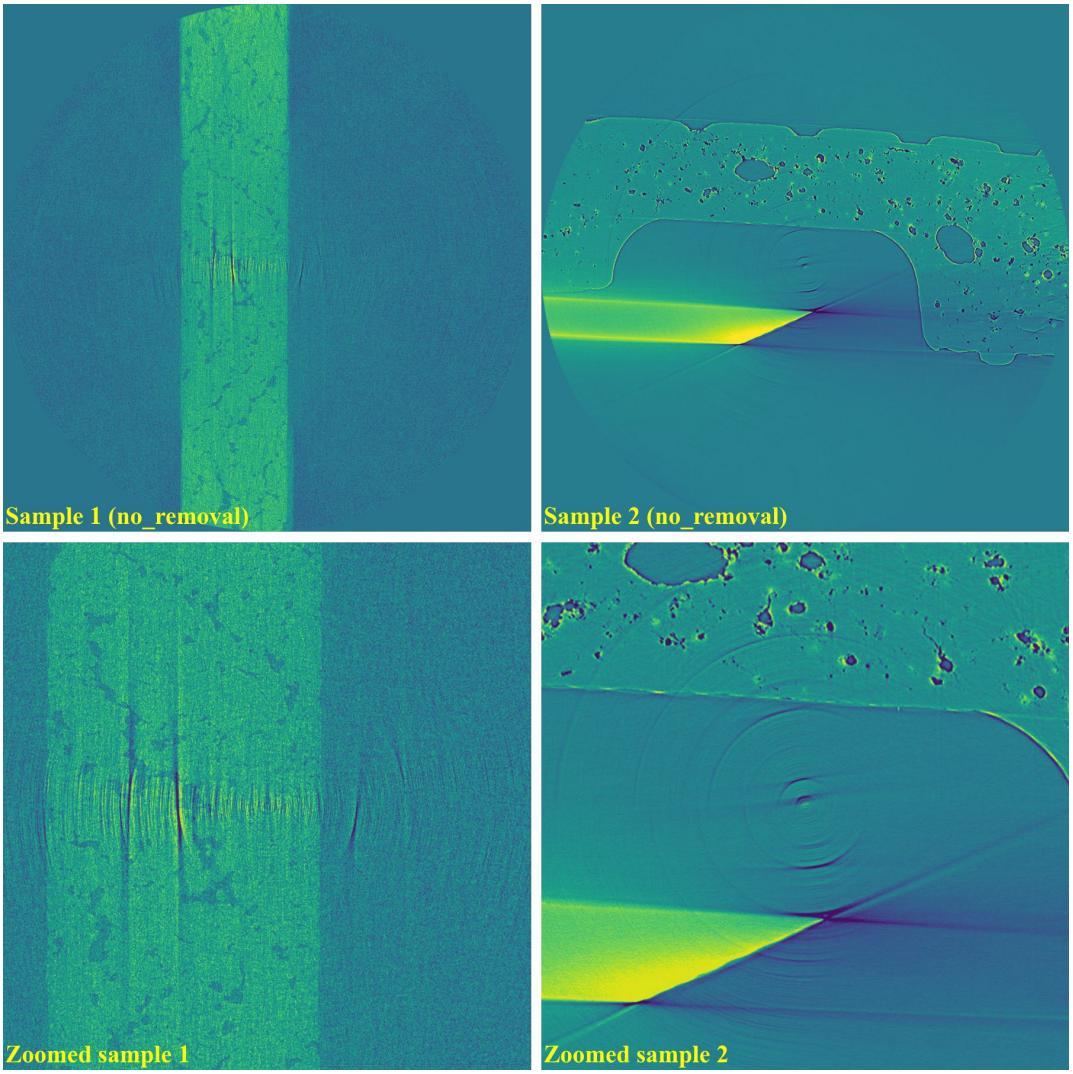
Partial ring artifacts

The following images show sinograms and reconstructed images of two samples in slab shapes which cause partial ring artifacts.

- Sinograms:



- Reconstructed images without using a ring removal method:



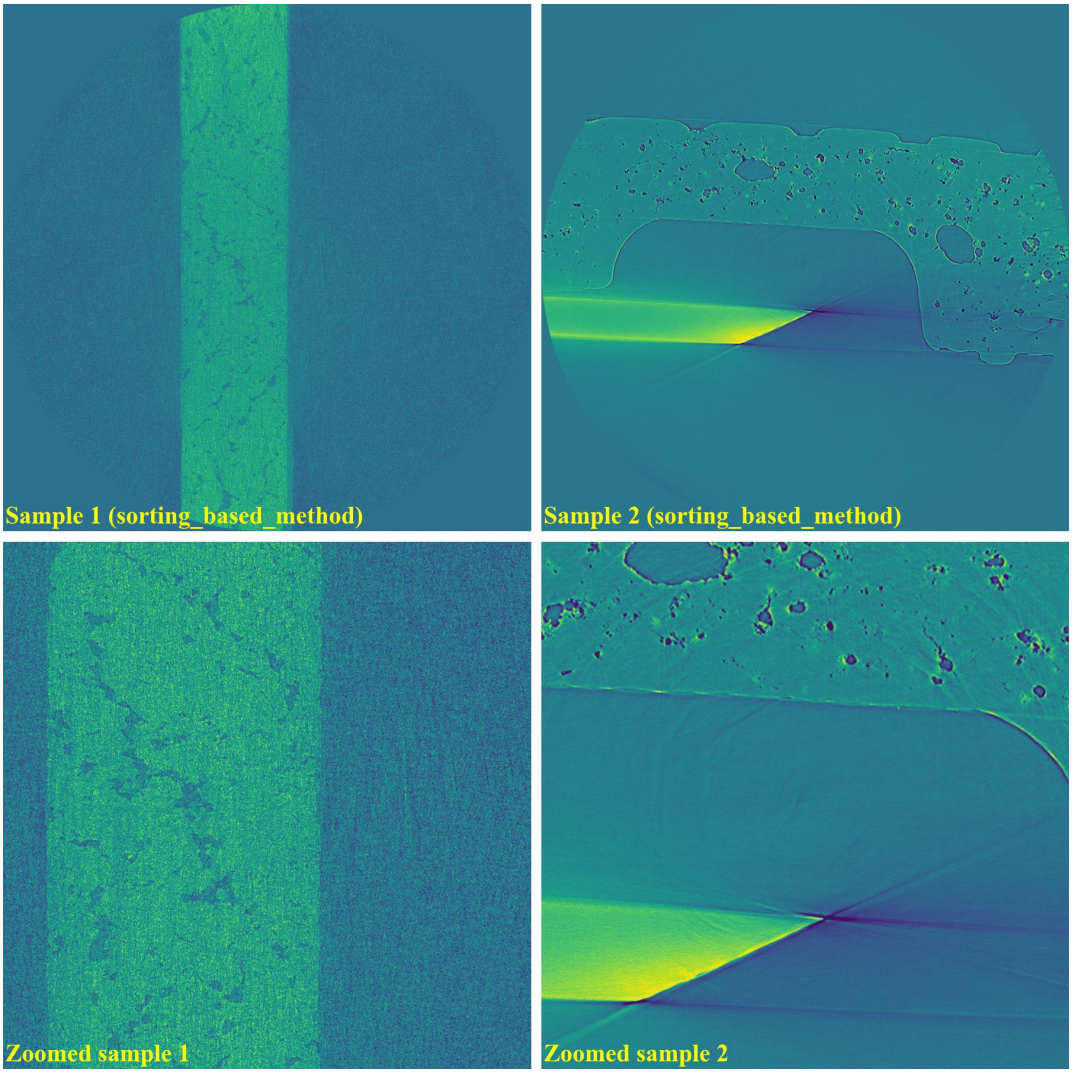
- If using the sorting-based method (algorithm 3 in [R19]):

```
import algotom.io.loadersaver as losa
import algotom.prep.calculation as calc
import algotom.prep.removal as rem
import algotom.rec.reconstruction as rec

input_base = "E:/data/"
output_base = "E:/rings_removed/sorting_based_method/"

sinogram1 = losa.load_image(input_base + "/sinogram_partial_stripe.tif")
sinogram2 = losa.load_image(input_base + "/large_partial_rings.tif")
center1 = calc.find_center_vo(sinogram1)
center2 = calc.find_center_vo(sinogram2)
print("center1 = ", center1)
print("center2 = ", center2)

sinogram1 = rem.remove_stripe_based_sorting(sinogram1, 51)
sinogram2 = rem.remove_stripe_based_sorting(sinogram2, 51)
```

(continues on next page)

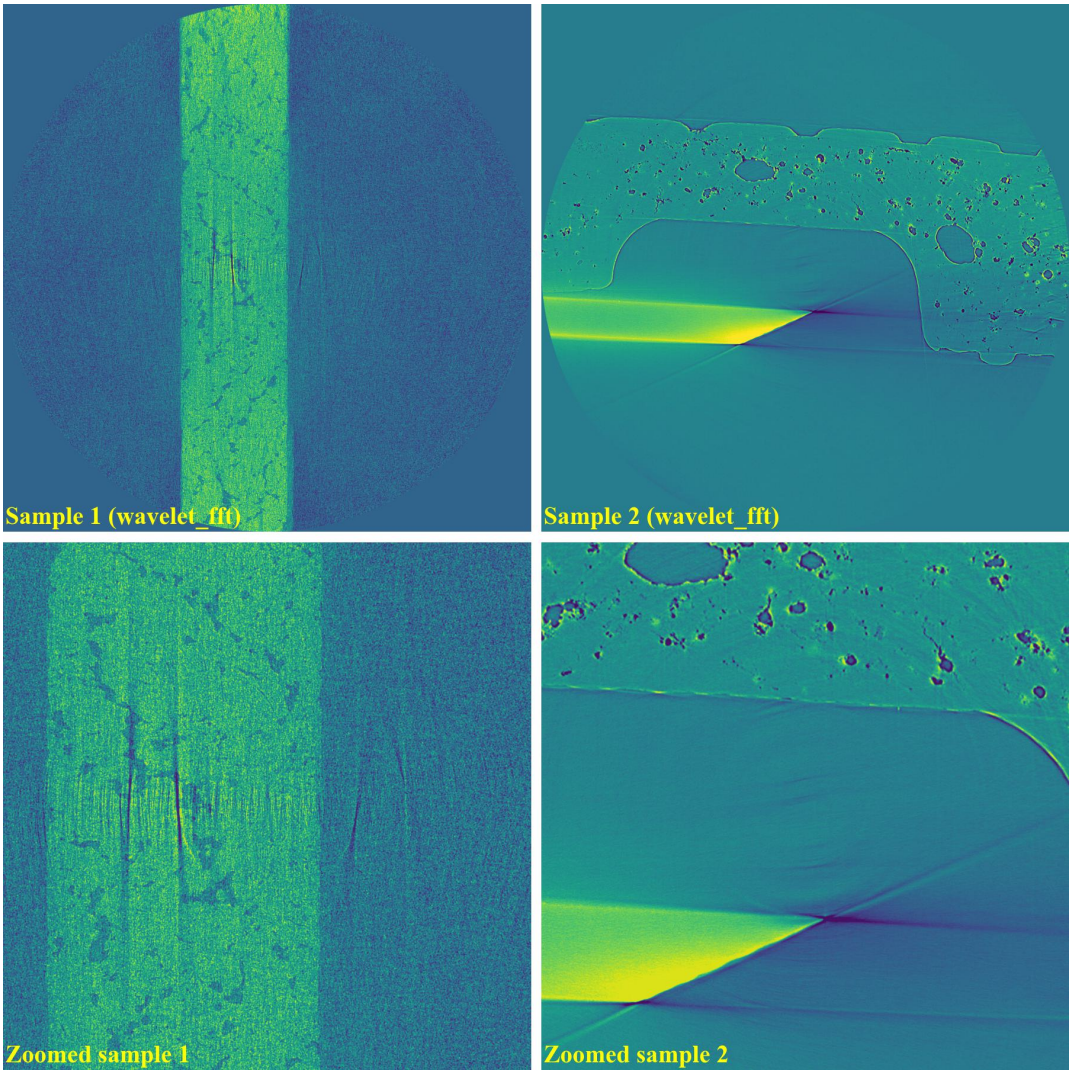
(continued from previous page)

```
img_rec1 = rec.dfi_reconstruction(sinogram1, center1)
img_rec2 = rec.dfi_reconstruction(sinogram2, center2)
losa.save_image(output_base + "/rec_sample1.tif", img_rec1)
losa.save_image(output_base + "/rec_sample2.tif", img_rec2)
```



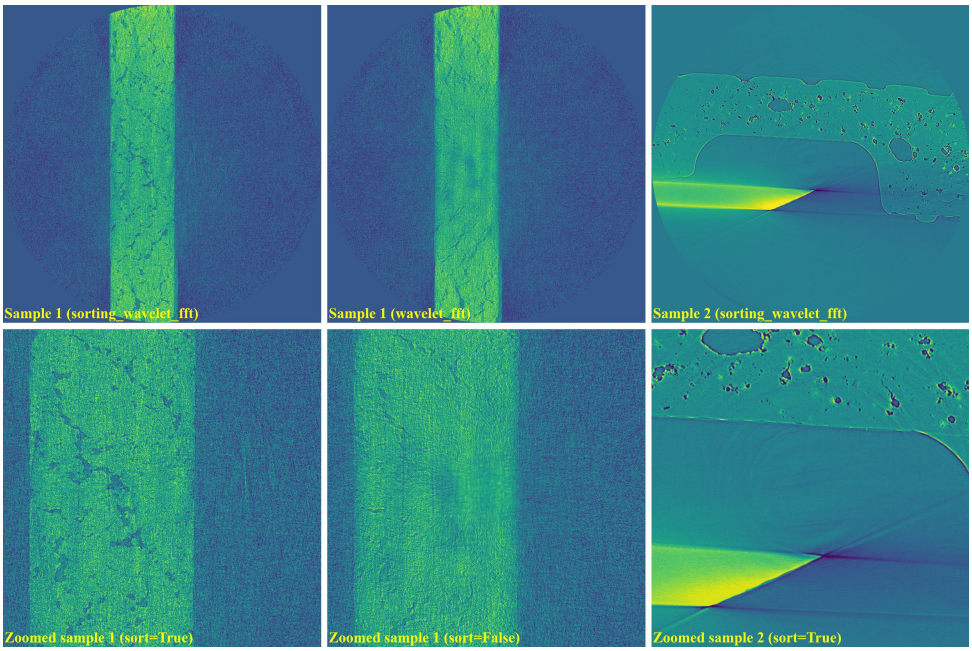
- If using the wavelet-fft-based method:

```
sinogram1 = rem.remove_stripe_based_wavelet_fft(sinogram1, level=5, size=2,␣
↪wavelet_name="db10")
sinogram2 = rem.remove_stripe_based_wavelet_fft(sinogram2, level=5, size=2,␣
↪wavelet_name="db10")
```



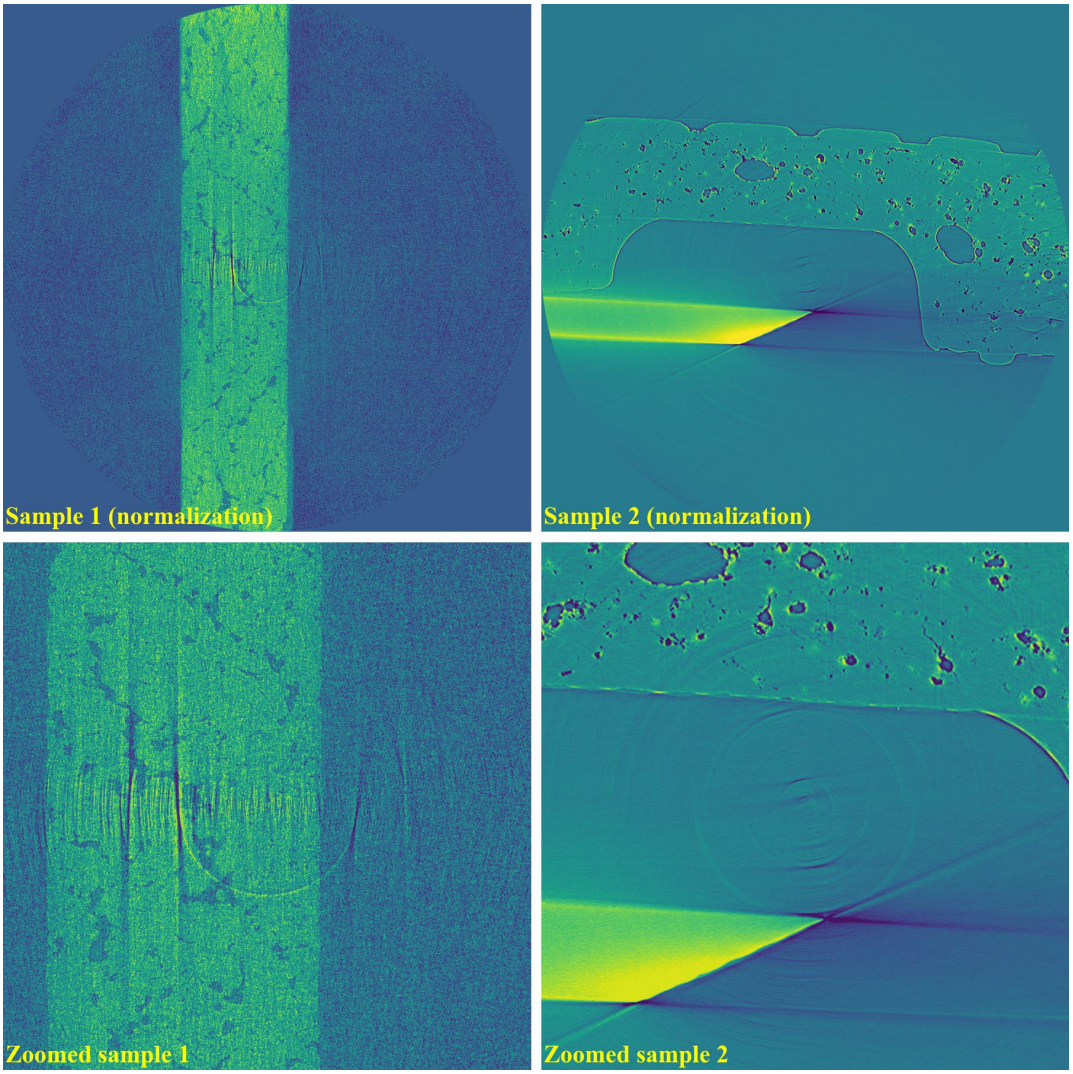
As can be seen, the original wavelet-fft-based method can't remove partial rings effectively. In Algotom, this method is improved by combining with the sorting method, which is the key part of algorithm 3 in [R19]. This helps to avoid void-center artifacts when strong parameters of the wavelet-fft-based method are used as demonstrated below

```
sinogram1a = rem.remove_stripe_based_wavelet_fft(sinogram1, level=6,␣
↪size=31, wavelet_name="db10", sort=True)
sinogram1b = rem.remove_stripe_based_wavelet_fft(sinogram1, level=6,␣
↪size=31, wavelet_name="db10", sort=False)
sinogram2 = rem.remove_stripe_based_wavelet_fft(sinogram2, level=5,␣
↪size=5, wavelet_name="db10", sort=True)
```



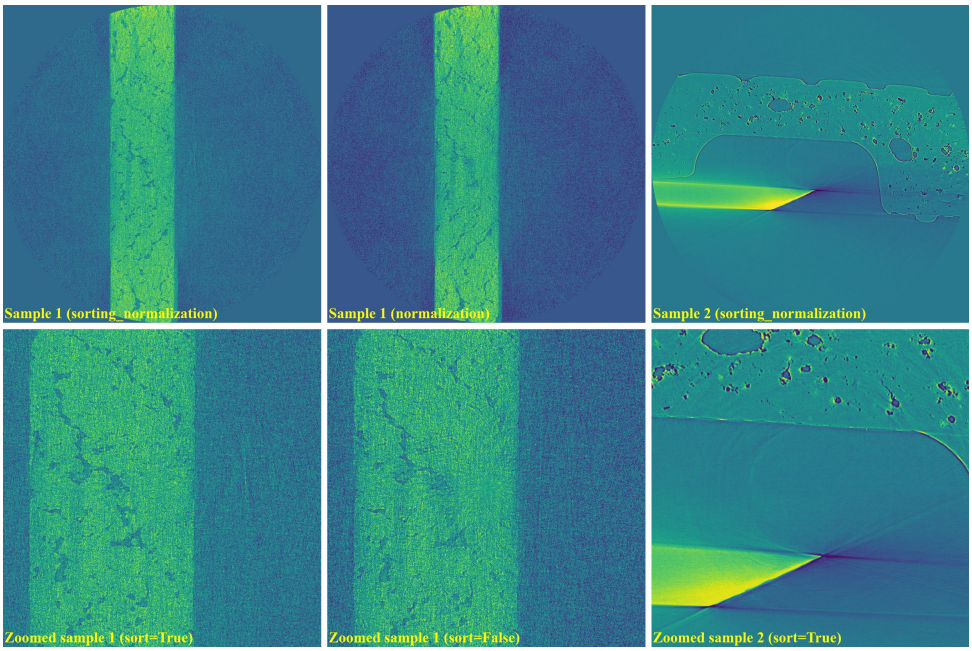
- If using the normalization-based method:

```
sinogram1 = rem.remove_stripe_based_normalization(sinogram1, sigma=17, num_
↪chunk=1)
sinogram2 = rem.remove_stripe_based_normalization(sinogram2, sigma=31, num_
↪chunk=1)
```



As shown above, the normalization-based method is not suitable for removing partial rings. However it can be improved by dividing a sinogram into many chunks of rows and combining with the sorting method.

```
sinogram1a = rem.remove_stripe_based_normalization(sinogram1,␣
↪sigma=17, num_chunk=30, sort=True)
sinogram1b = rem.remove_stripe_based_normalization(sinogram1,␣
↪sigma=17, num_chunk=30, sort=False)
sinogram2 = rem.remove_stripe_based_normalization(sinogram2,␣
↪sigma=31, num_chunk=30, sort=True)
```

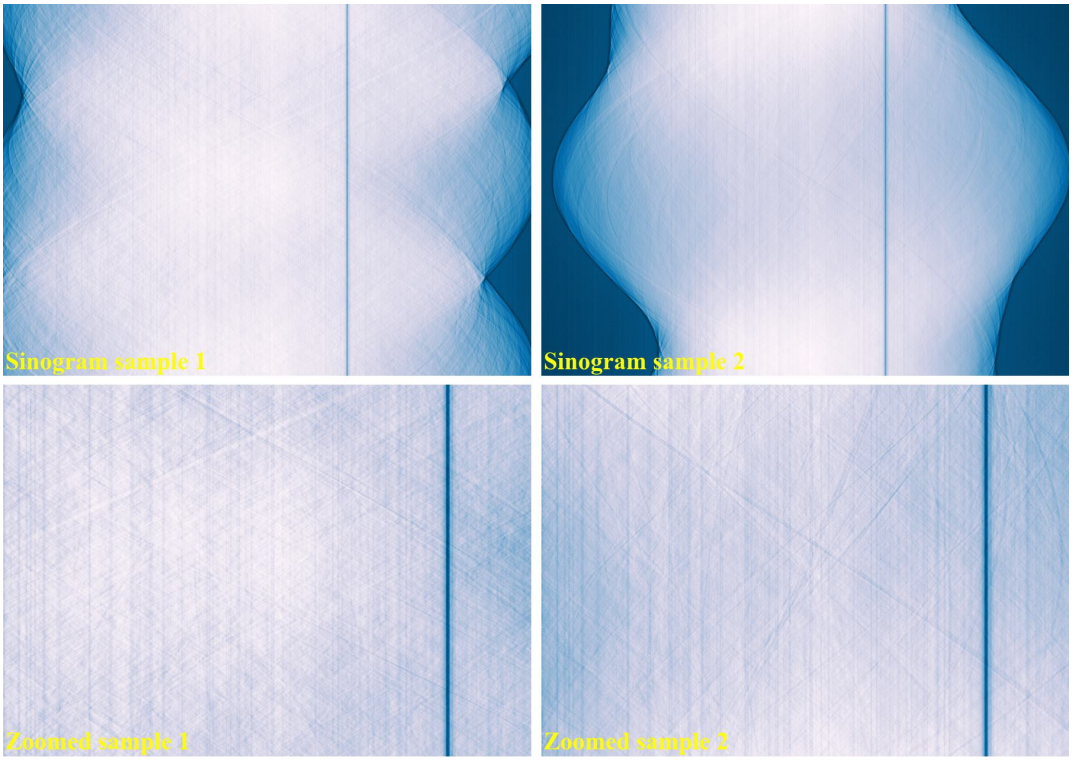


The above sub-section is to demonstrate the effectiveness of the sorting-based method in removing partial ring artifacts and improving other methods in avoiding void-center artifacts. Results of using the fft-based method and regularization-based method are not demonstrated here because their performance is similar to the wavelet-fft-based method and the normalization-based method.

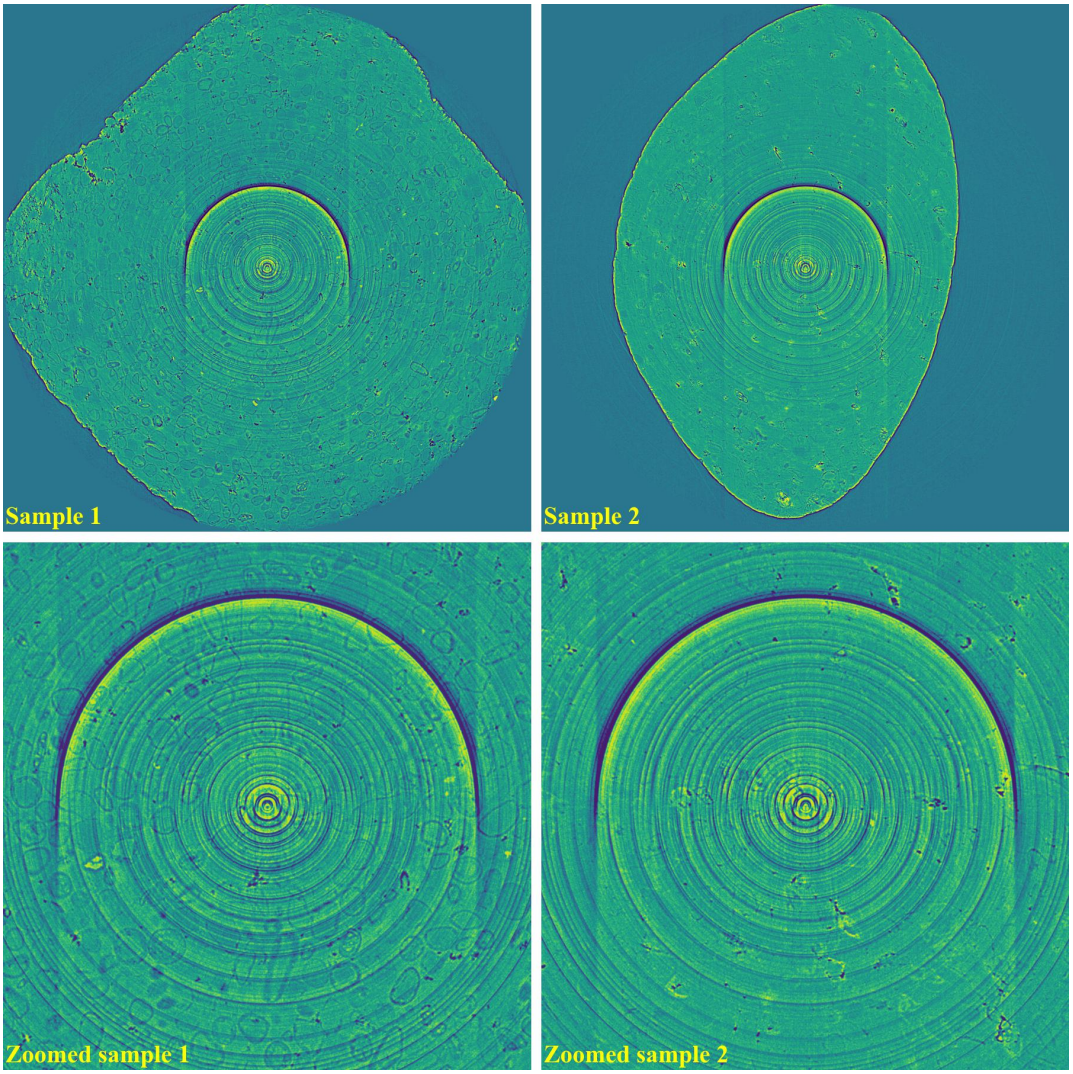
All types of ring artifacts

The following images show sinograms and reconstructed images of two limestone rocks with different shapes having all types of stripe/ring artifacts in one slice.

- Sinograms:



- Reconstructed images without using a ring removal method:



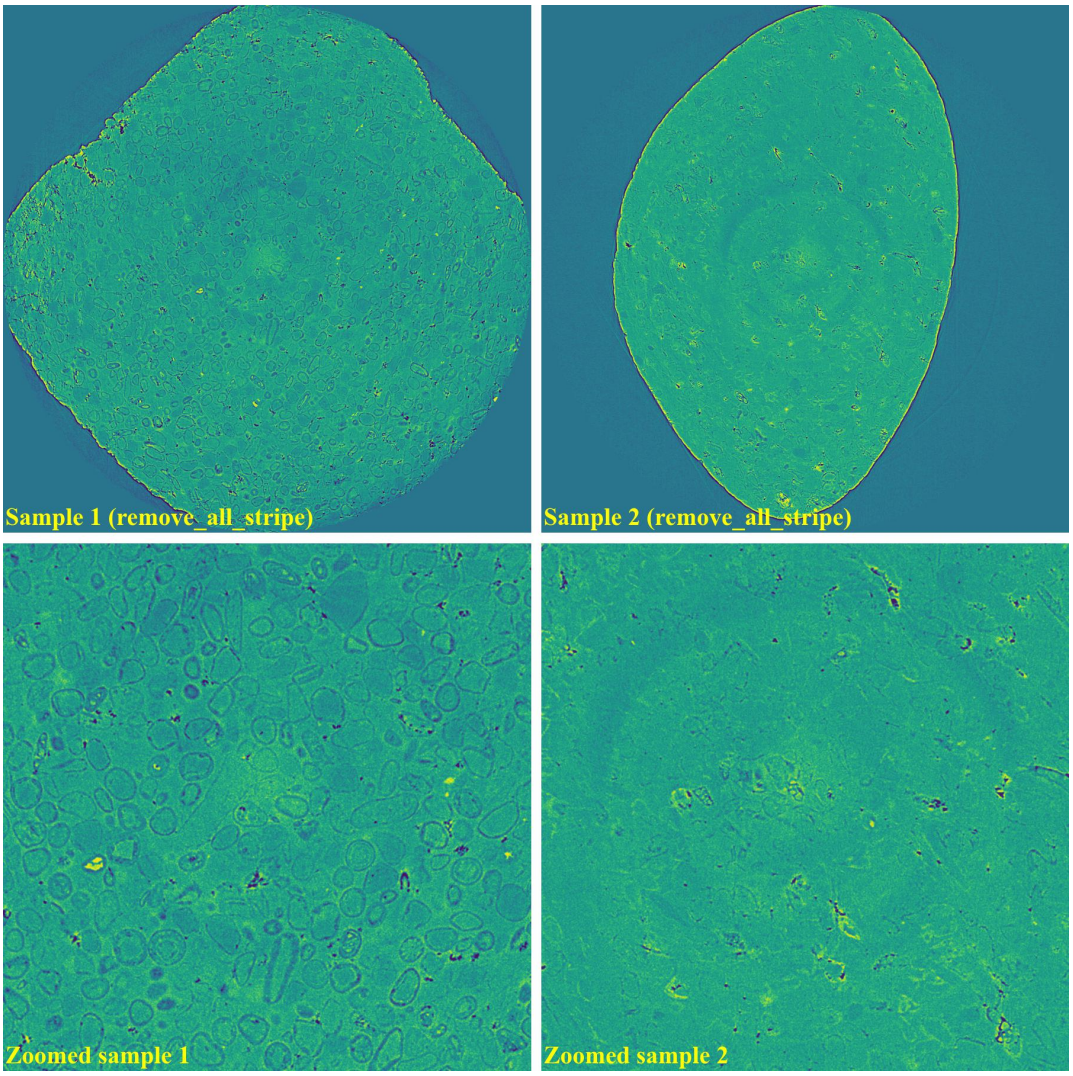
- If using the combination of methods:

```
import algotom.io.loadersaver as losa
import algotom.prep.calculation as calc
import algotom.prep.removal as rem
import algotom.rec.reconstruction as rec

input_base = "E:/data/"
output_base = "E:/rings_removed/remove_all_stripe/"

sinogram1 = losa.load_image(input_base + "/all_stripe_types_sample1.tif")
sinogram2 = losa.load_image(input_base + "/all_stripe_types_sample2.tif")

center1 = calc.find_center_vo(sinogram1)
center2 = calc.find_center_vo(sinogram2)

print("center1 = ", center1)
print("center2 = ", center2)
```

(continues on next page)

(continued from previous page)

```
sinogram1 = rem.remove_all_stripe(sinogram1, snr=2.0, la_size=81, sm_
↪size=31)
sinogram2 = rem.remove_all_stripe(sinogram2, snr=3.0, la_size=81, sm_
↪size=31)

img_rec1 = rec.dfi_reconstruction(sinogram1, center1)
img_rec2 = rec.dfi_reconstruction(sinogram2, center2)
losa.save_image(output_base + "/rec_sample1.tif", img_rec1)
losa.save_image(output_base + "/rec_sample2.tif", img_rec2)
```



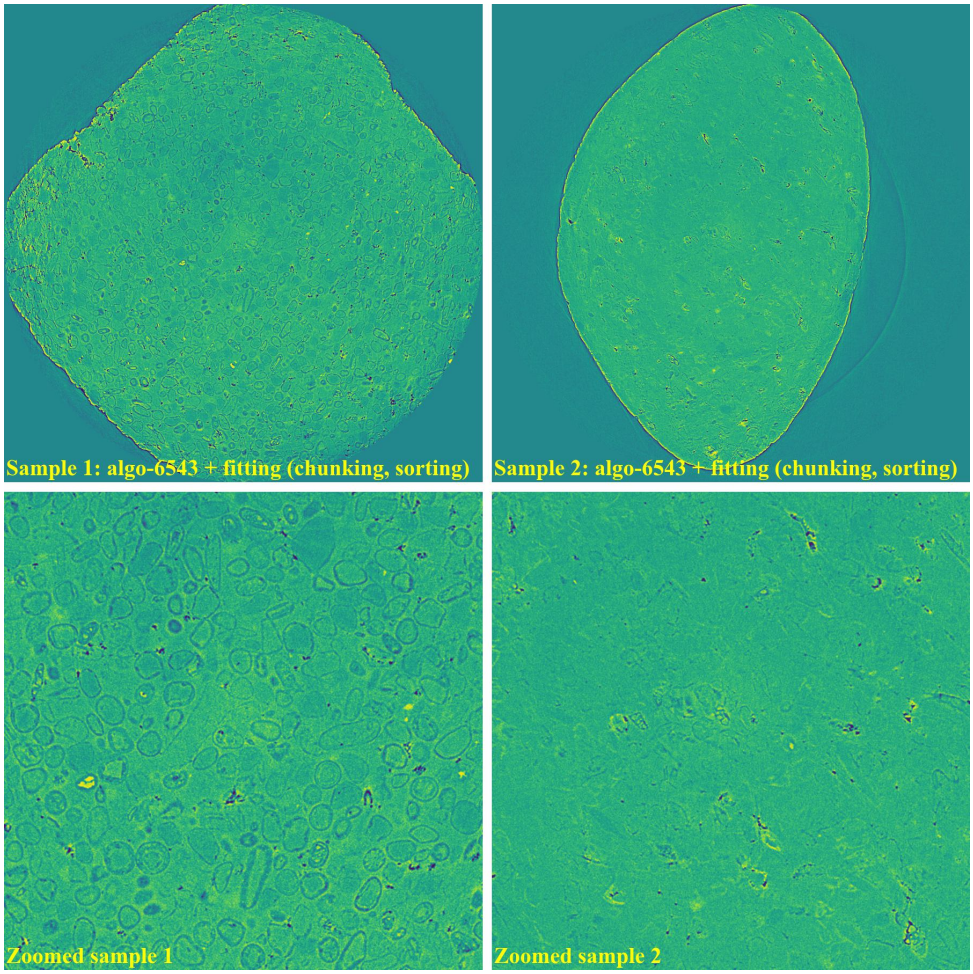
As can be seen, there are still low-contrast ring artifacts which are difficult to detect and remove. These low-contrast rings are caused by the halo effect around blob areas on a scintillator. There is a strong removal method proposed in [R19] and its improvement can help to deal with such ring artifacts as below.

```
sinogram1 = rem.remove_all_stripe(sinogram1, snr=2.0, la_size=81, sm_
↪size=31)
sinogram2 = rem.remove_all_stripe(sinogram2, snr=3.0, la_size=81, sm_
```

(continues on next page)

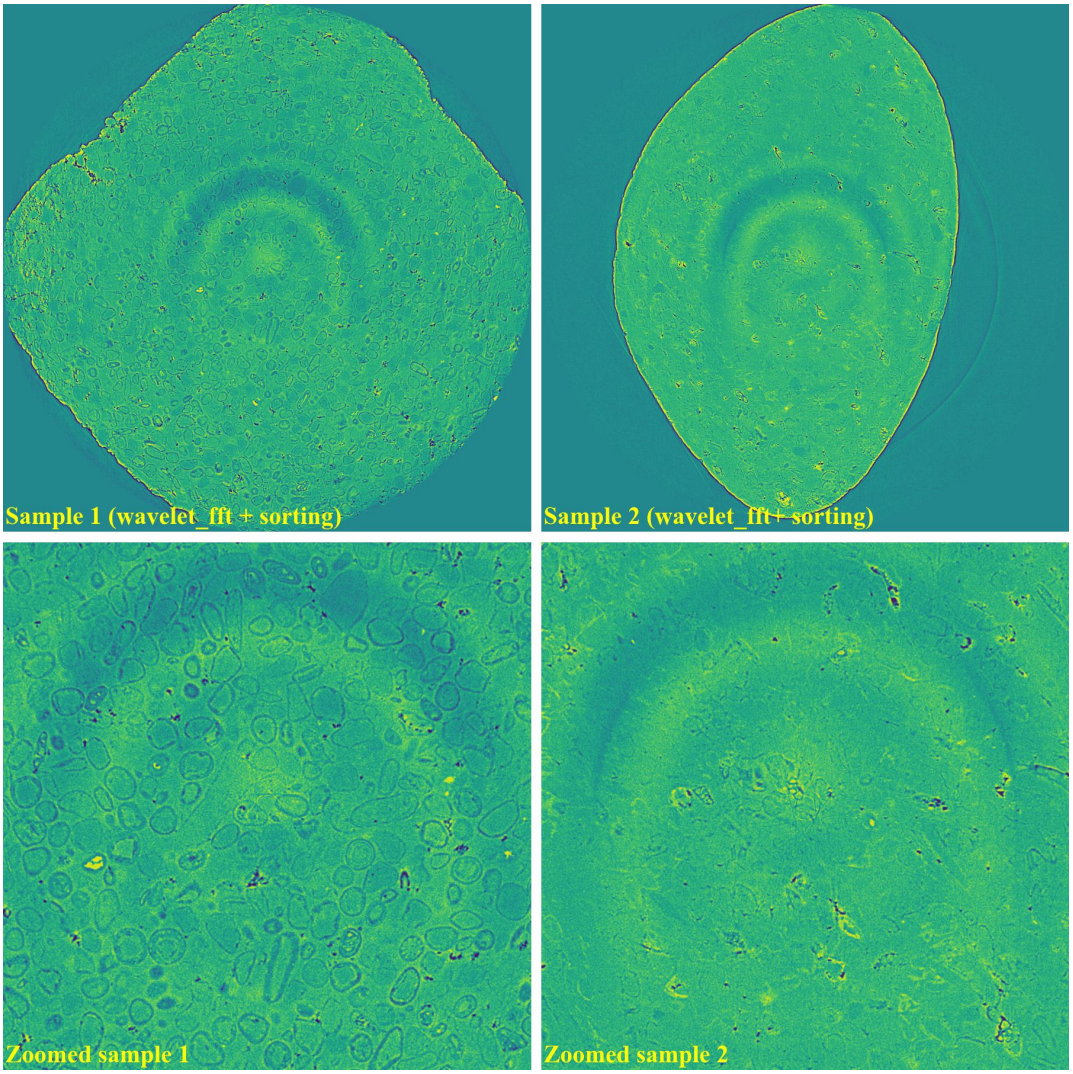
(continued from previous page)

```
↪size=31)
sinogram1 = rem.remove_stripe_based_fitting(sinogram1, order=1,␣
↪sigma=10, num_chunk=9, sort=True)
sinogram2 = rem.remove_stripe_based_fitting(sinogram2, order=1,␣
↪sigma=10, num_chunk=9, sort=True)
```



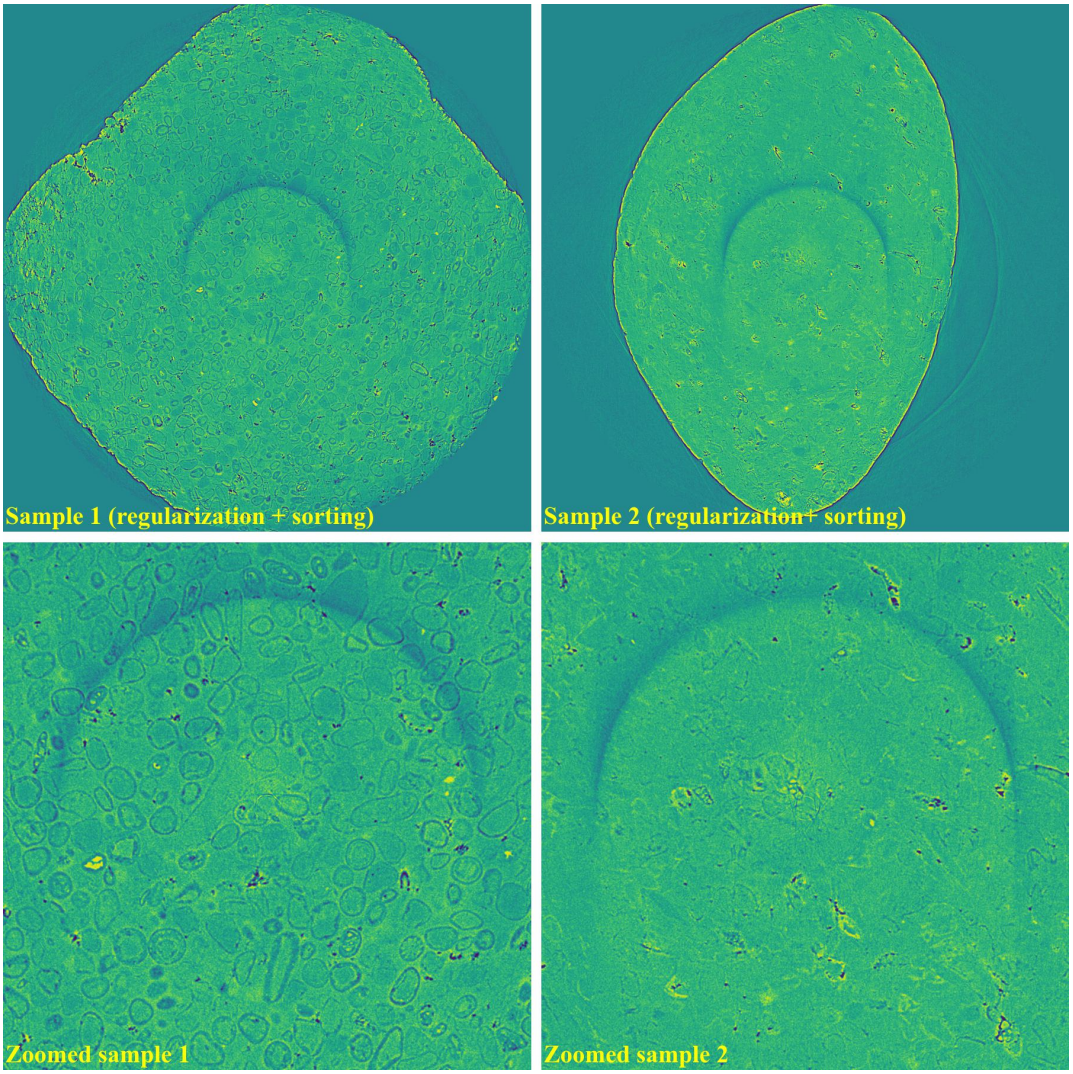
- If using the wavelet-fft-based method with the sorting-based method:

```
sinogram1 = rem.remove_stripe_based_wavelet_fft(sinogram1, level=6, size=5,␣
↪wavelet_name="db10", sort=True)
sinogram2 = rem.remove_stripe_based_wavelet_fft(sinogram2, level=6, size=5,␣
↪wavelet_name="db10", sort=True)
```



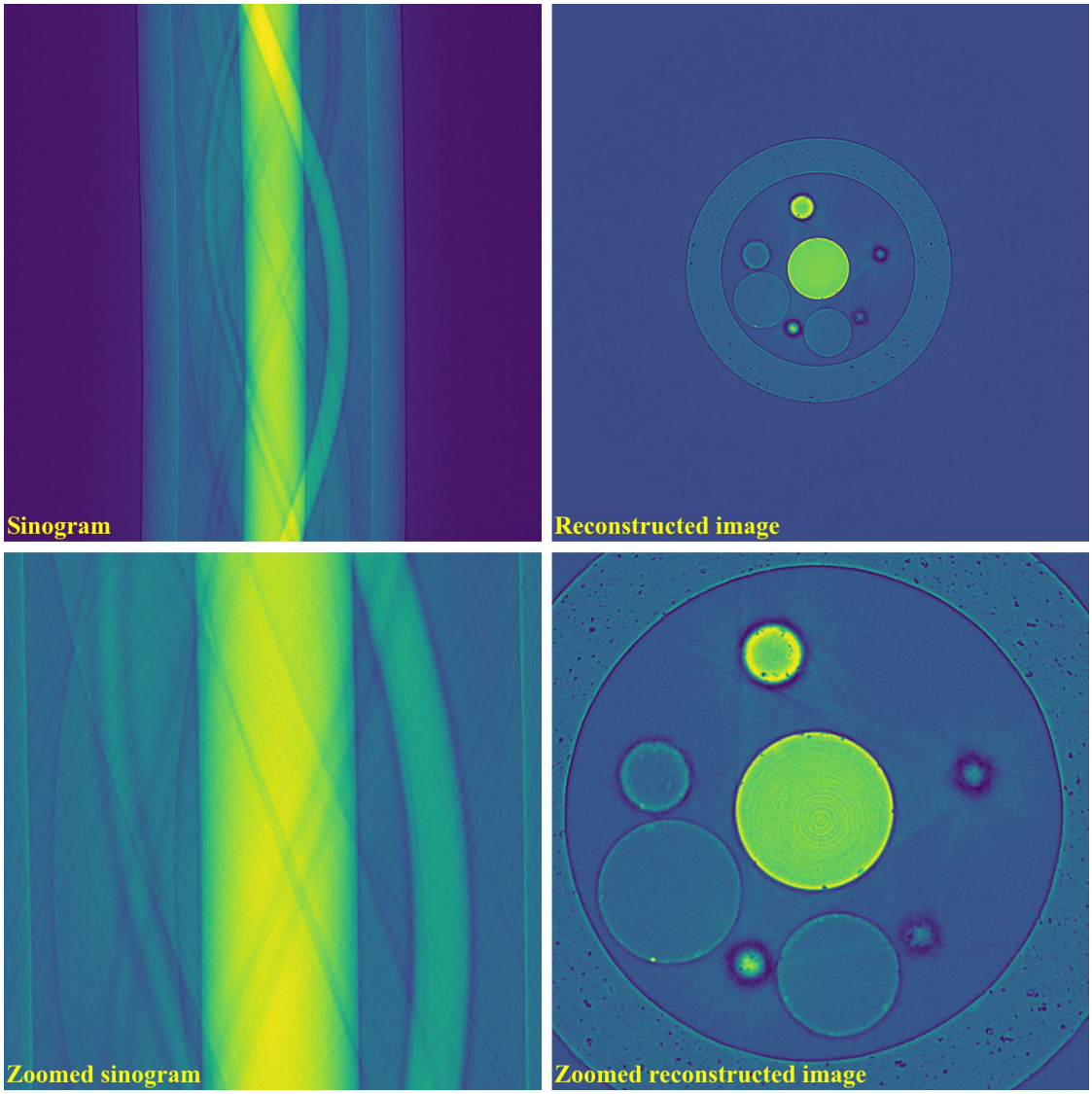
- If using the regularization-based method with the sorting-based method:

```
sinogram1 = rem.remove_stripe_based_regularization(sinogram1, alpha=0.001,␣
↪num_chunk=15, sort=True)
sinogram2 = rem.remove_stripe_based_regularization(sinogram2, alpha=0.001,␣
↪num_chunk=15, sort=True)
```



Having valid stripes (not artifacts)

For samples containing round-shape objects (tubes, spheres), they can produce sinograms having valid stripes. This is a problem for fft-based methods or normalization-based methods, but not for sorting-based methods.



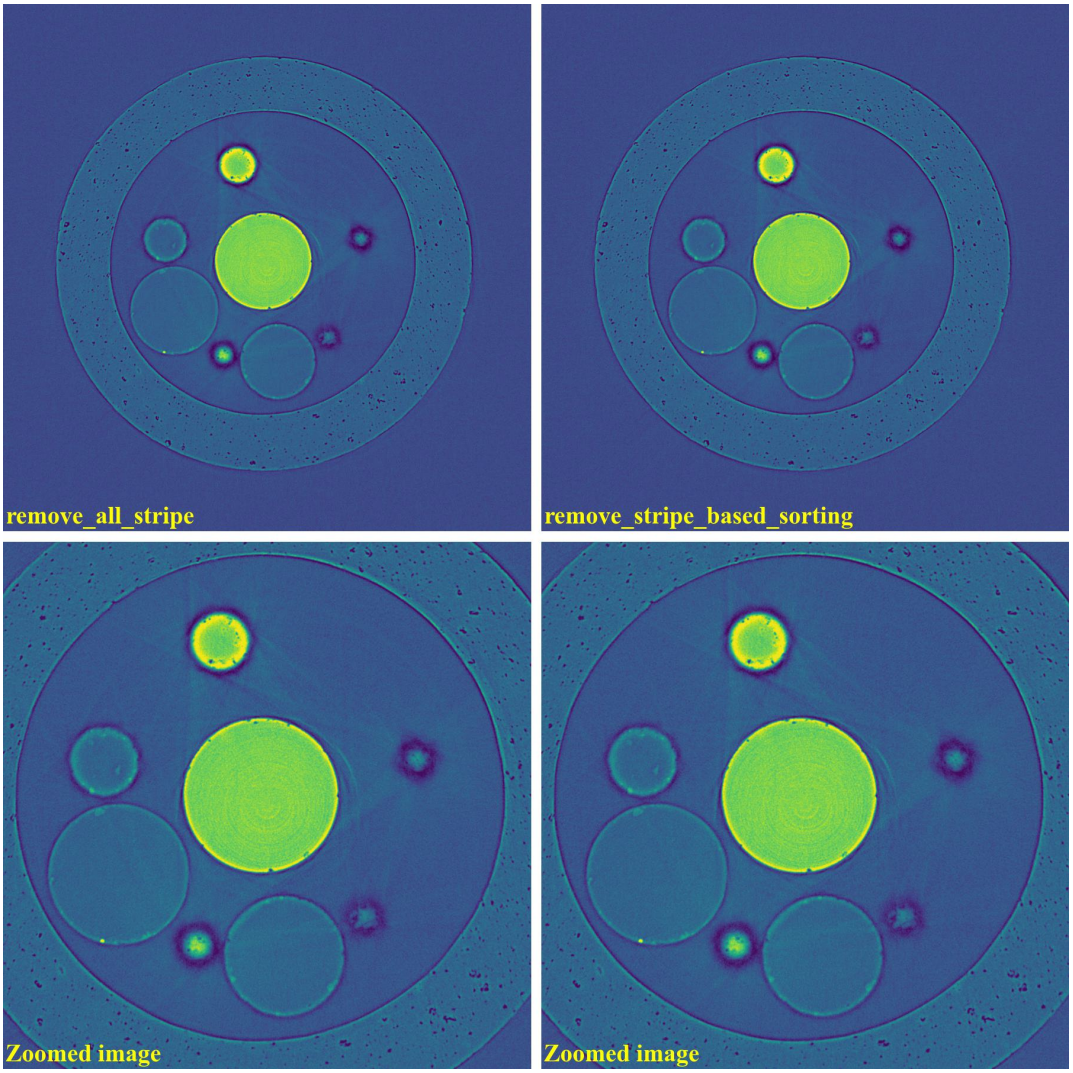
- Results of using the combined method and the sorting-based method as below. Note that the remaining ring artifacts are insignificant. Although visible, they have nearly the same SNR (signal-to-noise ratio) as nearby background.

```
import algotom.io.loadersaver as losa
import algotom.prep.calculation as calc
import algotom.prep.removal as rem
import algotom.rec.reconstruction as rec

input_base = "E:/data/"
output_base = "E:/valid_stripes/rings_removed/"

sinogram = losa.load_image(input_base + "/valid_stripes.tif")
center = calc.find_center_vo(sinogram)
print("center =", center)

sinogram1 = rem.remove_all_stripe(sinogram, snr=3.0, la_size=31, sm_size=21)
sinogram2 = rem.remove_stripe_based_sorting(sinogram, 21)
```

(continues on next page)

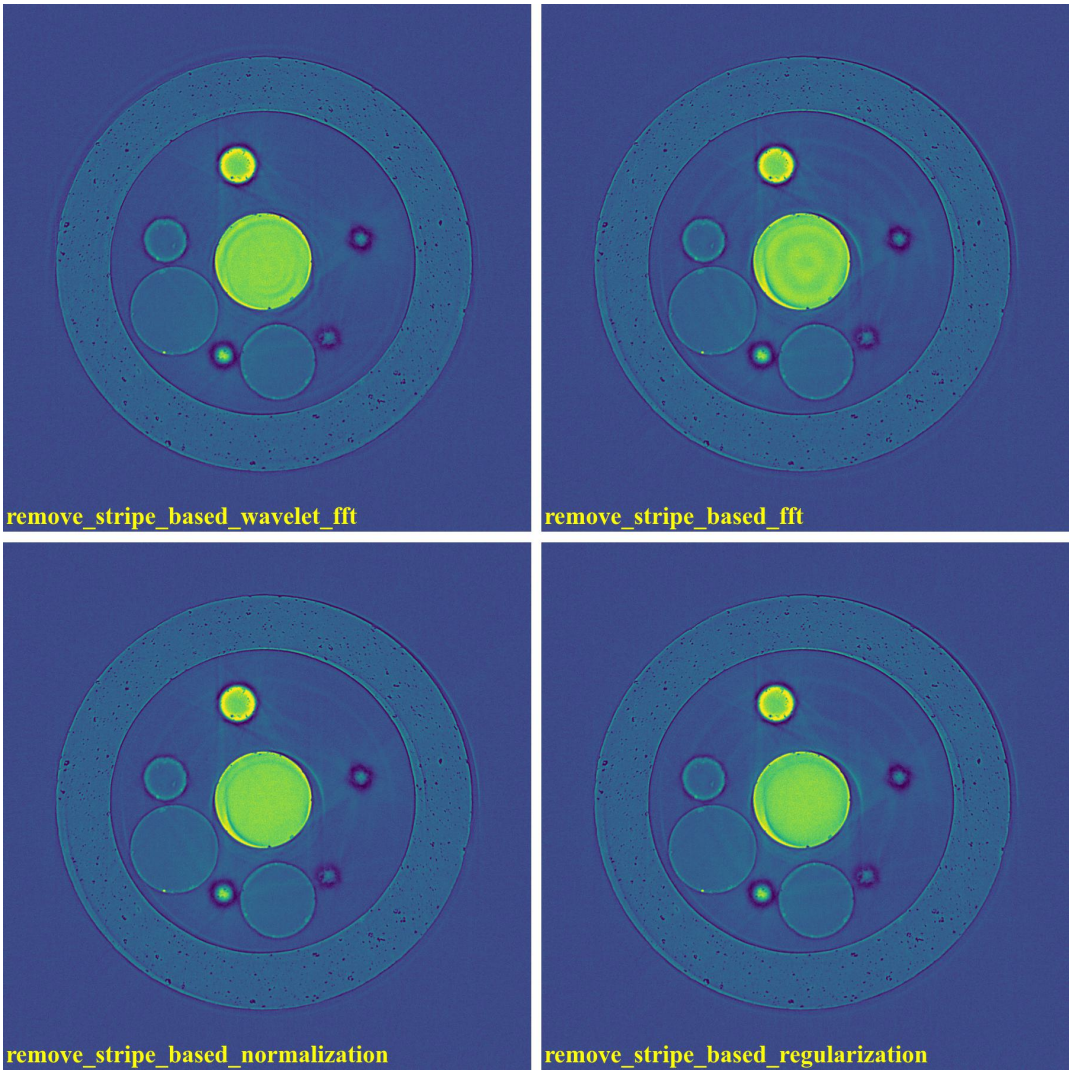
(continued from previous page)

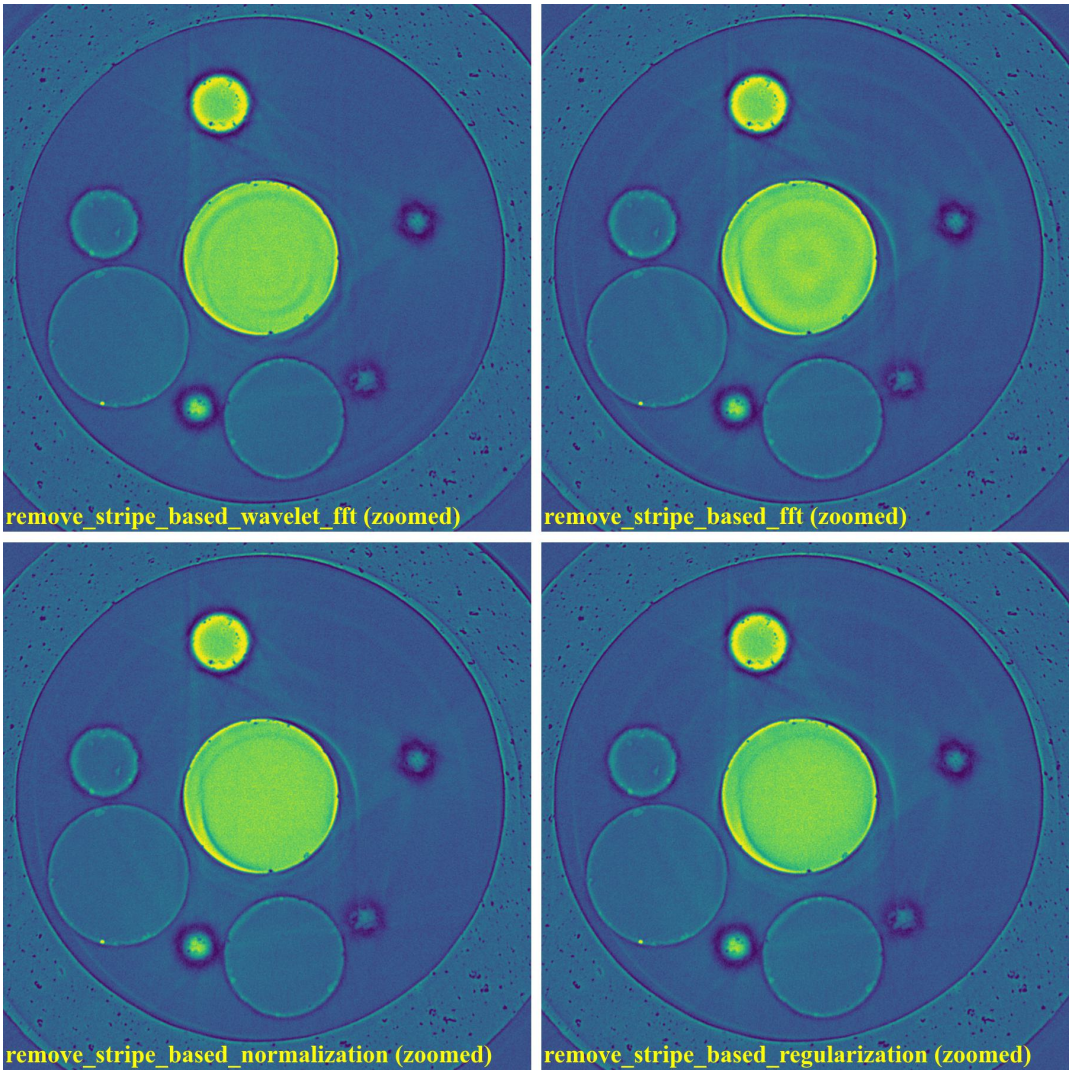
```
img_rec1 = rec.dfi_reconstruction(sinogram1, center)
img_rec2 = rec.dfi_reconstruction(sinogram2, center)
losa.save_image(output_base + "/rec_img1.tif", img_rec1)
losa.save_image(output_base + "/rec_img2.tif", img_rec2)
```



- Results of using other methods are shown below. Although reduced strength, they still produce lots of side-effect artifacts for such a pretty clean sinogram.

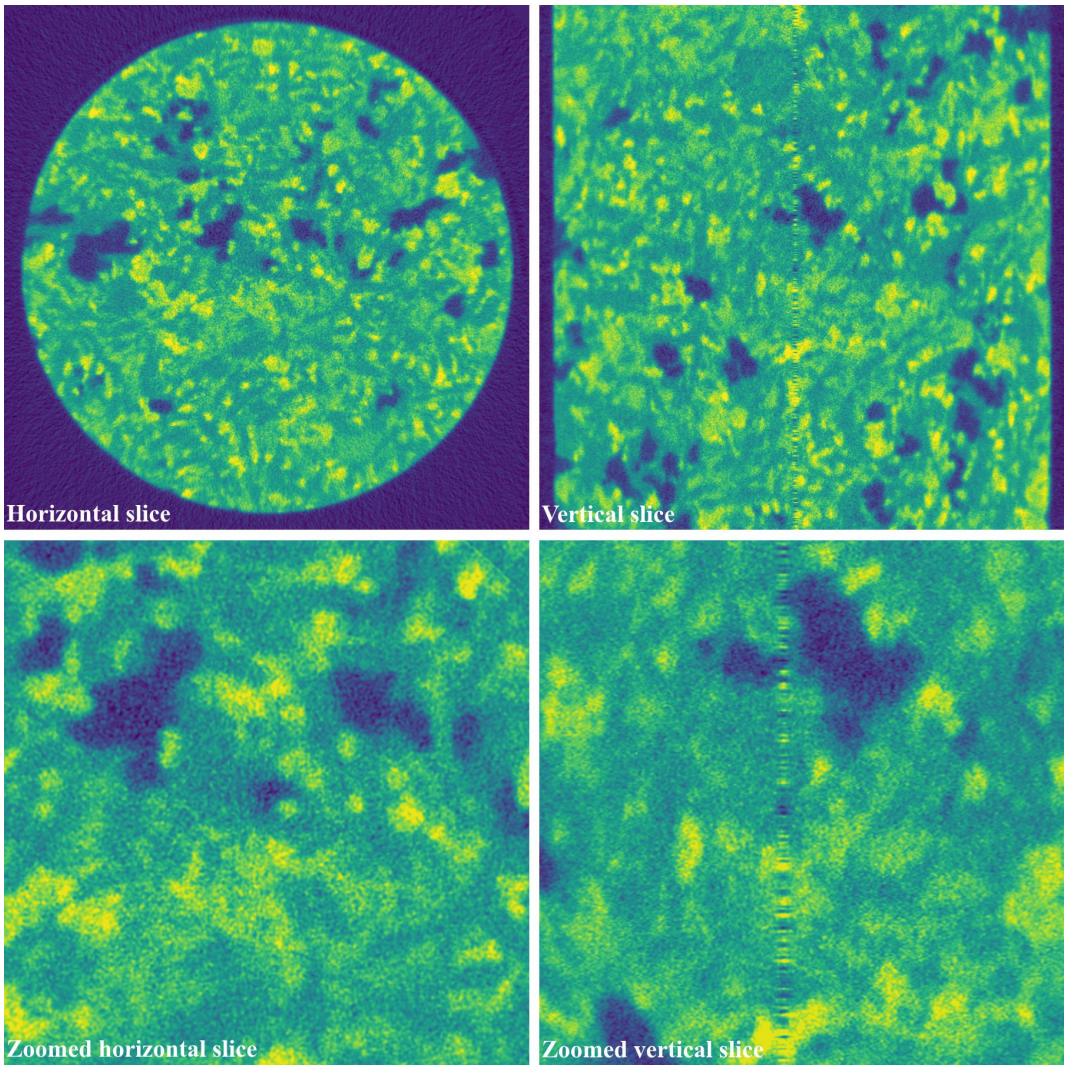
```
sinogram1 = rem.remove_stripe_based_wavelet_fft(sinogram, level=4, size=1)
sinogram2 = rem.remove_stripe_based_fft(sinogram, u=40, n=8, v=0)
sinogram3 = rem.remove_stripe_based_normalization(sinogram, sigma=11)
sinogram4 = rem.remove_stripe_based_regularization(sinogram, alpha=0.005)
```


remove_stripe_based_wavelet_fft
remove_stripe_based_fft
remove_stripe_based_normalization
remove_stripe_based_regularization



For cone-beam tomography

Post-processing ring-removal methods are often used for cone-beam tomography because reconstruction can't be done sinogram-by-sinogram. However, they can cause void-center artifacts, which may not be visible in horizontal slices but clearly visible along vertical slices. More than that, these methods can't remove side effects of unresponsive-stripe artifacts and fluctuating-stripe artifacts which not only give rise to ring artifacts but also streak artifacts in a reconstructed image.



Certainly, we can apply pre-processing ring-removal methods along the sinogram direction. The only downside is that we have to store intermediate results for switching between the *projection space and the sinogram space*. It is common that commercial tomography systems output flat-field-corrected projection-images as 16-bit tif format (grayscale in the range of 0-65535). The following shows how to apply pre-processing methods along the sinogram direction step-by-step:

- First of all, we convert tiffs to hdf file-format for fast slicing 3D data.

```
import timeit
import numpy as np
import algotom.io.converter as conv
import algotom.io.loadersaver as losa

input_base = "E:/cone_beam/rawdata/tif_projections/"
output_file = "E:/tmp/projections.hdf"

t0 = timeit.default_timer()
list_files = losa.find_file(input_base + "/*.tif*")
depth = len(list_files)
(height, width) = np.shape(losa.load_image(list_files[0]))
```

(continues on next page)

(continued from previous page)

```
conv.convert_tif_to_hdf(input_base, output_file, key_path='entry/data', crop=(0, 0,␣
↪0, 0))
t1 = timeit.default_timer()
print("Done!!!. Total time cost: {}".format(t1 - t0))
```

- Then load the converted data and apply pre-processing methods. Note about the change of data shape in each step.

```
import timeit
import multiprocessing as mp
from joblib import Parallel, delayed
import numpy as np
import algotom.io.loadersaver as losa
import algotom.prep.removal as rem
import algotom.prep.correction as corr

input_file = "E:/tmp/projections.hdf"
output_file = "E:/tmp/tmp/projections_preprocessed.hdf"

data = losa.load_hdf(input_file, key_path='entry/data')
(depth, height, width) = data.shape

# Note that the shape of output data is (height, depth, width)
# for faster writing to hdf file.
output = losa.open_hdf_stream(output_file, (height, depth, width), data_type=
↪"float32")

t0 = timeit.default_timer()
# For parallel processing
ncore = mp.cpu_count()
chunk_size = np.clip(ncore - 1, 1, height - 1)
last_chunk = height - chunk_size * (height // chunk_size)
for i in np.arange(0, height - last_chunk, chunk_size):
    sinograms = np.float32(data[:, i:i + chunk_size, :])
    # Note about the change of the shape of output_tmp (which is a list of␣
↪processed sinogram)
    output_tmp = Parallel(n_jobs=ncore, prefer="threads")(delayed(rem.remove_all_
↪stripe)(sinograms[:, j, :], 3.0, 51, 21) for j in range(chunk_size))

    # Apply beam hardening correction if need to
    # output_tmp = np.asarray(output_tmp)
    # output_tmp = Parallel(n_jobs=ncore, prefer="threads")(
    #     delayed(corr.beam_hardening_correction)(output_tmp[j], 40, 2.0, False)␣
↪for j in range(chunk_size))

    output[i:i + chunk_size] = np.asarray(output_tmp, dtype=np.float32)
    t1 = timeit.default_timer()
    print("Done sinograms: {0}-{1}. Time {2}".format(i, i + chunk_size, t1 - t0))

if last_chunk != 0:
    sinograms = np.float32(data[:, height - last_chunk:height, :])
    output_tmp = Parallel(n_jobs=ncore, prefer="threads")(delayed(rem.remove_all_
```

(continues on next page)

(continued from previous page)

```
↪stripe)(sinograms[:, j, :], 3.0, 51, 21) for j in range(last_chunk))

    # Apply beam hardening correction if need to
    # output_tmp = np.asarray(output_tmp)
    # output_tmp = Parallel(n_jobs=ncore, prefer="threads")(
    #     delayed(corr.beam_hardening_correction)(output_tmp[j], 40, 2.0, False)␣
↪for j in range(last_chunk))

    output[height - last_chunk:height] = np.asarray(output_tmp, dtype=np.float32)
    t1 = timeit.default_timer()
    print("Done sinograms: {0}-{1}. Time {2}".format(height - last_chunk, height -␣
↪1, t1 - t0))

t1 = timeit.default_timer()
print("Done!!!. Total time cost: {}".format(t1 - t0))
```

- Processed sinograms in the hdf-file then can be converted to 16-bit tiff images (i.e. to be used by cone-beam reconstruction software provided by tomography-system suppliers). Otherwise, Astra Toolbox can be used for reconstruction without the need of this conversion step.

```
import timeit
import multiprocessing as mp
from joblib import Parallel, delayed
import numpy as np
import algotom.io.loadersaver as losa

input_file = "E:/tmp/projections_preprocessed.hdf"
output_base = "E:/tmp/tif_projections/"

data = losa.load_hdf(input_file, key_path='entry/data')
# Note that the shape of data has been changed after the previous step
# where sinograms are arranged along 0-axis. Now we want to save the data
# as projections which are arranged along 1-axis.
(height, depth, width) = data.shape

t0 = timeit.default_timer()
# For parallel writing tif-images
ncore = mp.cpu_count()
chunk_size = np.clip(ncore - 1, 1, depth - 1)
last_chunk = depth - chunk_size * (depth // chunk_size)

for i in np.arange(0, depth - last_chunk, chunk_size):
    mat_stack = data[:, i: i + chunk_size, :]
    mat_stack = np.uint16(mat_stack)  # Convert to 16-bit data for tif-format
    file_names = [(output_base + "/proj_" + ("0000" + str(j))[-5:] + ".tif") for j␣
↪in range(i, i + chunk_size)]
    # Save files in parallel
    Parallel(n_jobs=ncore, prefer="processes")(delayed(losa.save_image)(file_
↪names[j], mat_stack[:, j, :]) for j in range(chunk_size))

if last_chunk != 0:
    mat_stack = data[:, depth - last_chunk:depth, :]
```

(continues on next page)

(continued from previous page)

```
    mat_stack = np.uint16(mat_stack)  # Convert to 16-bit data for tif-format
    file_names = [(output_base + "/proj_" + ("0000" + str(j))[-5:] + ".tif") for j␣
↪in range(depth - last_chunk, depth)]
    # Save files in parallel
    Parallel(n_jobs=ncore, prefer="processes")(delayed(losa.save_image)(file_
↪names[j], mat_stack[:, j, :]) for j in range(last_chunk))

t1 = timeit.default_timer()
print("Done!!!. Total time cost: {}".format(t1 - t0))
```

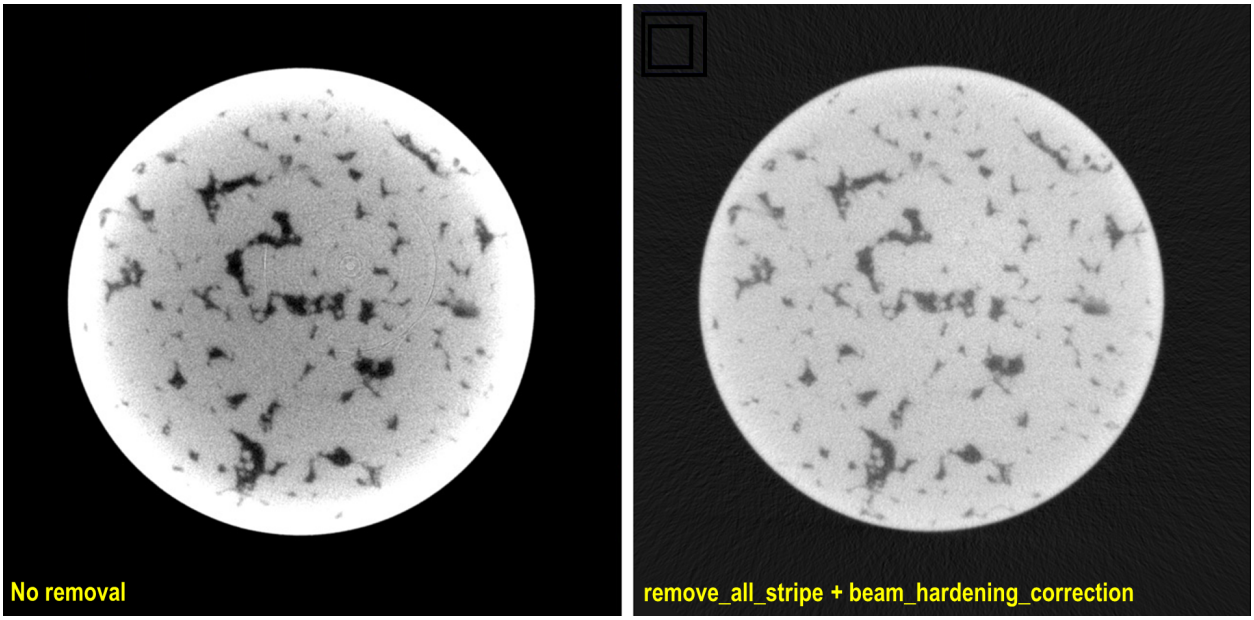


Fig. 1.4.9: Reconstructed images, before and after applied pre-processing methods, from projection-images acquired by a commercial cone-beam system. Data provided by Dr Mohammed Azeem

1.4.5 Complete workflow for processing tomographic data

This guide presents a comprehensive workflow for processing tomographic data, starting from raw data. In addition, it includes useful tips and explanations to users' common mistakes.

Assessing raw data

A typical tomographic dataset includes:

- Raw data acquired by a detector which are: projection images, flat-field image, and dark-field images.
- Metadata that provides information about the experimental setup, such as rotation angles, energy, pixel size, sample-detector distance, and any other relevant parameters.

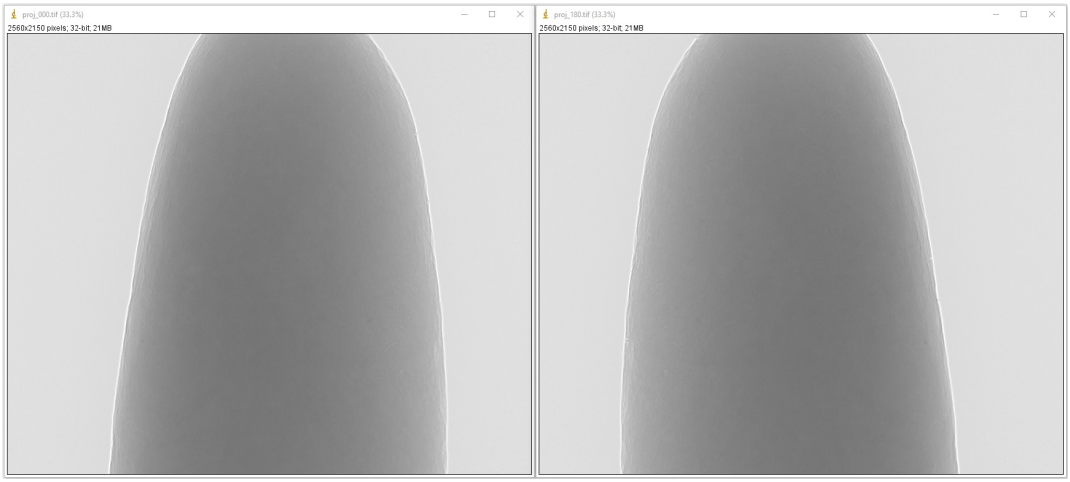
Visual inspection (using ImageJ) can help determine whether raw data are of sufficient quality. Users may want to perform the following checks:

- Verify that the first and the last projection are 180-degree apart:

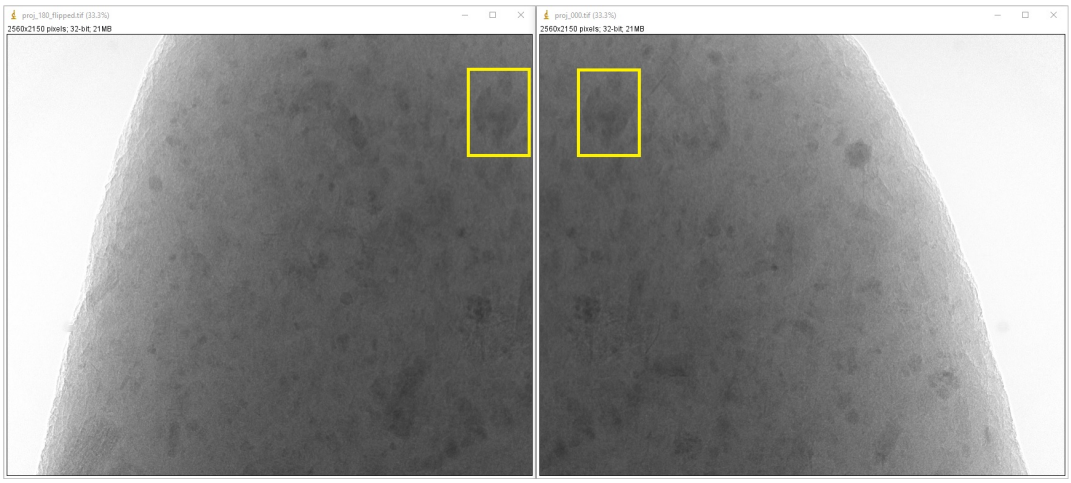
If raw data are in *hdf/h5/nxs format*, the projections can be extracted and saved as tif images as the following:

```
import numpy as np
import algotom.io.loadersaver as losa

file_path = "E:/Tomo_data/scan_68067.hdf"
output_base = "E:/tmp/extract_tifs/"
proj_path = "entry/projections"  # Refer section 1.2.1 to know how to get
                                 # path to a dataset in a hdf file.
flat_path = "entry/flats"
flat_img = np.mean(np.asarray(losa.load_hdf(file_path, flat_path)), axis=0)
nmean = np.mean(flat_img)
flat_img[flat_img == 0.0] = nmean  # To avoid zero division

proj_obj = losa.load_hdf(file_path, proj_path)  # hdf object
proj_0 = proj_obj[0, :, :] / flat_img
losa.save_image(output_base + "/proj_0.tif", proj_0)
proj_180 = proj_obj[-1, :, :] / flat_img
losa.save_image(output_base + "/proj_180.tif", proj_180)
```

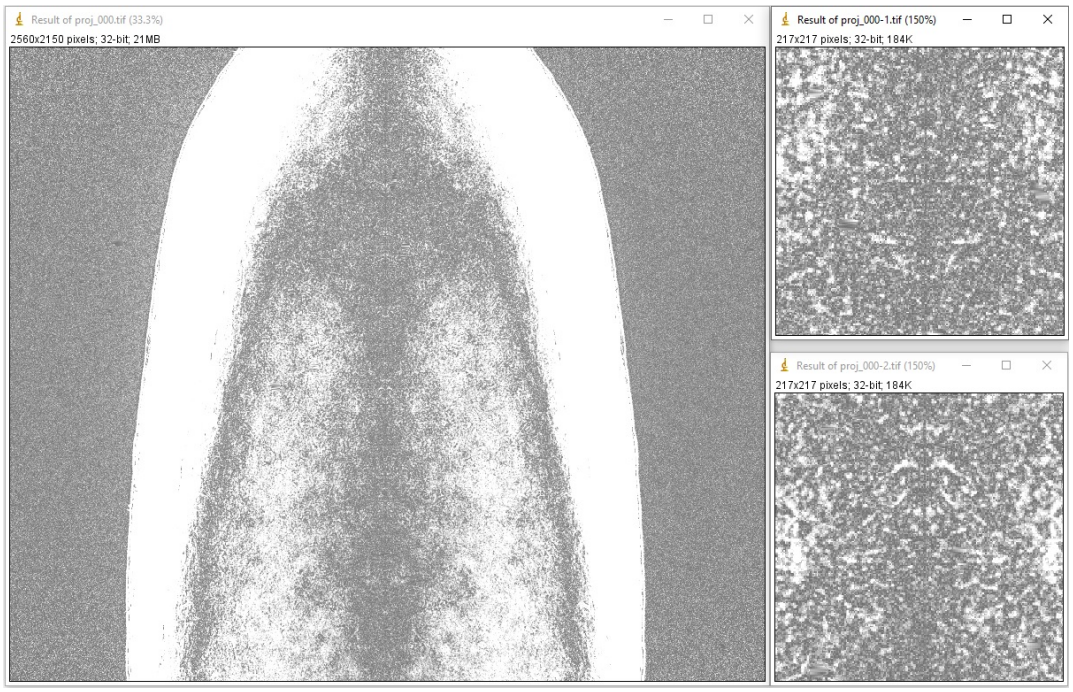
If the first and last projection are 180-degree apart, the second image should be a mirror reflection (left-right flip) of the first image.



- If data were acquired using a 360-degree scan with an offset rotation axis, it is important to verify that the rotation axis is positioned to one side of the field of view (FOV). This can be done by checking for an overlap between the 0-degree projection and the left-right flipped 180-degree projection image.

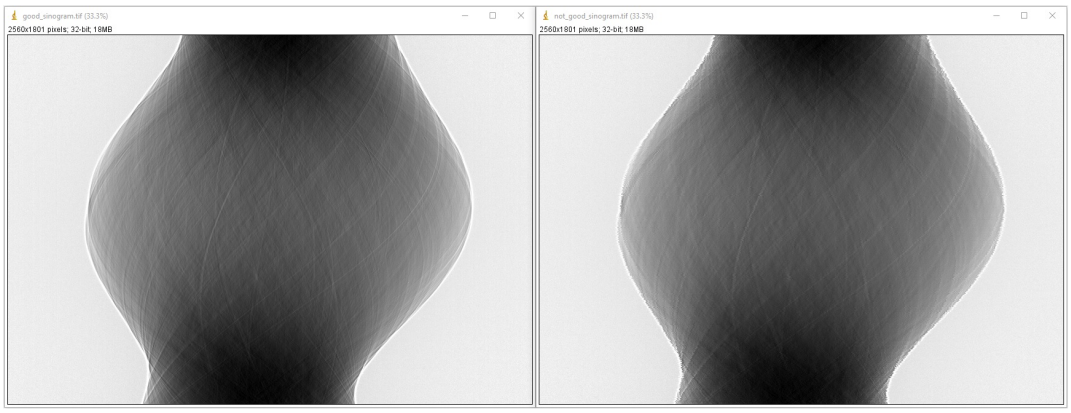


- Check if the rotation axis is tilted. This can be done by calculating the difference/average between the 0-degree projection and the 180-degree projection, then examining the resulting image for a symmetric line. If the x-location of the symmetric line is the same at the top and bottom of the image, the rotation axis is properly aligned.



  If a tilt is detected, the tilt angle can be accurately calculated by locating the center of rotation using sinograms generated at the top, middle, and bottom of the FOV; then applying a linear fit to the results. The resulting tilt angle can be used to correct the tilted tomographic images, as shown here.

- Ensure that projection images were acquired at evenly spaced angles and there was no stage jittering during the scan by inspecting a sinogram image:



If raw data are in hdf/h5/nxs format, the sinogram can be extracted and saved as tif format as follows:

```
import numpy as np
import algotom.io.loadersaver as losa

file_path = "E:/Tomo_data/scan_68067.hdf"
output_base = "E:/tmp/extract_tifs/"
proj_path = "entry/projections"  # Refer section 1.2.1 to know how to get
                                 # path to a dataset in a hdf file.
flat_path = "entry/flats"
flat_img = np.mean(np.asarray(losa.load_hdf(file_path, flat_path)), axis=0)
nmean = np.mean(flat_img)
flat_img[flat_img == 0.0] = nmean  # To avoid zero division

proj_obj = losa.load_hdf(file_path, proj_path)  # hdf object
(depth, height, width) = proj_obj.shape
sino_idx = height // 2
sinogram = proj_obj[:, sino_idx, :] / flat_img[sino_idx]
losa.save_image(output_base + "/sinogram.tif", sinogram)
```

If input data are in tif format, we need to convert them to the hdf format for fast extracting sinogram image:

```
import os
import shutil
import numpy as np
import algotom.io.converter as cvr
import algotom.io.loadersaver as losa

proj_path = "E:/Tomo_data/scan_68067/projections/"
flat_path = "E:/Tomo_data/scan_68067/flats/"

output_base = "E:/tmp/extract_tifs/"

flat_img = np.mean(np.asarray(
    [losa.load_image(file) for file in losa.find_file(flat_path + "/*tif*
↪")]), axis=0)
nmean = np.mean(flat_img)
flat_img[flat_img == 0.0] = nmean  # To avoid zero division
```

(continues on next page)

(continued from previous page)

```
# Convert data to hdf format for fast extracting sinograms.
hdf_file_path = output_base + "/hdf_converted/" + "tomo_data.hdf"
key_path = "entry/data"
cvr.convert_tif_to_hdf(proj_path, hdf_file_path, key_path=key_path)
proj_obj, hdf_obj = losa.load_hdf(hdf_file_path, key_path, return_file_
↪obj=True)
(depth, height, width) = proj_obj.shape

sino_idx = height // 2
sinogram = proj_obj[:, sino_idx, :] / flat_img[sino_idx]
losa.save_image(output_base + "/sinogram.tif", sinogram)
hdf_obj.close()
# Remove the hdf file if needs to
if os.path.isdir(output_base + "/hdf_converted/"):
    shutil.rmtree(output_base + "/hdf_converted/")
```

Reconstructing several slices

In high throughput tomographic systems, it's common that users want to quickly reconstruct only a few slices in order to verify the quality of the data or to locate the region of interest for higher resolution scans. This can be achieved by following these steps:

- Load the raw data and the corresponding flat-field and dark-field images. It's common to acquire multiple flat and dark images (usually between 10 and 50) and average them to improve the signal-to-noise (SNR) ratio. Once the flat and dark images have been averaged, they can be used for *flat-field correction*.

  If raw data are in tif format, we need to convert them to hdf format first:

```
import numpy as np
import algotom.io.loadersaver as losa
import algotom.io.converter as cvr
import algotom.prep.correction as corr
import algotom.prep.calculation as calc
import algotom.prep.removal as remo
import algotom.prep.filtering as filt
import algotom.rec.reconstruction as rec


proj_path = "E:/Tomo_data/scan_68067_tif/projections/"
flat_path = "E:/Tomo_data/scan_68067_tif/flats/"
dark_path = "E:/Tomo_data/scan_68067_tif/darks/"

output_base = "E:/output/rec_few_slices/"

# Load dark-field images and flat-field images.
flats = losa.get_tif_stack(flat_path)
darks = losa.get_tif_stack(dark_path)

# Convert tif images to hdf format for fast extracting sinograms.
file_path = output_base + "/tmp_/" + "tomo_data.hdf"
key_path = "entry/projections"
```

(continues on next page)

(continued from previous page)

```
cvr.convert_tif_to_hdf(proj_path, file_path, key_path=key_path,
                       option={"entry/flats": flats, "entry/darks": darks})
```

Working with a hdf file is straightforward as follows:

```
import numpy as np
import algotom.io.loadersaver as losa
import algotom.io.converter as cvr
import algotom.prep.correction as corr
import algotom.prep.calculation as calc
import algotom.prep.removal as remo
import algotom.prep.filtering as filt
import algotom.rec.reconstruction as rec


file_path = "E:/Tomo_data/scan_68067.hdf"
output_base = "E:/output/rec_few_slices/"
proj_path = "entry/projections"  # Refer section 1.2.1 to know how to get
                                 # path to a dataset in a hdf file.
flat_path = "entry/flats"
dark_path = "entry/darks"

# Load data, average flat and dark images
proj_obj = losa.load_hdf(file_path, proj_path)  # hdf object
(depth, height, width) = proj_obj.shape
flat_field = np.mean(np.asarray(losa.load_hdf(file_path, flat_path)),␣
↪axis=0)
dark_field = np.mean(np.asarray(losa.load_hdf(file_path, dark_path)),␣
↪axis=0)

# If the rotation angles are not provided, e.g. from metadata of the HDF␣
↪file,
# they can be generated automatically in a reconstruction method. Note that
# the rotation angles are in radians as requested by the reconstruction␣
↪method.
# To rotate the reconstructed image, simply add an offset angle using the␣
↪following method:
offset_angle = 0.0  # Degree
angles = np.deg2rad(offset_angle + np.linspace(0.0, 180.0, depth))

# Specify the range of slices to be reconstructed
start_slice = 100
stop_slice = height - 100
step_slice = 100

# Extract sinogram at the middle for calculating the center of rotation
idx = height // 2
sinogram = corr.flat_field_correction(proj_obj[:, idx, :], flat_field[idx],
                                      dark_field[idx])
center = calc.find_center_vo(sinogram)
print("Center of rotation: {}".format(center))
```

(continues on next page)

(continued from previous page)

```
# Extract sinograms and perform flat-field correction
for idx in range(start_slice, stop_slice + 1, step_slice):
    sinogram = corr.flat_field_correction(proj_obj[:, idx, :], flat_
↪field[idx],
                                          dark_field[idx])
    # Apply pre-processing methods
```

- Apply pre-processing methods: zinger removal, ring artifact removal, and/or denoising to sinograms. Note that there are many choices for ring-removal methods, but for this step we may just want a fast method.

```
# ...
    # Apply pre-processing methods
    sinogram = remo.remove_zinger(sinogram, 0.08)
    sinogram = remo.remove_stripe_based_normalization(sinogram, 15)
    sinogram = filt.fresnel_filter(sinogram, 100)
    # Perform reconstruction
```

- Perform reconstruction and save the results to tif. Algotom provides reconstruction methods that can run on either CPU or GPU. It also provides the wrappers of the *gridrec* method, available in Tomopy, which is very fast for CPU-only computers; and iterative methods available in Astra Toolbox. Note that if users want to use these additional wrappers, Tomopy and Astra will need to be installed along with Algotom.

```
# ...
    # Perform reconstruction
    # Using a cpu method
    rec_img = rec.dfi_reconstruction(sinogram, center, angles=angles,
                                     apply_log=True)
    # # Other options:
    # # Using a gpu method
    # rec_img = rec.fbp_reconstruction(sinogram, center, angles=angles,
    #                                  apply_log=True, gpu=True)
    # # Using a cpu method, available in Tomopy
    # rec_img = rec.gridrec_reconstruction(sinogram, center, angles=angles,
    #                                      apply_log=True)
    # # Using a gpu method, available in Astra Toolbox
    # rec_img = rec.astra_reconstruction(sinogram, center, angles=angles,
    #                                    method="SIRT_CUDA", num_iter=150,
    #                                    apply_log=True)
    out_file = output_base + "/rec_" + ("00000" + str(idx))[-5:] + ".tif"
    losa.save_image(out_file, rec_img)
```

Finding the center of rotation

Algotom offers several methods for automatically calculating the center of rotation (COR), which refers to the rotation axis of the sample stage with respect to the FOV. These methods work on different processing spaces (Fig. 1.4.10) and can be selected according to specific types of input images.

- Methods that work in the projection space are the fastest and simplest, but they are also the least reliable.

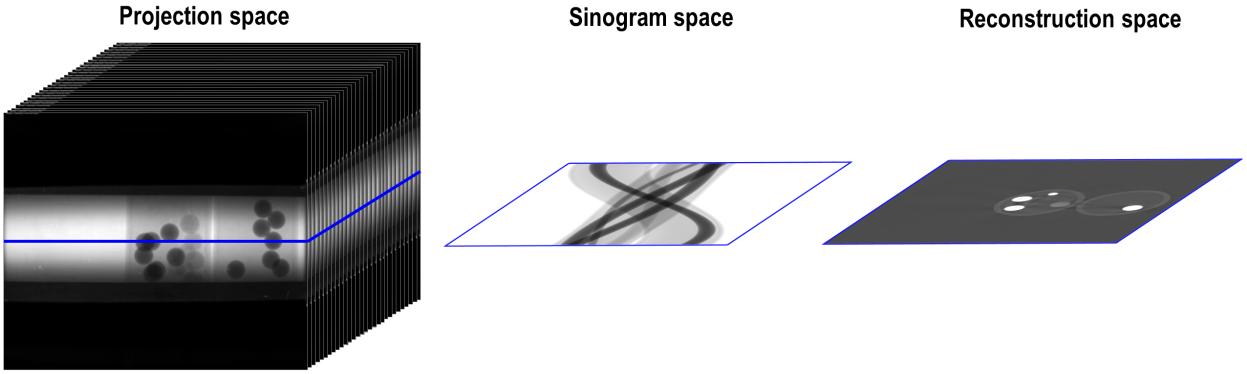


Fig. 1.4.10: Different processing spaces can be used for finding the center of rotation.

```
import timeit
import algotom.prep.calculation as calc

# Data is at: https://doi.org/10.5281/zenodo.1443567
# Steps for loading data are similar to above sections

proj_0 = proj_obj[0, :, :] / flat_field
proj_180 = proj_obj[-1, :, :] / flat_field

print("Image size: {}".format(flat_field.shape))
t0 = timeit.default_timer()
center = calc.find_center_based_phase_correlation(proj_0, proj_180)
t1 = timeit.default_timer()
print("Using phase correlation. Center: {0}. Time: {1}".format(center, t1 -
↪t0))

t0 = timeit.default_timer()
center = calc.find_center_projection(proj_0, proj_180, chunk_height=100)
t1 = timeit.default_timer()
print("Using image correlation. Center: {0}. Time: {1}".format(center, t1 -
↪t0))
```

```
>>>
Image size: (2160, 2560)
Using phase correlation. Center: 1272.8564415436447. Time: 1.
↪6949839999999998
Using image correlation. Center: 1272.8176879882812. Time: 15.
↪652110699999998
```

- The most reliable method for automatically calculating the center of rotation is a method that works on a 180-degree sinogram image, as proposed in [R21]. This method has been extensively tested on 2,000 micro-tomography datasets, achieving a success rate of 98%. A visual explanation of how the method works is provided in Fig. 1.4.11.

```
idx = height // 2
sinogram = corr.flat_field_correction(proj_obj[:, idx, :], flat_field[idx],
                                      dark_field[idx])
```

(continues on next page)

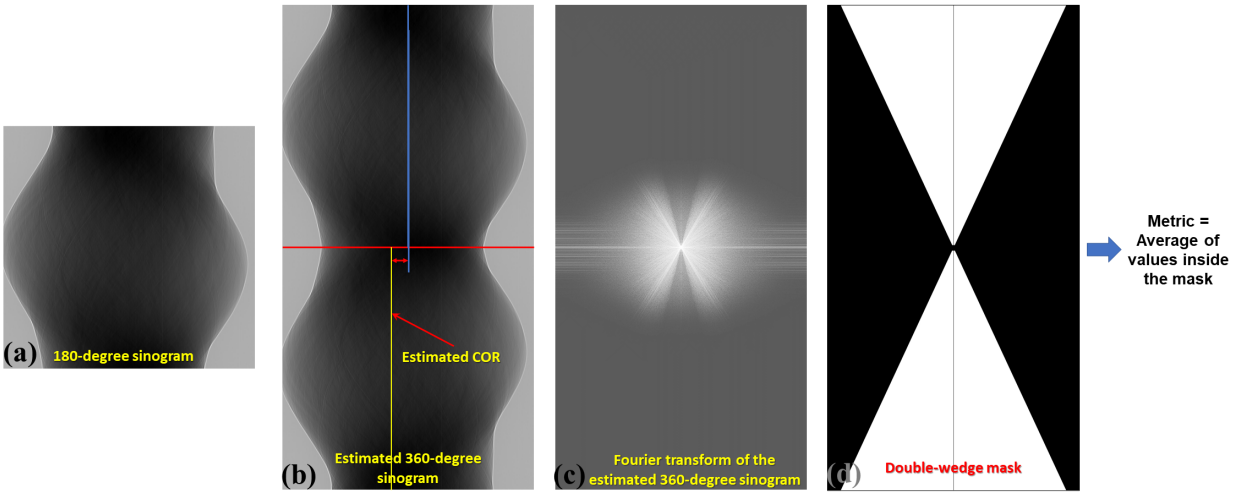


Fig. 1.4.11: Explanation of how the autocentering method in the sinogram space works.

(continued from previous page)

```
t0 = timeit.default_timer()
radius = width // 16
mid = width // 2
# Enable parallel computing using the "ncore" option.
center = calc.find_center_vo(sinogram, start=mid - radius, stop=mid +␣
↪radius,
                             ncore=8)
t1 = timeit.default_timer()
print("Using sinogram metric. Center: {0}. Time: {1}".format(center, t1 -␣
↪t0))
```

```
>>>
Using sinogram metric. Center: 1272.75. Time: 8.0966264
```

The method's default parameters work for most X-ray microtomography datasets, as extensively tested. However, users can adjust these parameters, such as the *ratio* and *ver_drop* parameters, to suit their data. A unique feature of this method is the ability to average multiple sinograms to improve the signal-to-noise ratio and use the result as input for the method. Note that strongly smoothed or blurry sinograms resulting from denoising methods or phase-retrieval methods can impact the performance of this method.

- Another method, available from Algotom 1.3, works in the reconstruction space and evaluates a slice metric to determine the best center of rotation. This method is slower than the other methods and is most suitable for performing small, fine searching ranges around the coarse center found by previous methods. It may not be suitable for use on low SNR data.

```
import algotom.rec.reconstruction as rec

t0 = timeit.default_timer()
center = rec.find_center_based_slice_metric(sinogram, mid-radius, mid +␣
↪radius,
                                            zoom=0.5, method='fbp',␣
```

(continues on next page)

(continued from previous page)

```
↪gpu=True,
                                                apply_log=True)
t1 = timeit.default_timer()
print("Using slice metric. Reconstruction method: FBP. Center: {0}. Time:
↪{1}".format(center, t1 - t0))

t0 = timeit.default_timer()
center = rec.find_center_based_slice_metric(sinogram, mid-radius, mid +␣
↪radius,
                                                zoom=0.5, method='dfi',
                                                apply_log=True)
t1 = timeit.default_timer()
print("Using slice metric. Reconstruction method: DFI. Center: {0}. Time:
↪{1}".format(center, t1 - t0))

t0 = timeit.default_timer()
center = rec.find_center_based_slice_metric(sinogram, mid-radius, mid +␣
↪radius,
                                                zoom=0.5, method='gridrec',
                                                apply_log=True)
t1 = timeit.default_timer()
print("Using slice metric. Reconstruction method: Gridrec. Center: {0}.␣
↪Time: {1}".format(center, t1 - t0))
```

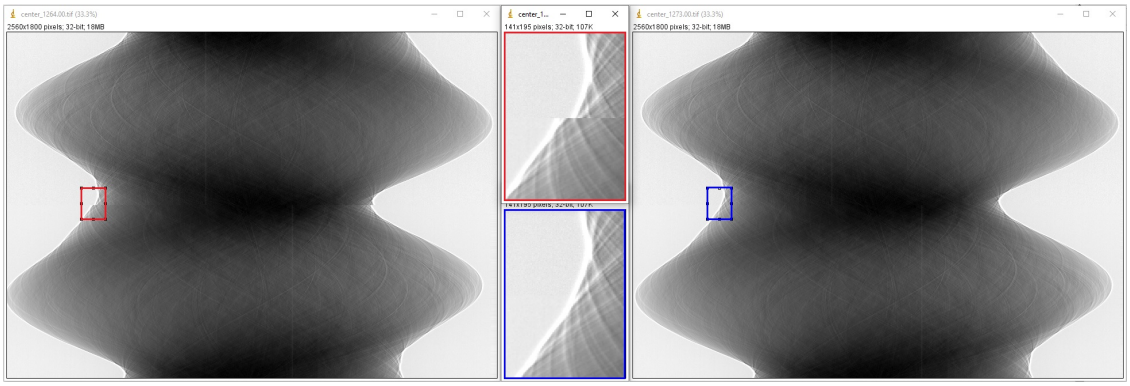
```
>>>
Using slice metric. Reconstruction method: FBP. Center: 1272.5. Time: 104.
↪3659703
Using slice metric. Reconstruction method: DFI. Center: 1272.5. Time: 85.
↪9248028
Using slice metric. Reconstruction method: Gridrec. Center: 1272.5. Time:␣
↪14.54944309999999
```

If users would like to apply a customized function for calculating a slice metric, it can be done as follows:

```
def measure_metric(mat, n=2):
    metric = np.abs(np.mean(mat[mat < 0.0])) ** n
    return metric
center = rec.find_center_based_slice_metric(sinogram, mid-10, mid + 10,
                                                zoom=1.0, method='fbp',␣
↪gpu=True,
                                                apply_log=True,
                                                metric_function=measure_metric,␣
↪n=2)
```

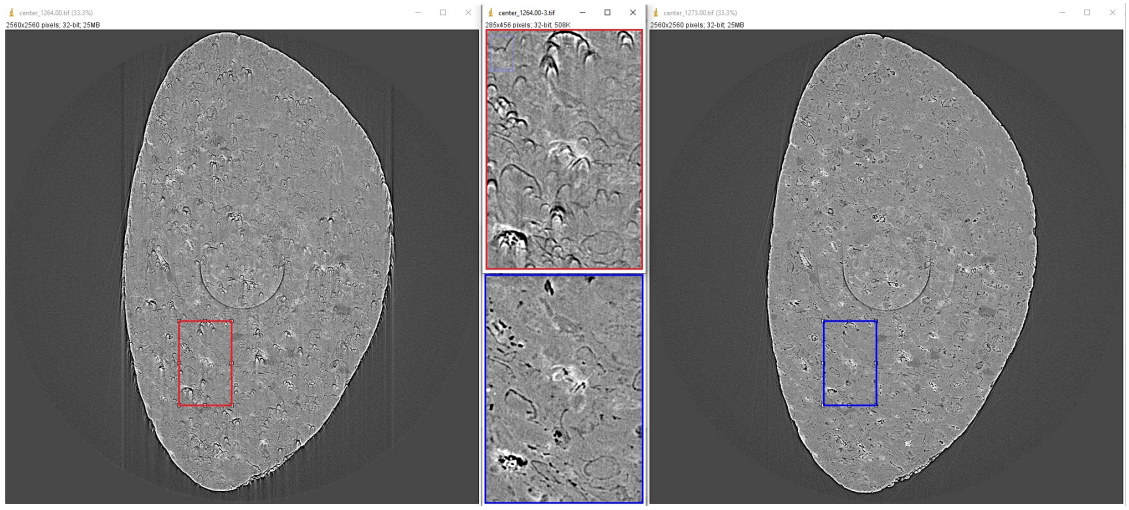
- If the automated methods fail to find the center of rotation, users can rely on the following manual methods (available from Algotom 1.3) to locate it:
  - The first manual method involves generating a list of 360-degree sinograms created from the input 180-degree sinogram using a list of estimated CORs. Users can find the best COR by identifying the generated sinogram that has a continuous transition between the two halves of the sinogram, as illustrated in the figure below.

```
import algotom.util.utility as util

output_base = "E:/tmp/manual_finding/using_sinograms/"
util.find_center_visual_sinograms(sinogram, output_base, width // 2 - 20, width␣
↪// 2 + 20,
                                  step=1, zoom=1.0)
```



– The second manual method involves reconstructing a list of slices using a list of estimated CORs. Users can find the best COR by visually inspecting the reconstructed slices and selecting the one with the least streak artifacts.

```
output_base = "E:/tmp/manual_finding/using_slices/"
util.find_center_visual_slices(sinogram, output_base, width // 2 - 20,
                               width // 2 + 20, 1, zoom=1.0, method="fbp",␣
↪gpu=True)
```



Tweaking parameters of preprocessing methods

When reconstructing synchrotron-based X-ray microtomography data, users often spend most of time tweaking parameters of preprocessing methods such as ring artifact removal or contrast-enhancement methods. We can setup different workflows to test methods as below:

- To compare different ring removal methods; note that in Algotom, some well-known methods are improved and have additional options for customization:

```
# Steps for loading data are similar to above sections

# To create new output-folder for each time of running the script.
output_base0 = "E:/tmp/compare_ring_removal_methods/"
folder_name = losa.make_folder_name(output_base0, name_prefix="Ring_removal", zero_
↪prefix=3)
output_base = output_base0 + "/" + folder_name + "/"

idx = height // 2
sinogram = corr.flat_field_correction(proj_obj[:, idx, :], flat_field[idx],
                                      dark_field[idx])
center = calc.find_center_vo(sinogram)

# Using the combination of algorithms
sinogram1 = remo.remove_all_stripe(sinogram, snr=3.0, la_size=51, sm_size=21)
rec_img = rec.fbp_reconstruction(sinogram1, center)
losa.save_image(output_base + "/remove_all_stripe.tif", rec_img)

# Using the sorting-based method
sinogram2 = remo.remove_stripe_based_sorting(sinogram, size=21, dim=1)
rec_img = rec.fbp_reconstruction(sinogram2, center)
losa.save_image(output_base + "/remove_stripe_based_sorting.tif", rec_img)

# Using the fitting-based method
sinogram3 = remo.remove_stripe_based_fitting(sinogram, order=2, sigma=10)
rec_img = rec.fbp_reconstruction(sinogram3, center)
losa.save_image(output_base + "/remove_stripe_based_fitting.tif", rec_img)

# Using the filtering-based method
sinogram4 = remo.remove_stripe_based_filtering(sinogram, sigma=3, size=21, dim=1,
                                               sort=True)
rec_img = rec.fbp_reconstruction(sinogram4, center)
losa.save_image(output_base + "/remove_stripe_based_filtering.tif", rec_img)

# Using the 2d filtering and sorting-based method
sinogram5 = remo.remove_stripe_based_2d_filtering_sorting(sinogram, sigma=3,
                                                          size=21, dim=1)
rec_img = rec.fbp_reconstruction(sinogram5, center)
losa.save_image(output_base + "/remove_stripe_based_2d_filtering_sorting.tif", rec_
↪img)

# Using the interpolation-based method
sinogram6 = remo.remove_stripe_based_interpolation(sinogram, snr=3.0, size=51)
rec_img = rec.fbp_reconstruction(sinogram6, center)
losa.save_image(output_base + "/remove_stripe_based_interpolation.tif", rec_img)
```

(continues on next page)

(continued from previous page)

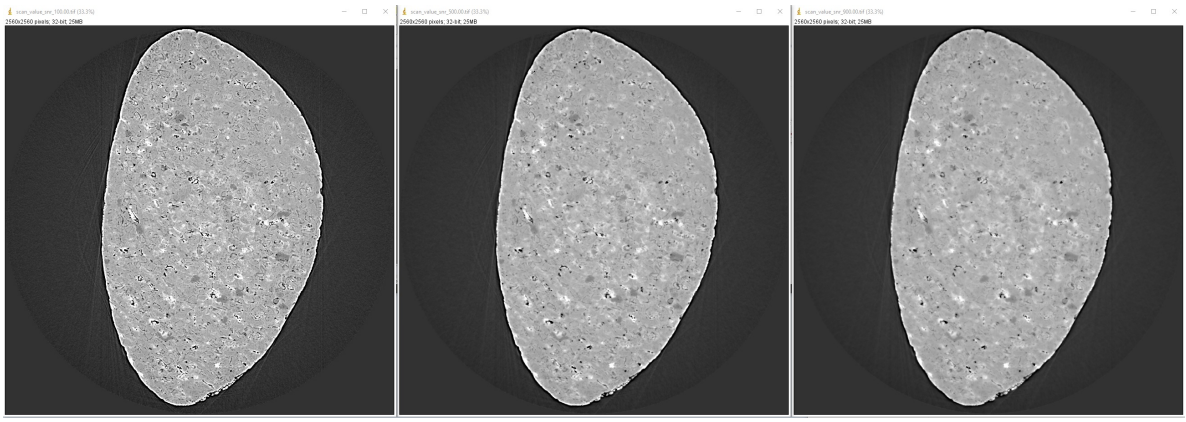
```
# Using the normalization-based method
sinogram7 = remo.remove_stripe_based_normalization(sinogram, sigma=15)
rec_img = rec.fbp_reconstruction(sinogram7, center)
losa.save_image(output_base + "/remove_stripe_based_normalization.tif", rec_img)

# Using the regularization-based method
sinogram8 = remo.remove_stripe_based_regularization(sinogram, alpha=0.0005,
                                                    num_chunk=1, apply_log=True,
                                                    sort=False)
rec_img = rec.fbp_reconstruction(sinogram8, center)
losa.save_image(output_base + "/remove_stripe_based_regularization.tif", rec_img)

# Using the fft-based method
sinogram9 = remo.remove_stripe_based_fft(sinogram, u=20, n=8, v=1, sort=False)
rec_img = rec.fbp_reconstruction(sinogram9, center)
losa.save_image(output_base + "/remove_stripe_based_fft.tif", rec_img)

# Using the wavelet-fft-based method
sinogram10 = remo.remove_stripe_based_wavelet_fft(sinogram, level=5, size=1,
                                                  wavelet_name='db9',
                                                  window_name='gaussian', sort=False)
rec_img = rec.fbp_reconstruction(sinogram10, center)
losa.save_image(output_base + "/remove_stripe_based_wavelet_fft.tif", rec_img)
```

- To perform scanning a parameter of a ring removal method

```
# To create new output-folder for each time of running the script.
output_base0 = "E:/tmp/scan_parameters/"
folder_name = losa.make_folder_name(output_base0, name_prefix="Scan_ratio", zero_
↪prefix=3)
output_base = output_base0 + "/" + folder_name + "/"

for value in np.linspace(1.1, 3.0, 20):
    sinogram1 = remo.remove_all_stripe(sinogram, snr=value, la_size=51, sm_size=21)
    name = "snr_{0:2.2f}".format(value)
    rec_img = rec.fbp_reconstruction(sinogram1, center)
    losa.save_image(output_base + "/scan_value_" + name + ".tif", rec_img)
```

or a contrast-enhancement method

```
for ratio in np.arange(100, 1600, 400):
    sinogram1 = filt.fresnel_filter(sinogram, ratio, dim=1)
    name = "snr_{0:4.2f}".format(ratio)
    rec_img = rec.fbp_reconstruction(sinogram1, center)
    losa.save_image(output_base + "/scan_value_" + name + ".tif", rec_img)
```

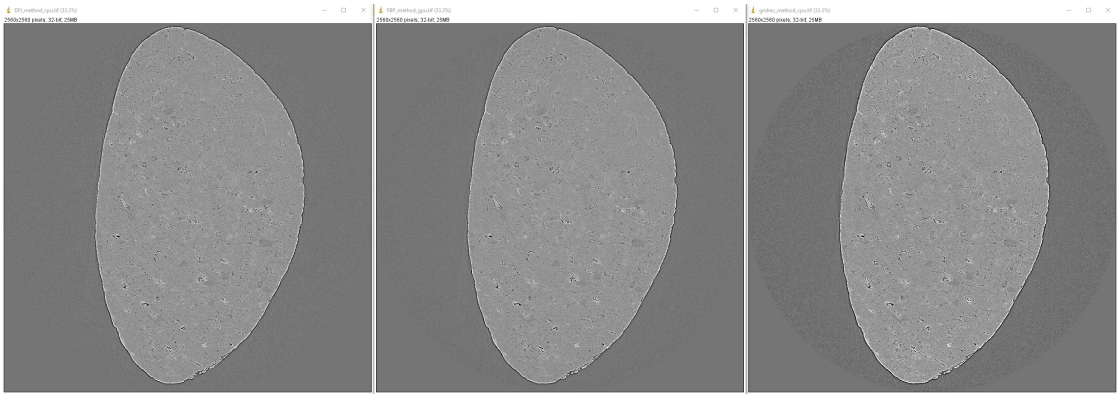


Choosing a reconstruction method

The quality of reconstructed data in synchrotron-based X-ray microtomography depends heavily on the preprocessing methods applied. If the number of acquired projections is standard and the data are properly cleaned, the choice of reconstruction method will have less impact on the quality of the final results. Therefore, users can choose a reconstruction method based on the availability of computing resources.

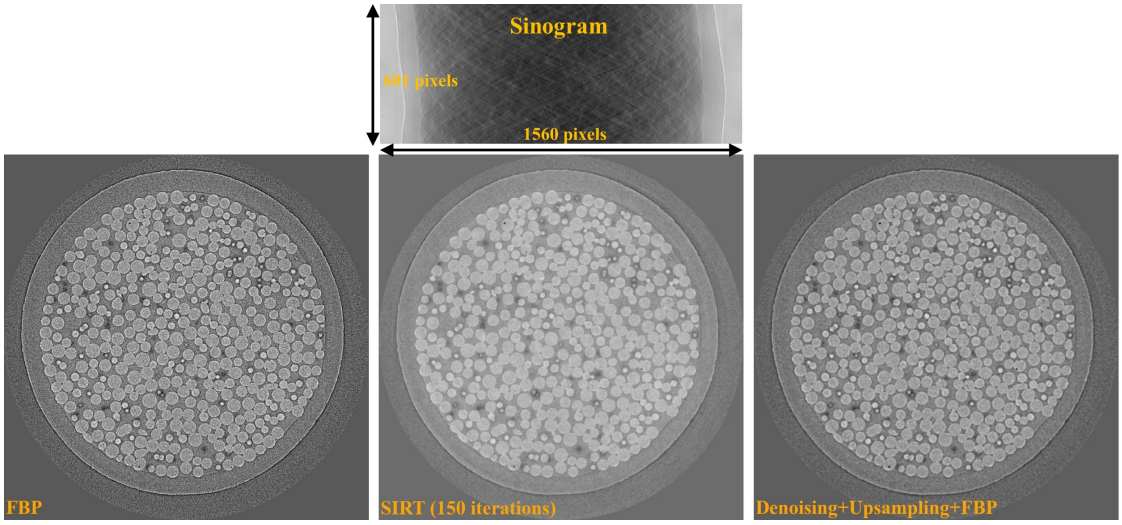
```
output_base = "E:/tmp/compare_reconstruction_methods/"

# Using the direct Fourier inversion method (CPU)
t0 = timeit.default_timer()
rec_img = rec.dfi_reconstruction(sinogram, center)
print("Reconstructed image size: {}".format(rec_img.shape))
losa.save_image(output_base + "/DFI_method_cpu.tif", rec_img)
t1 = timeit.default_timer()
print("Using the DFI method (CPU). Time: {}".format(t1 - t0))

# Using the filtered back-projection method (CPU)
t0 = timeit.default_timer()
rec_img = rec.fbp_reconstruction(sinogram, center, gpu=False)
losa.save_image(output_base + "/FBP_method_cpu.tif", rec_img)
t1 = timeit.default_timer()
print("Using the FBP method (CPU). Time: {}".format(t1 - t0))

# Using the filtered back-projection method (GPU)
t0 = timeit.default_timer()
rec_img = rec.fbp_reconstruction(sinogram, center, gpu=True)
losa.save_image(output_base + "/FBP_method_gpu.tif", rec_img)
t1 = timeit.default_timer()
print("Using the FBP method (GPU). Time: {}".format(t1 - t0))

# Using the gridrec method (CPU)
t0 = timeit.default_timer()
rec_img = rec.gridrec_reconstruction(sinogram, center, ncore=1)
losa.save_image(output_base + "/gridrec_method_cpu.tif", rec_img)
t1 = timeit.default_timer()
print("Using the gridrec method (CPU). Time: {}".format(t1 - t0))
```

```
>>>
Reconstructed image size: (2560, 2560)
Using the DFI method (CPU). Time: 12.7383788
Using the FBP method (CPU). Time: 5.827241100000002
Using the FBP method (GPU). Time: 3.001648600000003
Using the gridrec method (CPU). Time: 1.7366413999999963
```



When dealing with undersampled sinogram, iterative reconstruction methods like SIRT (Simultaneous iterative reconstruction technique) can be advantageous over Fourier-based methods. However, iterative methods are computationally expensive. A workaround is to improve the Fourier-based methods by applying denoising and upsampling methods (Algotom >=1.3) to the sinogram.

```
output_base = "E:/tmp/improve_fft_method/"

print("Sinogram size {}".format(sinogram.shape))
# Using FBP method
rec_img1 = rec.fbp_reconstruction(sinogram, center, filter_name="hann")
losa.save_image(output_base + "/fbp_recon.tif", rec_img1)

# Using SIRT method with 150 number of iterations
rec_img2 = rec.astra_reconstruction(sinogram, center, method="SIRT_CUDA", num_
↪iter=150)
losa.save_image(output_base + "/sirt_recon.tif", rec_img2)

# Denosing + upsampling sinogram + FBP reconstruction
sinogram = filt.fresnel_filter(sinogram, 100)
sinogram = corr.upsample_sinogram(sinogram, 2, center)
print("Upsampled sinogram size {}".format(sinogram.shape))
rec_img3 = rec.fbp_reconstruction(sinogram, center, filter_name="hann")
losa.save_image(output_base + "/fbp_denoising_upsampling.tif", rec_img3)
```



Performing full reconstruction

After completing all the steps for selecting parameters and testing methods, we can proceed with the full reconstruction process. The main difference compared to the previous steps is that sinograms are processed in chunks, which reduces I/O overhead and utilizes parallel processing. The following codes are available here for both tif and hdf input formats, but we can break down the workflow and provide detailed explanations:

- Import the necessary modules from Algotom, specify the input and output paths, and add options to make it easier to modify the workflow later on.

```
import numpy as np
import timeit
import algotom.io.loadersaver as losa
import algotom.prep.correction as corr
import algotom.prep.calculation as calc
import algotom.rec.reconstruction as rec
import algotom.prep.removal as remo
import algotom.prep.filtering as filt
import algotom.util.utility as util

# Input file
file_path = "E:/Tomo_data/scan_68067.hdf"

# Specify output path, create new folder each time of running to avoid overwriting␣
↪data.
output_base0 = "E:/full_reconstruction/"
folder_name = losa.make_folder_name(output_base0, name_prefix="recon", zero_
↪prefix=3)
output_base = output_base0 + "/" + folder_name + "/"

# Optional parameters
start_slice = 10
stop_slice = -1
chunk = 100  # Number of slices to be reconstructed in one go. Adjust to suit RAM␣
↪or GPU memory.
ncore = 16  # Number of cpu-core for parallel processing. Set to None for␣
```

(continues on next page)

(continued from previous page)

```
↪autoselecting.
output_format = "tif"  # "tif" or "hdf".
preprocessing = True  # Clean data before reconstruction.

# Give alias to a reconstruction method which is convenient for later change
# recon_method = rec.dfi_reconstruction
# recon_method = rec.fbp_reconstruction
recon_method = rec.gridrec_reconstruction # Fast cpu-method. Must install Tomopy.
# recon_method = rec.astra_reconstruction # To use iterative methods. Must install␣
↪Astra.

# Provide metadata for loading hdf file, get data shape and rotation angles.
proj_path = "/entry/projections"
flat_path = "/entry/flats"
dark_path = "/entry/darks"
angle_path = "/entry/rotation_angle"
```

- Load dark-field images, flat-field images, rotation angles; and calculate the center of rotation.

```
t_start = timeit.default_timer()
print("------------------------------------------------------------------")
print("-----------------------------Start-----------------------------\n")
print("1 -> Load dark-field and flat-field images, average each result")
# Load data, average flat and dark images, get data shape and rotation angles.
proj_obj = losa.load_hdf(file_path, proj_path)  # hdf object
(depth, height, width) = proj_obj.shape
flat_field = np.mean(np.asarray(losa.load_hdf(file_path, flat_path)), axis=0)
dark_field = np.mean(np.asarray(losa.load_hdf(file_path, dark_path)), axis=0)
angles = np.deg2rad(np.squeeze(np.asarray(losa.load_hdf(file_path, angle_path))))
(depth, height, width) = proj_obj.shape

print("2 -> Calculate the center-of-rotation")
# Extract sinogram at the middle for calculating the center of rotation
index = height // 2
sinogram = corr.flat_field_correction(proj_obj[:, index, :], flat_field[index, :],
                                      dark_field[index, :])
center = calc.find_center_vo(sinogram)
print("Center-of-rotation is {}".format(center))
```

- Loop through the sinograms chunk-by-chunk, apply the selected pre-processing methods in parallel, and perform the reconstruction.

```
if (stop_slice == -1) or (stop_slice > height):
    stop_slice = height
total_slice = stop_slice - start_slice
if output_format == "hdf":
    # Note about the change of data-shape
    recon_hdf = losa.open_hdf_stream(output_base + "/recon_data.hdf",
                                     (total_slice, width, width),
                                     key_path='entry/data',
                                     data_type='float32', overwrite=True)
t_load = 0.0
```

(continues on next page)

(continued from previous page)

```
t_prep = 0.0
t_rec = 0.0
t_save = 0.0
chunk = np.clip(chunk, 1, total_slice)
last_chunk = total_slice - chunk * (total_slice // chunk)

# Perform full reconstruction
for i in np.arange(start_slice, start_slice + total_slice - last_chunk, chunk):
    start_sino = i
    stop_sino = start_sino + chunk
    # Load data, perform flat-field correction
    t0 = timeit.default_timer()
    sinograms = corr.flat_field_correction(
        proj_obj[:, start_sino:stop_sino, :],
        flat_field[start_sino:stop_sino, :],
        dark_field[start_sino:stop_sino, :])
    t1 = timeit.default_timer()
    t_load = t_load + t1 - t0

    # Perform pre-processing
    if preprocessing:
        t0 = timeit.default_timer()
        sinograms = util.apply_method_to_multiple_sinograms(sinograms,
                                                            "remove_zinger",
                                                            [0.08, 1],
                                                            ncore=ncore,
                                                            prefer="threads")
        sinograms = util.apply_method_to_multiple_sinograms(sinograms,
                                                            "remove_all_stripe",
                                                            [3.0, 51, 21],
                                                            ncore=ncore,
                                                            prefer="threads")
        sinograms = util.apply_method_to_multiple_sinograms(sinograms,
                                                            "fresnel_filter",
                                                            [200, 1],
                                                            ncore=ncore,
                                                            prefer="threads")
        t1 = timeit.default_timer()
        t_prep = t_prep + t1 - t0

    # Perform reconstruction
    t0 = timeit.default_timer()
    recon_imgs = recon_method(sinograms, center, angles=angles, ncore=ncore)
    t1 = timeit.default_timer()
    t_rec = t_rec + t1 - t0

    # Save output
    t0 = timeit.default_timer()
    if output_format == "hdf":
        recon_hdf[start_sino - start_slice:stop_sino - start_slice] = np.
↪moveaxis(recon_imgs, 1, 0)
    else:
```

(continues on next page)

(continued from previous page)

```
        for j in range(start_sino, stop_sino):
            out_file = output_base + "/rec_" + ("0000" + str(j))[-5:] + ".tif"
            losa.save_image(out_file, recon_imgs[:, j - start_sino, :])
    t1 = timeit.default_timer()
    t_save = t_save + t1 - t0
    t_stop = timeit.default_timer()
    print("Done slice: {0} - {1} . Time {2}".format(start_sino, stop_sino,
                                                   t_stop - t_start))
if last_chunk != 0:
    start_sino = start_slice + total_slice - last_chunk
    stop_sino = start_sino + last_chunk

    # Load data, perform flat-field correction
    t0 = timeit.default_timer()
    sinograms = corr.flat_field_correction(
        proj_obj[:, start_sino:stop_sino, :],
        flat_field[start_sino:stop_sino, :],
        dark_field[start_sino:stop_sino, :])
    t1 = timeit.default_timer()
    t_load = t_load + t1 - t0

    # Perform pre-processing
    if preprocessing:
        t0 = timeit.default_timer()
        sinograms = util.apply_method_to_multiple_sinograms(sinograms,
                                                            "remove_zinger",
                                                            [0.08, 1],
                                                            ncore=ncore,
                                                            prefer="threads")
        sinograms = util.apply_method_to_multiple_sinograms(sinograms,
                                                            "remove_all_stripe",
                                                            [3.0, 51, 21],
                                                            ncore=ncore,
                                                            prefer="threads")
        sinograms = util.apply_method_to_multiple_sinograms(sinograms,
                                                            "fresnel_filter",
                                                            [200, 1],
                                                            ncore=ncore)
        t1 = timeit.default_timer()
        t_prep = t_prep + t1 - t0

    # Perform reconstruction
    t0 = timeit.default_timer()
    recon_imgs = recon_method(sinograms, center, angles=angles, ncore=ncore)
    t1 = timeit.default_timer()
    t_rec = t_rec + t1 - t0

    # Save output
    t0 = timeit.default_timer()
    if output_format == "hdf":
        recon_hdf[start_sino - start_slice:stop_sino - start_slice] = np.
↪moveaxis(recon_imgs, 1, 0)
```

(continues on next page)

(continued from previous page)

```
    else:
        for j in range(start_sino, stop_sino):
            out_file = output_base + "/rec_" + ("0000" + str(j))[-5:] + ".tif"
            losa.save_image(out_file, recon_imgs[:, j - start_sino, :])
    t1 = timeit.default_timer()
    t_save = t_save + t1 - t0
    t_stop = timeit.default_timer()
    print("Done slice: {0} - {1} . Time {2}".format(start_sino, stop_sino,
                                                   t_stop - t_start))
print("-------------------------------------------------------------------")
print("-----------------------------Done----------------------------------")
print("Loading data cost: {0:0.2f}s".format(t_load))
print("Preprocessing cost: {0:0.2f}s".format(t_prep))
print("Reconstruction cost: {0:0.2f}s".format(t_rec))
print("Saving output cost: {0:0.2f}s".format(t_save))
print("Total time cost : {0:0.2f}s".format(t_stop - t_start))
```

```
>>>
-------------------------------------------------------------------
-----------------------------Start---------------------------------

1 -> Load dark-field and flat-field images, average each result
2 -> Calculate the center-of-rotation
Center-of-rotation is 1272.75
Done slice: 10 - 110 . Time 189.6021034
Done slice: 110 - 210 . Time 366.9538149
Done slice: 210 - 310 . Time 579.1721645
Done slice: 310 - 410 . Time 783.6394176
Done slice: 410 - 510 . Time 1001.0833168
Done slice: 510 - 610 . Time 1206.3565348
Done slice: 610 - 710 . Time 1415.9822423
Done slice: 710 - 810 . Time 1630.9875868
Done slice: 810 - 910 . Time 1844.1762275
Done slice: 910 - 1010 . Time 2052.5243417
Done slice: 1010 - 1110 . Time 2266.1704849000002
Done slice: 1110 - 1210 . Time 2485.4279775
Done slice: 1210 - 1310 . Time 2695.1756578000004
Done slice: 1310 - 1410 . Time 2902.663489
Done slice: 1410 - 1510 . Time 3122.5606983000002
Done slice: 1510 - 1610 . Time 3333.1580989000004
Done slice: 1610 - 1710 . Time 3545.0758953000004
Done slice: 1710 - 1810 . Time 3758.1900975000003
Done slice: 1810 - 1910 . Time 3974.6899012000003
Done slice: 1910 - 2010 . Time 4181.2648382
Done slice: 2010 - 2110 . Time 4389.6914713999995
Done slice: 2110 - 2160 . Time 4511.7352912
-------------------------------------------------------------------
-----------------------------Done----------------------------------
Loading data cost: 675.88s
Preprocessing cost: 3213.10s
Reconstruction cost: 337.11s
Saving output cost: 276.67s
```

(continues on next page)

(continued from previous page)

```
Total time cost : 4511.74s
```

As shown in the time cost list above, the most time-consuming step is pre-processing, specifically the *remove_all_stripe* method, which relies on the median filter. Although other options for faster ring removal methods are available, parameter tweaking may be required for individual slices or datasets within the same experiment, which is impractical. The advantage of the *remove_all_stripe* method is that the same set of parameters can be applied to the entire volume and different datasets.

Automating the workflow

In practice, we often need to reconstruct not just one but hundreds or even thousands of datasets per synchrotron beamtime. In these cases, manually processing each dataset would be time-consuming and impractical. Instead, we can leverage the power of Python to automate the workflow. The idea is to create a Python script that can iterate through a list of datasets and pass the path of each dataset to the full reconstruction script for processing, either one-by-one on a local workstation or in parallel on a cluster.

We need to modify the full-reconstruction script to accept the file path as a command-line argument. This will allow us to pass the file path to the script dynamically from our automation script. There are several ways of doing this:

- Using the *sys* module:

  Modify the top of the full reconstruction script:

```
#  Script to perform full reconstruction, named full_reconstruction.py
import sys
import time
import timeit
import numpy as np
import algotom.io.loadersaver as losa
import algotom.util.utility as util
import algotom.prep.correction as corr
import algotom.prep.calculation as calc
import algotom.prep.removal as remo
import algotom.prep.filtering as filt
import algotom.rec.reconstruction as rec


file_path = sys.argv[1]  #  sys.argv[0] is the name of this script.
output_base = sys.argv[2]
# To pass arguments to this script, run:
# python full_reconstruction.py arg1 arg2

print("Load file: {}".format(file_path))
#  Script body ...
```

  Then use the automation script as follows:

```
#  Script to call the full reconstruction script
import glob
import subprocess

python_interpreter = "C:/Users/nvo/Miniconda3/envs/algotom/python"
python_script = "full_reconstruction.py" #  At the same location of this␣
```

(continues on next page)

(continued from previous page)

```
↪script. Otherwise,
                                        #  providing the full path to full_
↪reconstruction.py

input_folder = "E:/datasets/"
output_base = "E:/full_reconstruction/"
# Get a list of hdf files in the input folder.
list_file = glob.glob(input_folder + "/*hdf")

for file in list_file:
    script = python_interpreter + " " + python_script + " " + file.replace(
↪"\\", "/") + " " + output_base
    subprocess.call(script, shell=True)
```

- Using the *argparse* module:

  Modify the full reconstruction script as below:

```
#  Script to perform full reconstruction, named full_reconstruction.py
import argparse
import time
import timeit
import numpy as np
import algotom.io.loadersaver as losa
import algotom.util.utility as util
import algotom.prep.correction as corr
import algotom.prep.calculation as calc
import algotom.prep.removal as remo
import algotom.prep.filtering as filt
import algotom.rec.reconstruction as rec


parser = argparse.ArgumentParser(description="Perform full reconstruction")
parser.add_argument("-i", dest="file_path", help="Path to input file",␣
↪type=str, required=True)
parser.add_argument("-o", dest="output", help="Output folder", type=str,␣
↪required=True)
args = parser.parse_args()
# To pass arguments to this script, run:
# python full_reconstruction.py -i file_path -o output

file_path = args.file_path
output_base = args.output

print("Load file: {}".format(file_path))
#  Script body ...
```

  Then just slightly modify the automation script:

```
#  Script to call the full reconstruction script
import glob
import subprocess
```

(continues on next page)

(continued from previous page)

```
python_interpreter = "C:/Users/nvo/Miniconda3/envs/algotom/python"
python_script = "full_reconstruction.py" #  At the same location of this␣
↪script. Otherwise,
                                         #  providing the full path to full_
↪reconstruction.py

input_folder = "E:/datasets/"
output_base = "E:/full_reconstruction/"
# Get a list of hdf files in the input folder.
list_file = glob.glob(input_folder + "/*hdf")

for file in list_file:
    script = python_interpreter + " " + python_script + " -i " + file.
↪replace("\\", "/") + " -o " + output_base
    subprocess.call(script, shell=True)
```

The instructions above are for running the reconstruction on a local machine (WinOS). However, if users have access to a cluster system (LinuxOS), they can take advantage of its resources to process multiple datasets in parallel using an embarrassingly parallel approach. The procedure of how to run reconstruction process on a cluster is as follows:

- Install Python packages. Although a cluster may already have a standard Python environment with a set of pre-installed packages, it may not include the package users need. In this case, users can create their own Python environment. There are several ways to create a new Python environment, but one popular method is to use *conda*. Conda is a package management system that makes it easy to create, manage environments and packages. One of the advantages of *conda* is that it includes many popular Python packages, and it also includes *pip*, which allows users to install packages only available on PyPI.org. If *conda* is not installed on the cluster system, users can follow instructions here to install it, then installing Python packages as shown here.
- Insert the full-path to the Python interpreter of the created environment at the top of python scripts:

```
#!/path/to/python/environment/bin/python

#  Script to perform full reconstruction, named full_reconstruction.py
import sys
# ...
```

  then making the file executable by run the following command in a Linux terminal:

```
chmod +x <filename>
```

- Write a bash script to submit jobs to the cluster scheduler. The bash script can be embed inside a Python script to make it easy to customize the workflow. The following example demonstrates how to do that for a SLURM cluster scheduler (for Univa Grid Engine scheduler, refer the example here):

```
#!/path/to/python/environment/bin/python

import os
import glob
import subprocess

python_script = "full_reconstruction.py"
use_gpu = True
input_folder = "/facility/beamline/data/year/proposals/visit/raw_data/"
# Get a list of nxs files in the input folder.
```

(continues on next page)

(continued from previous page)

```
list_file = glob.glob(input_folder + "/*nxs")
# Specify where to save the processed data
output_base = "/facility/beamline/data/year/proposals/visit/processing/
↪reconstruction"
# Specify the folder for cluster output-file and error-file.
cluster_dir = "/facility/beamline/data/year/proposals/visit/processing/cluster_
↪output/"

# Define a method to create a folder for saving output message from the cluster.
def make_folder(folder_path):
    file_base = os.path.dirname(folder_path)
    if not os.path.exists(file_base):
        try:
            os.makedirs(file_base)
        except FileExistsError:
            pass
        except OSError:
            raise ValueError("Can't create the folder: {}".format(file_base))

sbatch_script_cpu = """#!/bin/bash

#SBATCH --job-name=demo_workflow
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 16
#SBATCH --nodes=1
#SBATCH --mem=16G
#SBATCH --qos=normal
#SBATCH --time=60:00

srun -o {0}/output_%j.txt -e {0}/error_%j.txt ./{1} {2} {3}
"""

sbatch_script_gpu = """#!/bin/bash

#SBATCH --job-name=demo_workflow
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 16
#SBATCH --nodes=1
#SBATCH --mem=16G
#SBATCH --gres=gpu:1
#SBATCH --qos=normal
#SBATCH --time=60:00

srun -o {0}/output_%j.txt -e {0}/error_%j.txt ./{1} {2} {3}
"""

for file_path in list_file:
    file_name = os.path.basename(file_path)
    name = file_name.replace(".nxs", "")
    output_folder = output_base + "/" + file_name + "/"
    print("Submit to process the raw-data file : {}...".format(file_name))
    cluster_output = cluster_dir + "/" + name + "/"
```

(continues on next page)

(continued from previous page)

```
    make_folder(cluster_output)
    if use_gpu:
        sbatch_script = sbatch_script_gpu.format(cluster_output, python_script,
                                                 file_path, output_folder)
    else:
        sbatch_script = sbatch_script_cpu.format(cluster_output, python_script,
                                                 file_path, output_folder)
    # Call sbatch and pass the sbatch script contents as input
    process = subprocess.Popen(['sbatch'], stdin=subprocess.PIPE, stdout=subprocess.
↪PIPE, stderr=subprocess.PIPE)
    stdout, stderr = process.communicate(input=sbatch_script.encode())

    # Print the output and error messages
    print(stdout.decode())
    print(stderr.decode())
print("***********************************************")
print("             !!!!!! Done !!!!!!                ")
print("***********************************************")
```

- To run the script, make it executable and log in to a submitting job node. Users can modify the workflow above by reconstructing multiple datasets at once, such as 10 datasets in one batch, and waiting for them to finish before submitting another batch. This approach ensures fair use of cluster resources among multiple users.

Downsampling, rescaling, and reslicing reconstructed volume

Reconstructed volume is in 32-bit tif or hdf format. In the above example, the size of the volume is 2150 x 2560 x 2560 pixels, which corresponds to ~50 GB of data. To enable post-analysis on software for volume visualization and analysis; e.g. Avizo, Amira, DragonFly, Drishti, Paraview, 3D Slicer, ...; it is often necessary to apply data reduction techniques such as cropping, downsampling, or rescaling. Algotom provides convenient functions for these tasks, which can be applied to a folder of tif slices or a hdf/nxs file.

```
import timeit
import algotom.io.loadersaver as losa
import algotom.post.postprocessing as post

output_base = "E:/output/data_reduction/"

# Rescale the volume to 16-bit data including cropping.
# Input is tif, output is tif
tif_folder = "E:/full_reconstruction/recon_001"
output0 = output_base + "/rescaling/"
folder_name = losa.make_folder_name(output0)  # To avoid overwriting
output = output0 + "/" + folder_name + "/"
t_start = timeit.default_timer()
post.rescale_dataset(tif_folder, output, nbit=16, minmax=None, skip=None,
                     crop=(100, 100, 200, 200, 200, 200))

# # Input is hdf, output is tif
# file_path = "E:/full_reconstruction/recon_002/recon_data.hdf"
# key_path = "entry/data"
# post.rescale_dataset(file_path, output, key_path=key_path, nbit=16,␣
```

(continues on next page)

(continued from previous page)

```
↪minmax=None,
#                       skip=None, crop=(100, 100, 200, 200, 200, 200))
t_stop = timeit.default_timer()
print("Done rescaling! Time cost {}".format(t_stop - t_start))


# Downsample the volume by 2 x 2 x 2 with cropping and rescaling to 8-bit.
output0 = output_base + "/downsampling/"
folder_name = losa.make_folder_name(output0)  # To avoid overwriting
output = output0 + "/" + folder_name + "/"
t_start = timeit.default_timer()
post.downsample_dataset(tif_folder, output, (2, 2, 2), method='mean',
                        rescaling=True, nbit=8, minmax=None, skip=None,
                        crop=(100, 100, 200, 200, 200, 200))
t_stop = timeit.default_timer()
print("Done downsampling! Time cost {}".format(t_stop - t_start))
```

Reslicing the reconstructed volume is another important post-processing tool, especially for limited-angle tomography. While some software such as ImageJ or Avizo offer this function, they require loading the entire volume into memory, making it impossible to use on computers with limited RAM. Starting from version 1.3, Algotom provides a reslicing function that uses the hdf format as the back-end, eliminating the need for high memory usage. Additionally, options for cropping, rotating, and rescaling the volume are also included.

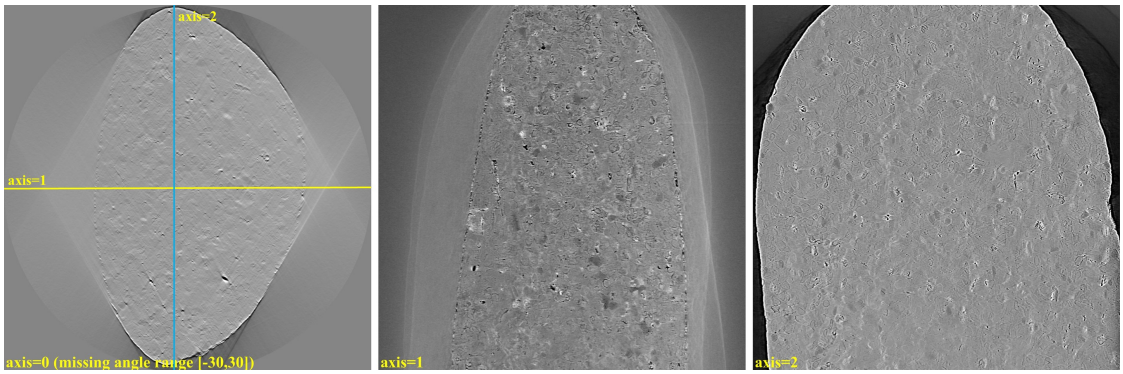
```
import timeit
import algotom.io.loadersaver as losa
import algotom.post.postprocessing as post

output_base = "E:/output/reslicing"

# Reslice the volume along axis 1, including rotating, cropping, and rescaling␣
↪to 8-bit data.
# Input is tif, output is tif
tif_folder = "E:/full_reconstruction/recon_001"
folder_name = losa.make_folder_name(output_base)  # To avoid overwriting
output = output_base + "/" + folder_name + "/"
t_start = timeit.default_timer()

post.reslice_dataset(tif_folder, output, rescaling=True, rotate=10.0,
                     nbit=8, axis=1, crop=(100, 100, 200, 200, 200, 200),
                     chunk=60, show_progress=True, ncore=None)

# # Input is hdf, output is tif. It's possible to slice a hdf volume directly
# # along axis 2 but it will be extremely slow. Better use the Algotom␣
↪function as below.
#
# file_path = "E:/full_reconstruction/recon_002/recon_data.hdf"
# key_path = "entry/data"
# post.reslice_dataset(file_path, output, key_path=key_path, rescaling=True,
#                      rotate=0.0, nbit=16, axis=2, crop=(100, 100, 200, 200,␣
↪200, 200),
#                      chunk=60, show_progress=True, ncore=None)
```

(continues on next page)

(continued from previous page)

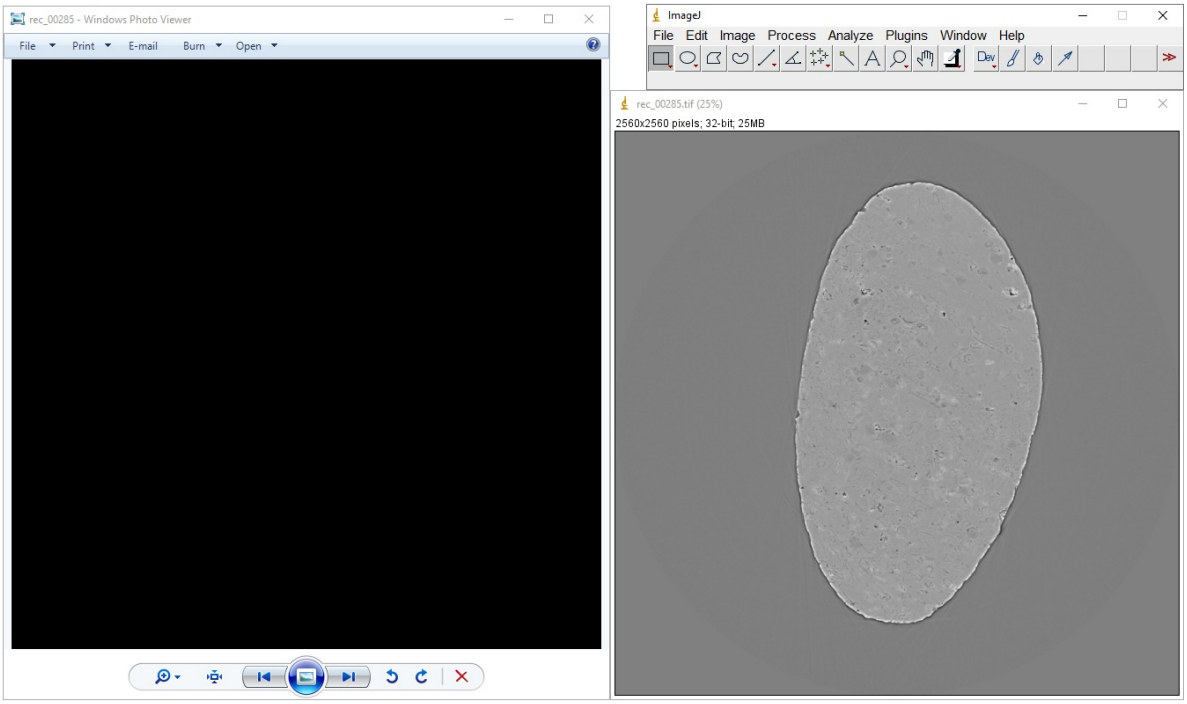
```
t_stop = timeit.default_timer()
print("Done reslicing! Time cost {}".format(t_stop - t_start))
```

As shown below, reslicing along the direction perpendicular to the missing wedge can produce high-quality images suitable for post-analysis.



Common mistakes and useful tips

1) We may see black images when using viewer software that does not support 32-bit tif images. Users need to use ImageJ or Fiji to view 32-bit tif reconstructed slices or flat-field-corrected images.



2) Black reconstructed slice is returned due to the zero division problem. Reconstruction methods in Algotom apply the logarithm function to a sinogram by default, based on Beer-Lambert's law. However, this can result in NaN values if there are zeros or negative values in the sinogram. Zeros or negative values may comes from phase-retrieved images or the *flat-field correction process* using projection images which may have the following:

   - Time stamp at one of the image corner.
   - Beam size is smaller than the field of view.

- Low signal-to-noise ratio.

To address these issues, there are several ways:

- Disable the logarithm function by setting *apply_log* to *False* in a reconstruction method if the input is a non-absorption-contrast image.
- Crop the images to exclude problematic regions.
- Not using dark-field image for low SNR data.
- Replace zeros and negative values in the sinogram as below

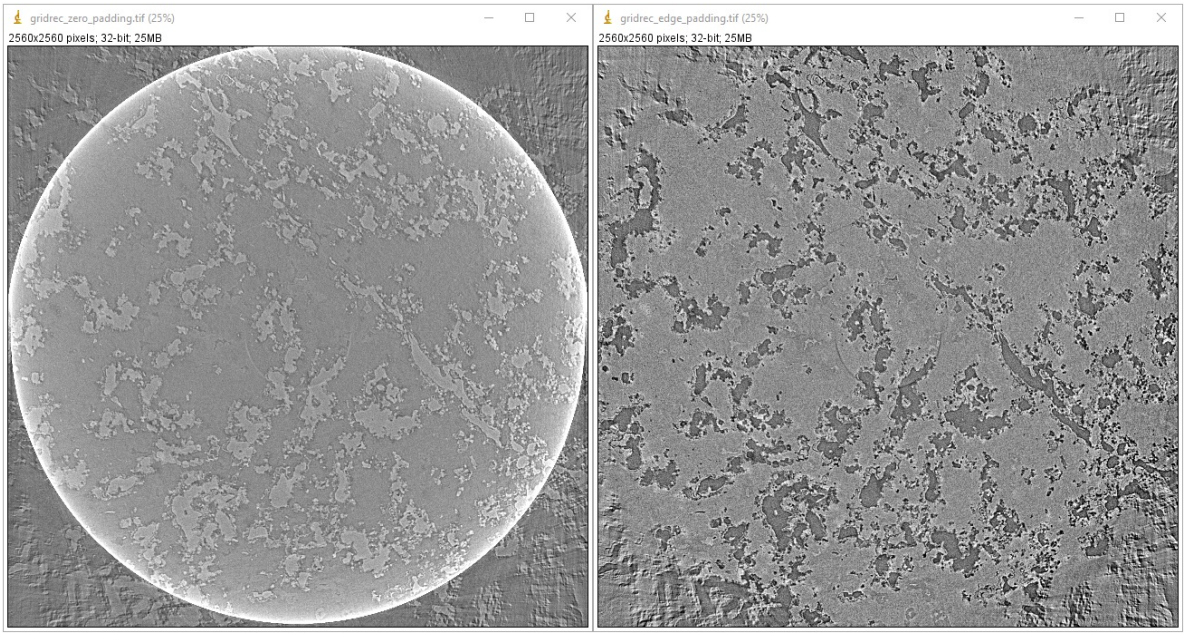
```
import numpy as np
nmean = np.mean(sinogram)
sinogram[sinogram<=0.0] = nmean
```

Algotom provides a convenient method for flat-field correction; with the options to correct zero division, not use dark-field image, or include other preprocessing methods.

3) Users may apply methods on the wrong space or slice data along incorrect axis. As shown in Fig. 1.4.10, it is assumed that the sample is upright, and therefore the rotation axis is parallel to the columns of the projection image. In 3D data, axis 0 represents the projection space; axis 1 represents the sinogram space and the reconstruction space. It is important to ensure that methods are applied correctly to the appropriate space and that data is sliced along the correct axis. Sometimes the rotation axis of a tomography system may be parallel to the rows of the projection image. In such cases, users need to rotate the projection image or adjust the slicing direction to obtain the sinogram image.

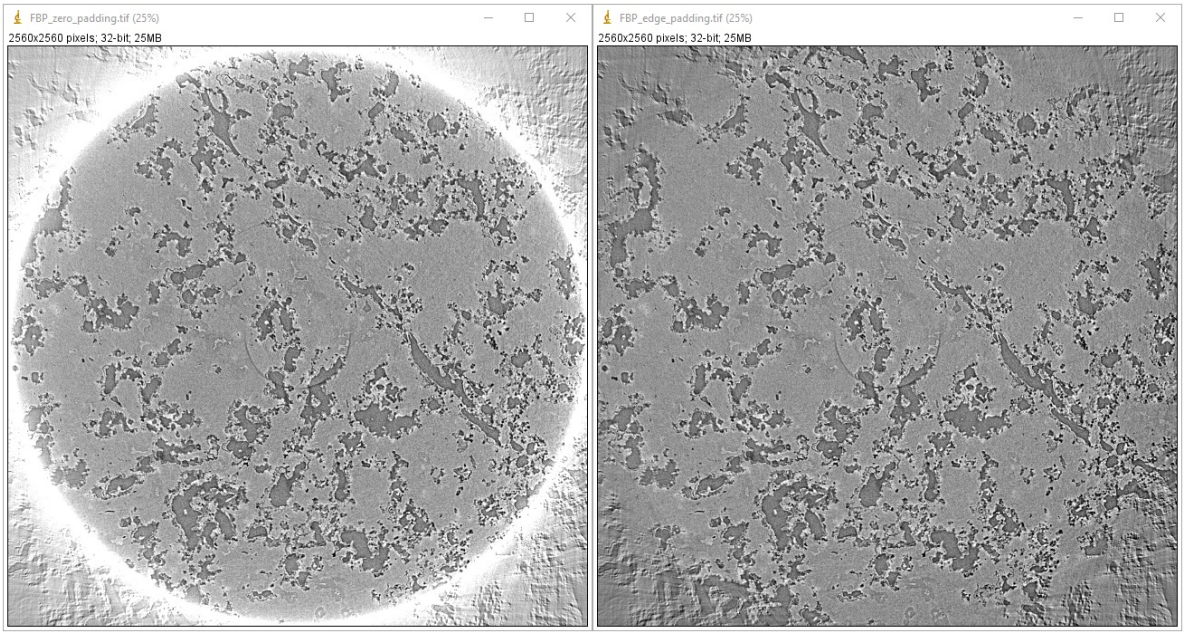
4) Cupping artifacts or outermost bright/dark ring artifacts can occur when padding is not used or wrong type of padding is used for Fourier-based reconstruction methods. This problem has a significant impact on post-analysis, particularly image segmentation, but very easy to fix simply by applying a proper padding such as 'edge', 'reflect', or 'symmetric'. In Algotom, 'edge' padding is enabled by default for FFT-based methods, but in other software this function may not be enabled by default or zero-padding is used. The following image demonstrates the difference between using zero padding and edge padding for the *gridrec* method.

```
import tomopy
import algotom.io.loadersaver as losa
import algotom.prep.calculation as calc
import algotom.rec.reconstruction as rec

center = calc.find_center_vo(sinogram)
# Algotom wrapper provides edge-padding.
rec_img1 = rec.gridrec_reconstruction(sinogram, center, ratio=None)
# Tomopy applies zero-padding by default.
rec_img2 = tomopy.recon(np.expand_dims(sinogram, 1),
                        np.deg2rad(np.linspace(0, 180.0, sinogram.shape[0])),
                        center=center, algorithm="gridrec")

losa.save_image(output_base + "/gridrec_edge_padding.tif", rec_img1)
losa.save_image(output_base + "/gridrec_zero_padding.tif", rec_img2[0])
```



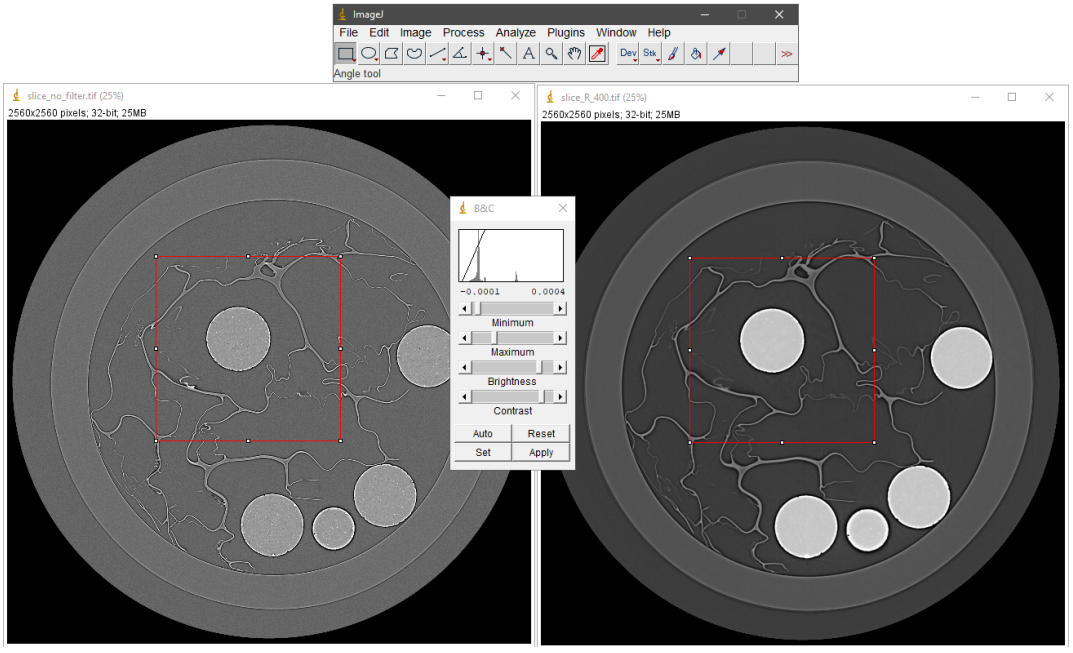
and demonstration for the FBP method:

```
import algotom.io.loadersaver as losa
import algotom.prep.calculation as calc
import algotom.rec.reconstruction as rec

center = calc.find_center_vo(sinogram)
# Using built-in FBP method in Algotom with edge padding.
rec_img1 = rec.fbp_reconstruction(sinogram, center, ratio=None)
# Using FBP through Astra Toolbox. Astra applies zero-padding behind the scene.
# The Algotom wrapper provides edge-padding in addition to Astra's zero-padding.
# However, the artifacts caused by the zero-padding can still persist, as it
# disrupts the intensities at the boundaries, which is problematic for
# Fourier-based methods.
rec_img2 = rec.astra_reconstruction(sinogram, center, ratio=None, method="FBP_CUDA",
↪ pad=0)

losa.save_image(output_base + "/FBP_edge_padding.tif", rec_img1)
losa.save_image(output_base + "/FBP_zero_padding.tif", rec_img2)
```

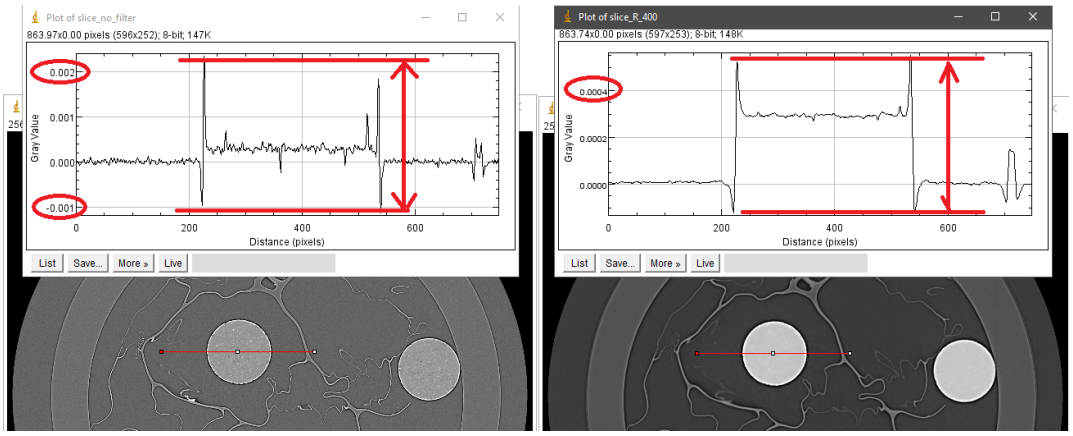


5) Users may not be aware of autoscaling implemented by image viewer software. Image viewers often apply autoscaling to account for differences in intensity range between different image types, such as 32-bit, 16-bit or 8-bit. However, this can lead to the displayed image having a contrast that does not accurately reflect the true contrast of the original image. The following shows examples of using the ImageJ software.

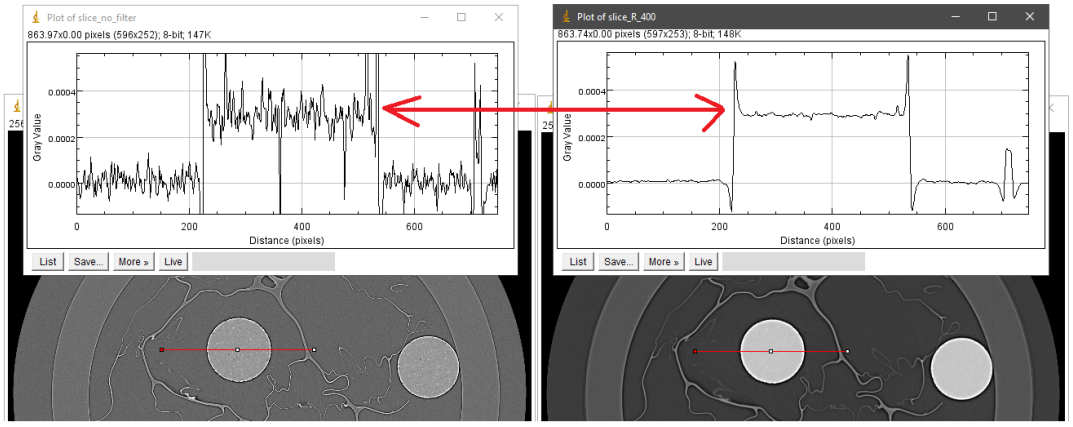
   Commonly, users may select a ROI and adjust the contrast of the image by autoscaling as shown below. An autoscaling method works by normalizing the whole image based on the local minimum gray-scale and local maximum gray-scale of the ROI. As can be seen, the left-side image is more noisy and has a higher dynamic range of intensities (distance between the maximum intensity and minimum intensity) compared to the right-side image. When the auto-scaling is applied, the contrast of the right-side image is improved because it has lower dynamic range.



   The following images shows the intensity profiles along the red lines in each image where the whole dynamic range of intensities are used to plot.



The following images show the intensity profiles along the red lines in each image where the dynamic range of intensities is set to be the same in both images. As can be seen, the gray-scale values of an Aluminum sphere are the same. Note that the intensities at the interfaces are strongly fluctuating due to the coherent effect of the X-ray source.



The above demonstration actually shows images reconstructed without and with Paganin filter (R=400), which was used to explain the common misconception that the resulting Paganin filter image is a phase-contrast image. From a mathematical point of view, Paganin's formula is a low-pass filter in Fourier space, with R as a tuning parameter that controls the strength of this filter. As a low-pass filter, it reduces noise and the dynamic range of an image, which can help enhance the contrast between low-contrast features. However, this can sometimes be confused with the phase effect, leading to the common misconception that the resulting image is a phase-contrast image.

6) Overlapping parallelization should be avoided as it can degrade performance. Many functions in Algotom are set to use multi-core by default. If users would like to write a wrapper on top to perform parallel work, such as processing multiple datasets, making sure that the *ncore* option in Algotom API is set to 1.

7) There are different ways of applying pre-processing methods to multiple-sinograms as shown below.

   Using with flat-field correction method:

```
import algotom.util.utility as util
import algotom.prep.correction as corr
import algotom.prep.removal as remo
import algotom.prep.filtering as filt

opt1 = {"method": "remove_zinger", "para1": 0.08, "para2": 1}
```

(continues on next page)

(continued from previous page)

```
opt2 = {"method": "remove_all_stripe", "para1": 3.0, "para2": 51, "para3":␣
↪17}
opt3 = {"method": "fresnel_filter", "para1": 200, "para2": 1}
sinograms = corr.flat_field_correction(proj_obj[:, 20:40, :], flat_field[20:
↪40, :], dark_field[20:40, :],
                                       option1=opt1, option2=opt2,␣
↪option3=opt3)
```

Applying methods one-by-one:

```
sinograms = corr.flat_field_correction(proj_obj[:, 20:40, :], flat_field[20:
↪40, :], dark_field[20:40, :])
sino_pro = []
for i in range(sinograms.shape[1]):
    sino_tmp = remo.remove_zinger(sinograms[:, i, :], 0.08, 1)
    sino_tmp = remo.remove_all_stripe(sino_tmp, 3.0, 51, 17)
    sino_tmp = filt.fresnel_filter(sino_tmp, 200, 1)
    sino_pro.append(sino_tmp)
# Convert results which is a Python list to a Numpy array and
# make sure axis 1 is corresponding to sinogram.
sinograms = np.moveaxis(np.asarray(sino_pro), 0, 1)
```

Applying methods in parallel manually:

```
import multiprocessing as mp
from joblib import Parallel, delayed

ncore = mp.cpu_count() - 1
sinograms = corr.flat_field_correction(proj_obj[:, 20:40, :], flat_field[20:
↪40, :], dark_field[20:40, :])
num_sino = sinograms.shape[1]

output_tmp = Parallel(n_jobs=ncore, prefer="threads")(delayed(
    remo.remove_zinger)(sinograms[:, j, :], 0.08, 1) for j in range(num_
↪sino))
sinograms = np.moveaxis(np.asarray(output_tmp), 0, 1)

output_tmp = Parallel(n_jobs=ncore, prefer="threads")(delayed(
    remo.remove_all_stripe)(sinograms[:, j, :], 3.0, 51, 21) for j in
                                                        range(num_sino))
sinograms = np.moveaxis(np.asarray(output_tmp), 0, 1)

output_tmp = Parallel(n_jobs=ncore, prefer="threads")(delayed(
    filt.fresnel_filter)(sinograms[:, j, :], 200, 1) for j in range(num_
↪sino))
sinograms = np.moveaxis(np.asarray(output_tmp), 0, 1)
```

Applying methods in parallel using Algotom API:

```
sinograms = corr.flat_field_correction(proj_obj[:, 20:40, :], flat_field[20:
↪40, :], dark_field[20:40, :])
sinograms = util.apply_method_to_multiple_sinograms(sinograms, "remove_
↪zinger", [0.08, 1],
```

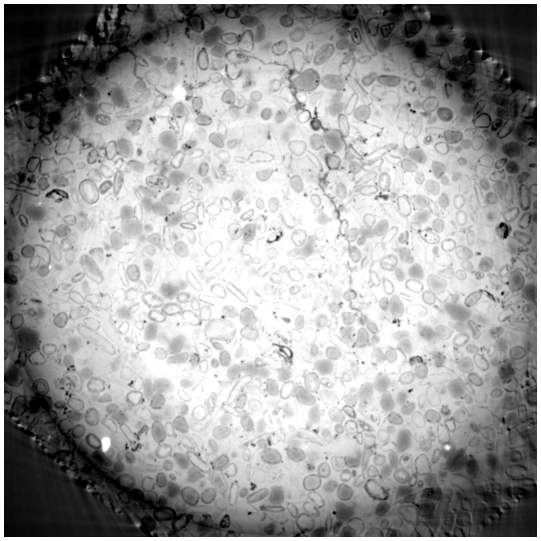
(continues on next page)

(continued from previous page)

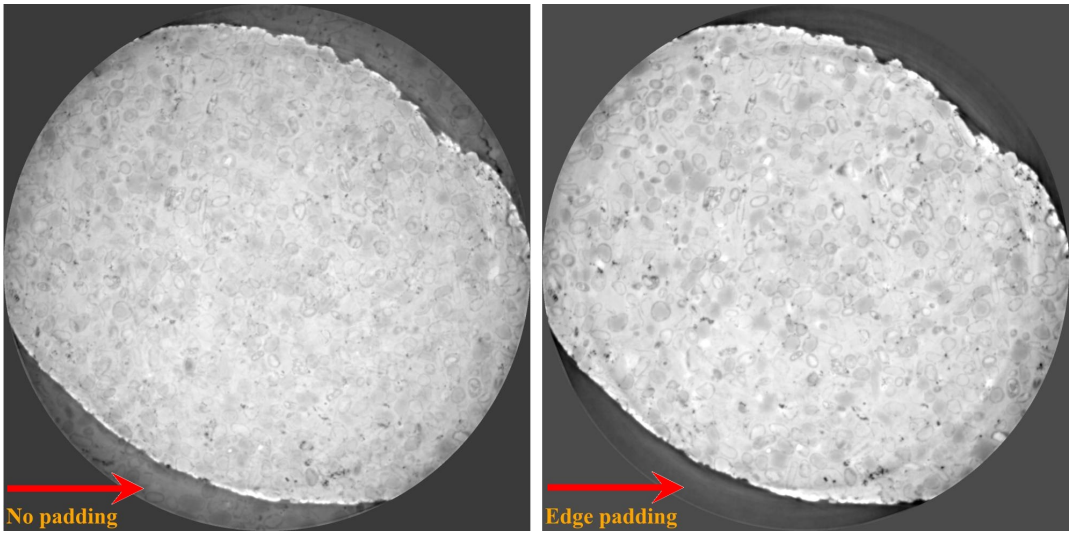
```
                                                    ncore=None, prefer=
↪"threads")
sinograms = util.apply_method_to_multiple_sinograms(sinograms, "remove_all_
↪stripe", [3.0, 51, 17],
                                                    ncore=None, prefer=
↪"threads")
sinograms = util.apply_method_to_multiple_sinograms(sinograms, "fresnel_
↪filter", [200, 1],
                                                    ncore=None, prefer=
↪"threads")
```

Starting from version 1.3, Algotom's reconstruction methods support batch processing of multiple sinograms at once. It is important to note that the axis of the reconstructed slices is 1, which is similar to the axis used for extracting sinograms.

8) Padding must be used for any Fourier-based image processing method, not just reconstruction as demonstrated in tip 5, to reduce/remove side-effect artifacts. Without padding, well-used Fourier-based filters, such as Paganin filter or Fresnel filter, applied on projection images can produce barrel-shaped intensity profiles in reconstructed images



or ghost features in the top and bottom slices caused by cross-shaped artifacts in the frequency domain due to spectral leakage.



9) In some cases, a tomography system may not be well-aligned, resulting in a rotation axis that is not perpendicular to the rows of projection images. The angle of misalignment can be very small and difficult to detect or calculate using projection images alone. A more accurate method involves extracting sinograms at the top, middle, and bottom of the tomographic data (or more, to improve the fitting result later), calculating the center of rotation, and then applying a linear fit to the results to obtain the tilt angle of the rotation axis.

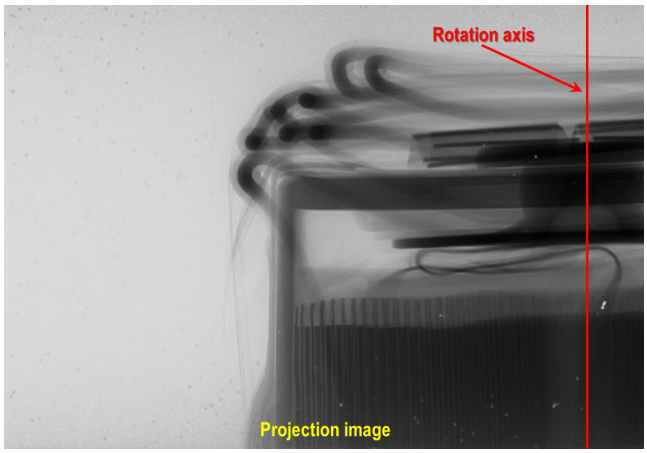
```
import numpy as np
import algotom.io.loadersaver as losa
import algotom.prep.correction as corr
import algotom.prep.calculation as calc
import algotom.rec.reconstruction as rec

file_path = "E:/Tomo_data/scan_68067.hdf"
output_base = "E:/output/tilted_projection/"
proj_path = "entry/projections"  # Refer section 1.2.1 to know how to get
                                 # path to a dataset in a hdf file.
flat_path = "entry/flats"
dark_path = "entry/darks"

# Load data, average flat and dark images
proj_obj = losa.load_hdf(file_path, proj_path)  # hdf object
(depth, height, width) = proj_obj.shape
flat_field = np.mean(np.asarray(losa.load_hdf(file_path, flat_path)),␣
↪axis=0)
dark_field = np.mean(np.asarray(losa.load_hdf(file_path, dark_path)),␣
↪axis=0)

# Find center at different height for calculating the tilt angle
slice_and_center = []
for i in range(10, height-10, height // 2 - 11):
    print("Find center at slice {}".format(i))
    sinogram = corr.flat_field_correction(proj_obj[:, i,:], flat_field[i],␣
↪dark_field[i])
    center = calc.find_center_vo(sinogram)
    print("Center is {}".format(center))
    slice_and_center.append([i, center])
```

(continues on next page)

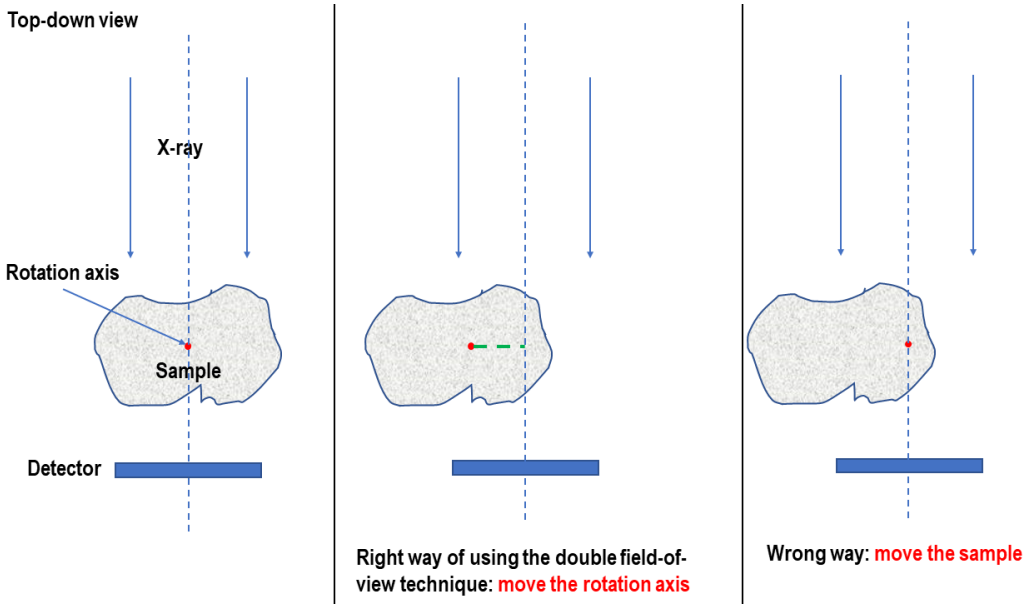
(continued from previous page)

```
slice_and_center = np.asarray(slice_and_center)

# Find the tilt angle using linear fit.
# Note that the sign of the tilt angle need to be changed if the projection
# images are flipped left-right or up-down by some detectors.
tilt_angle = -np.rad2deg(np.arctan(
    np.polyfit(slice_and_center[:, 0], slice_and_center[:, 1], 1)[0]))
print("Tilt angle: {} (degree)".format(np.deg2rad(tilt_angle)))

# Given tilted angle we can extract a single sinogram for reconstruction:
idx = height // 2
sino_tilted = corr.generate_tilted_sinogram(proj_obj, idx, tilt_angle)
flat_line = corr.generate_tilted_profile_line(flat_field, idx, tilt_angle)
dark_line = corr.generate_tilted_profile_line(dark_field, idx, tilt_angle)
sino_tilted = corr.flat_field_correction(sino_tilted, flat_line, dark_line)
center = calc.find_center_vo(sino_tilted)
rec_img = rec.fbp_reconstruction(sino_tilted, center)
losa.save_image(output_base + "/recon.tif", rec_img)
# or for a chunk of sinogram:
start_idx = 20
stop_idx = 40
sinos_tilted = corr.generate_tilted_sinogram_chunk(proj_obj, start_idx,␣
↪stop_idx, tilt_angle)
flats_tilted = corr.generate_tilted_profile_chunk(flat_field, start_idx,␣
↪stop_idx, tilt_angle)
darks_tilted = corr.generate_tilted_profile_chunk(dark_field, start_idx,␣
↪stop_idx, tilt_angle)
sinos_tilted = corr.flat_field_correction(sinos_tilted, flats_tilted, darks_
↪tilted)
center = calc.find_center_vo(sinos_tilted[:, start_idx, :])
recs_img = rec.fbp_reconstruction(sinos_tilted, center)
for i in range(start_idx, stop_idx):
    name = ("0000" + str(i))[-5:]
    losa.save_image(output_base + "/recon/recon_" + name + ".tif", recs_
↪img[:, i-start_idx, :])
```

10) For increasing the field of view of the reconstructed image, the technique of 360-degree scan with offset rotation axis, also known as half-acquisition (though this can be a confusing name), is commonly used. However, it is important to note that the rotation axis should be shifted to the side of the field of view, not the sample itself. From the projection image, it can be confusing as both shifts give the same results.



but it's much easier to understand using the sketch below



Data analysis

After cleaning and reconstructing all slices, the next step is to analyze the data to answer scientific questions. There are a variety of tools and software available to users. For beginners, the following resources may be helpful:

- For learning about the quantitative information that X-ray tomography can provide, a good starting point is the paper "Quantitative X-ray tomography" by E. Maire and P.J. Withers. This resource can provide a comprehensive overview of the field and help you understand the potential applications and benefits of this technique.
- Tutorials on YouTube are one of the most effective ways to learn quickly:
  - Rigaku virtual workshop: talk 1, talk 2.
  - Microscopy Australia channel: example talk.
  - Cscsch channel: example talk.
  - Channel of Dr. Sreenivas Bhattiprolu: tutorial playlists.

Examples of how to use the package are under the example folder of Algotom. They cover most of use-cases which users can adapt to process their own data. Examples of how to process speckle-based phase-contrast tomography is at here.

Users can use Algotom to re-process some old data collected at synchrotron facilities suffering from:

- Various types of ring artifacts.
- Cupping artifacts (also known as beam hardening artifacts) which are caused by using: FFT-based reconstruction methods without proper padding; polychromatic X-ray sources; or low-dynamic-range detectors to record high-dynamic-range projection-images.

There are tools and *methods* users can use to customize their own algorithms:

- Methods to transform images between the polar coordinate system and the Cartesian coordinate system.
- Methods to separate stripe artifacts.
- Methods to transform an image between the reconstruction space and the sinogram space.

Tomographic data for testing or developing methods can be downloaded from Zenodo.org or TomoBank. Methods can also be tested using simulation data as demonstrated here.

1.5 Technical notes

1.5.1 Implementations of X-ray speckle-based phase-contrast tomography

From version 1.1, methods for speckle-based phase contrast imaging have been implemented into Algotom. This is the result of the collaboration between the author, Nghia Vo, and his collaborators: Hongchang Wang, Marie-Christine Zdora, Lingfei Hu, and Tunhe Zhou; who are experienced with developing and using speckle-based imaging methods. This technical note is to summarize the work done (for more detail, see [R22]).

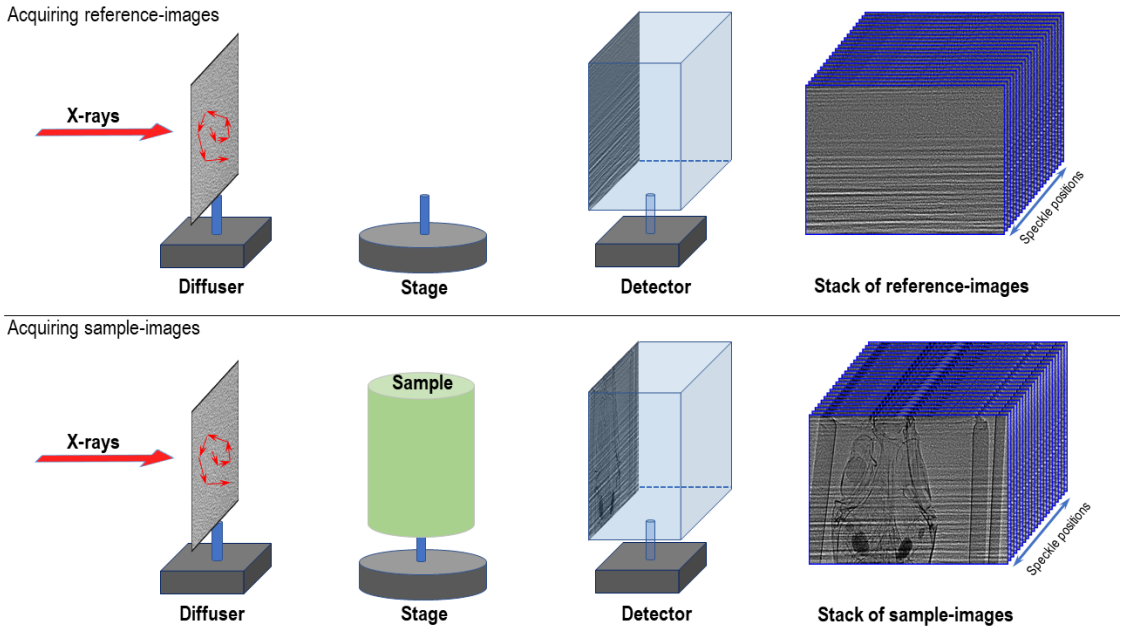
Introduction

When a sample interacts with a coherent X-ray beam, it will cause reduction in the intensity and change in the direction of the beam. The latter effect comes from the phase shift of the X-ray wave. Using the first effect to image samples in tomography, known as X-ray absorption-contrast tomography, is a widely used technique. However, using the second effect for imaging samples is much more challenging in practice. The resulting images, i.e. phase-shift images, can be used for tomography to visualize internal features of samples having small differences in densities at high contrast quality. This is the advantage of the technique, known as X-ray phase-contrast tomography (X-PCT), over conventional X-ray tomography.

To retrieve a phase-shift image, there are two basic approaches: measuring change in the direction of a beam then performing surface reconstruction; or matching a wave-propagation model to measured intensities. Speckle-based techniques use the first approach. They are very simple to use in terms of set-up, data acquisition, and data processing. The idea of the techniques is to measure the shift of each speckle-image pixel, caused by a sample, by comparing the images with and without the sample. Because the speckle-pattern size is larger than the pixel-size of a detector, to resolve the shift for each pixel we use a stack of speckle-images scanned at different positions, with and without sample, and analyze the images using a small window around each pixel.

Data acquisition

For producing a random speckle-pattern, a diffuser made of a sandpaper can be used. The purpose of a diffuser is to provide a reference-image used to detect local displacements caused by a sample. It is crucial to get a high-contrast reference-image with the average feature size of a few pixels, e.g. 5-9 pixels. A reference-image with the visibility, calculated as the ratio between the standard deviation value and the mean value of pixel intensities, above 10% is good enough for use. For high-energy X-ray sources, users can stack sandpapers together or using a box of material powder to improve the contrast of a speckle-image. Other practical considerations for setting up an experiment are as follows:



- The diffuser can be positioned before or after a sample depending on experiment conditions. For example, in a parallel-beam system with a highly spatial-coherent source the diffuser can be placed closer to the source than the sample to make use of the edge-enhancement effect which helps to improve the contrast of the speckle-image.
- The sample-detector distance should be chosen as a compromise between increasing the displacement effect and reducing the edge-enhancement effect caused by highly spatial-coherent sources.
- There are different ways of shifting a diffuser and acquiring a tomogram at each position. However, using a spiral path has been proven to be practical and efficient [R25]. The distance between two speckle positions should be larger than the analysis window, e.g. 5-11 pixels, to ensure that each speckle-pattern in the analysis window is completely different. This improves the robustness of methods measuring pixel shifts. Using this acquisition scheme, 20 positions of a diffuser is enough to retrieve a high-quality phase-shift image. However, for tomography systems with fluctuating sources, higher number of positions, e.g. 30-50, is needed.
- Due to mechanical error, moving a sample in and out of the field-of-view repeatedly for each diffuser position can cause small shifts between the same projections of different tomograms. This problem has a significant impact to the quality of processed data and is difficult to correct. To avoid it, the best scanning approach is to scan all positions of a diffuser first, then collect tomograms of a sample at each diffuser position. This approach may result in small displacements between the same speckle positions due to mechanical error. However, it is correctable by image alignment using a small area of empty space within the sample image.

Data processing

Finding pixel shifts

The core idea of the technique is to find the shift of each pixel of a speckle-image caused by a sample. This is done by: selecting a small window (5-11 pixels) around each pixel of the sample-stack image; sliding this window around a slightly larger window (margin ~10 pixels) taken from the reference-stack image and calculating the cost function [R25] or the correlation coefficient [R2] between two windows at each position. The resulting correlation-coefficient/cost-function map is used to locate the maximum/minimum point where sub-pixel accuracy can be achieved by using a differential approach or a polynomial fitting approach. The shift of a pixel is the distance from the maximum/minimum point to the center of the map. The procedure of finding the shift of each pixel is depicted in Fig. 1.5.1.

Performing 2D searching for every pixel of a 2k×2k image is computationally very expensive which is why using multicore-CPU and GPU for computing is crucially needed. An approximate approach to reduce the computational cost is to perform 1D search [R24] using middle slices in vertical and horizontal direction of image stacks, to find shifts in x and y-direction separately.

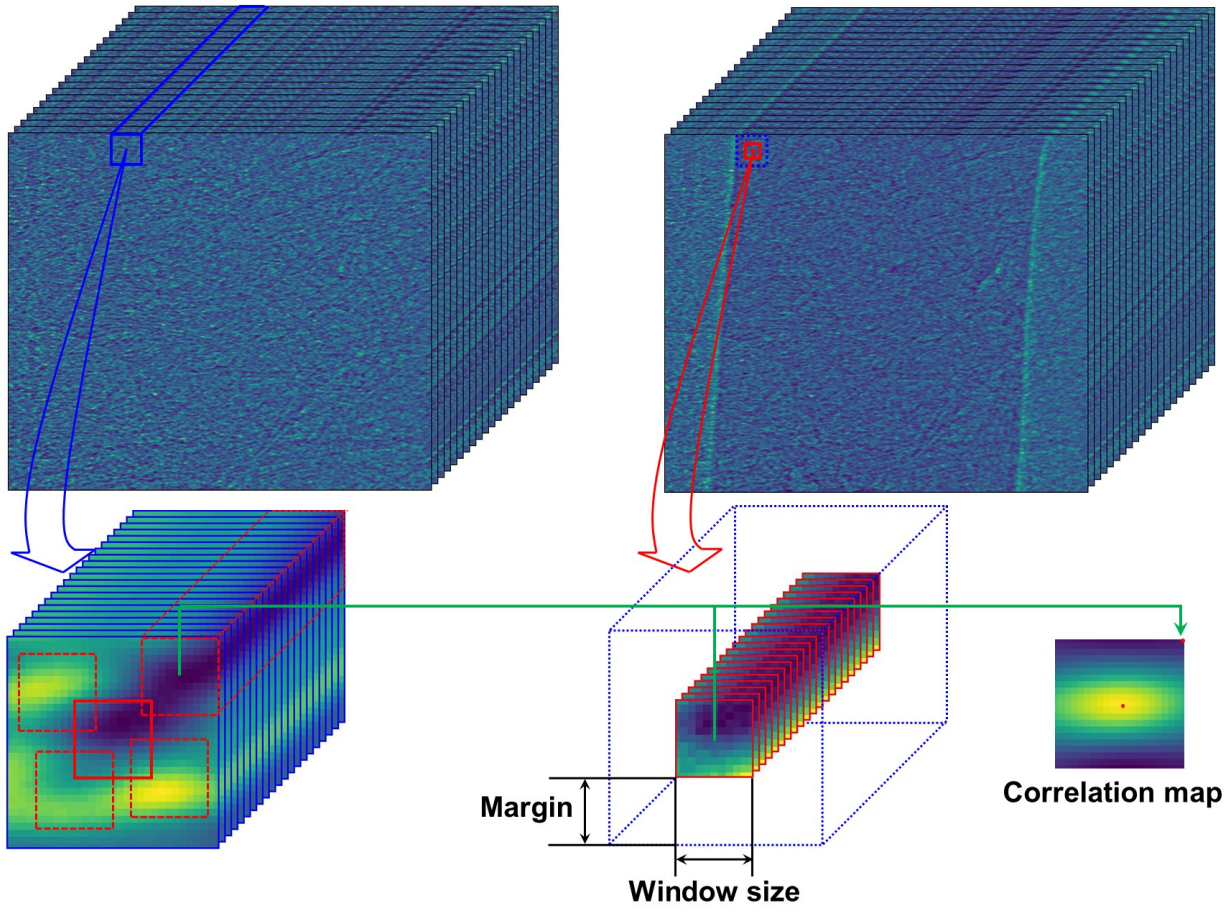


Fig. 1.5.1: Demonstration of how to find the shift of each speckle-pixel.

Surface reconstruction

The result of the previous step is separated into an x-shift image and a y-shift image, i.e. gradient images. A phase-shift image is then retrieved by applying a method of surface reconstruction, or normal integration (Fig. 1.5.2). There are many available options for implementing this step. However, Fourier-transform-based methods [R7, R15] are preferred over least-squares methods due to their low computational cost which is critical for tomography. The disadvantage of these Fourier methods is that the DC-component (average value of an image) is undefined resulting in the fluctuations in background between phase-retrieved images. This effect, however, can be corrected (Fig. 1.5.3) by using the double-wedge filter as described in [C1]

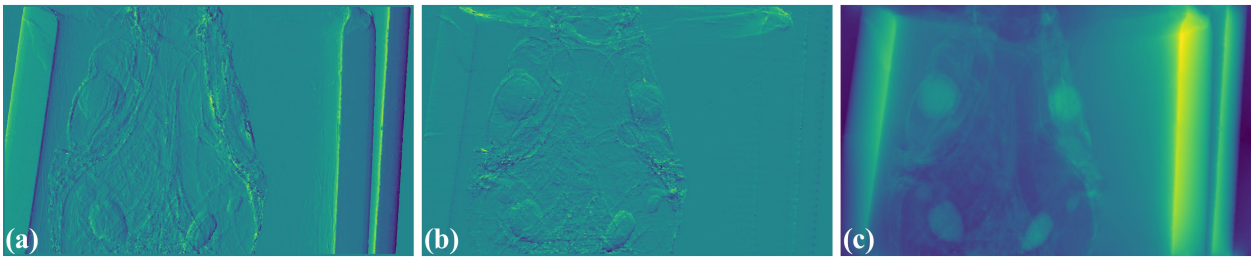


Fig. 1.5.2: Phase-shift image (c) is retrieved by normal integration using two gradient images: (a) x-direction; (b) y-direction.

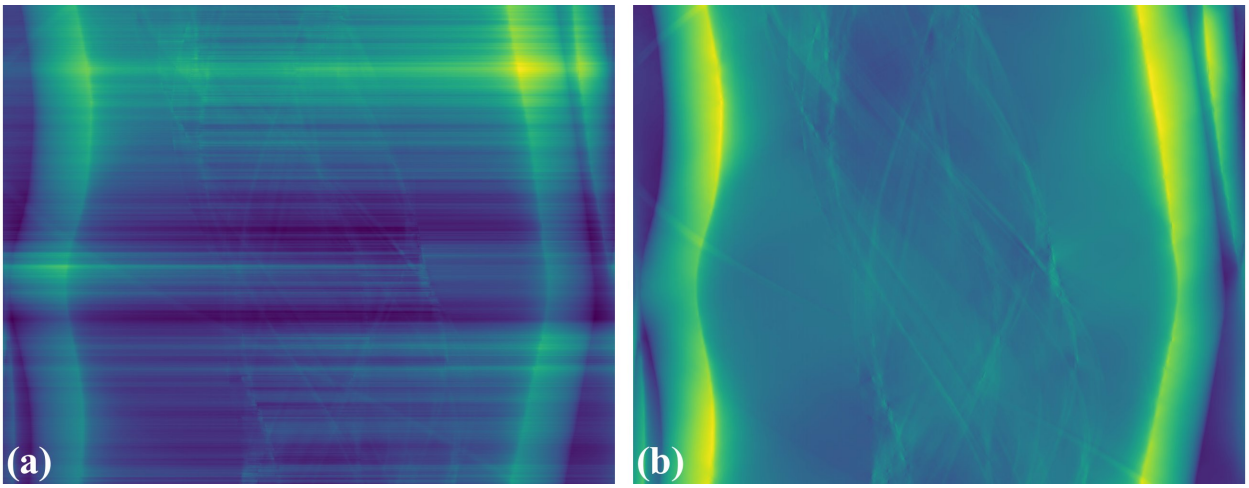


Fig. 1.5.3: (a) Fluctuation of grayscale values in a sinogram caused by the FT-based surface-reconstruction method. (b) Corrected image after using the double-wedge filter.

Extracting transmission and dark-field signals

Another interesting capability of the speckle-based technique is that transmission image (absorption-contrast image) and dark-field image (small-angle scattering signal, not to be confused with dark-noise of a camera) can be extracted from data together with the phase-shift image (Fig. 1.5.4). There are several ways to determine dark-signal image for correlation-based methods. For the cost-based approach [R25], dark-signal image is easily to be obtained as a part of the model equation.

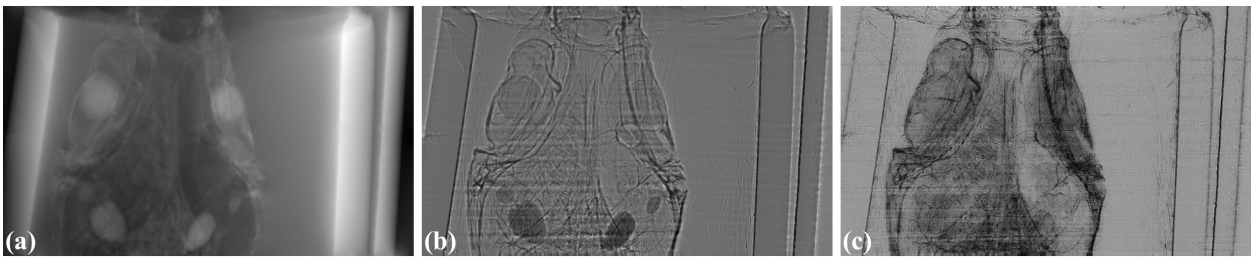


Fig. 1.5.4: All imaging signal retrieved by the speckle-based technique can be used for tomography. (a) Phase-shift image. (b) Transmission image. (c) Dark-field image.

Tomographic reconstruction

Above processing steps are repeated for every projection then the results are used for tomographic reconstruction as shown in Fig. 1.5.5

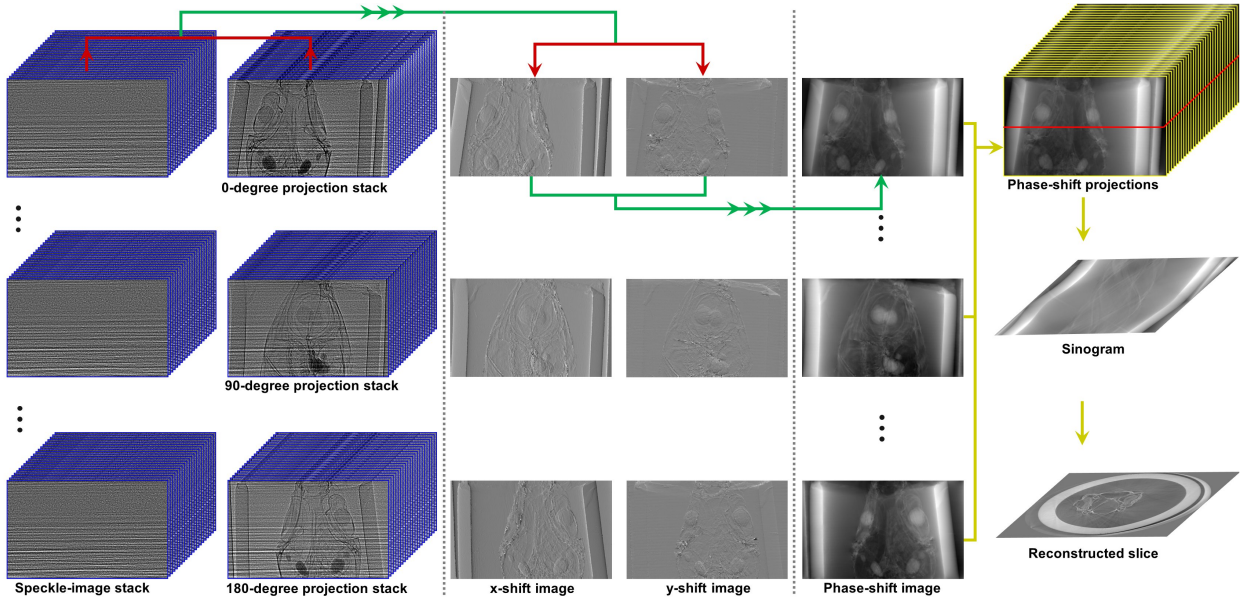


Fig. 1.5.5: Demonstration of the speckle-based phase-contrast tomography

Implementation

Design principles

Practical design-principles have been followed in the implementation:

- To ensure that the software can work across platforms and is easy-to-install; dependencies are minimized, and only well-maintained Python libraries are used.
- For high performance computing, making use of GPU, but ease of understanding and use; Numba library is used instead of Cupy or PyCuda.

- Methods are broken down into building blocks to be able to run on either small or large memory RAM/GPU. More importantly, this design allows users to customize methods or build data processing pipeline.

Top layer methods, API reference, for the software are as follows:

- Reading images from multiple datasets, in tif or hdf format, and stacking them.
- Finding local shifts between two images or two stacks of images.
- Performing surface reconstruction from gradient images.
- Retrieving phase-shift image given two stacks of images.
- Extracting transmission image and dark-field image.
- Aligning two images or two stacks of images.

Building blocks

A dedicated module in Algotom, named correlation, is a collection of methods as the building blocks for the top layer methods described in the previous section.

The first block is a method to generate correlation-coefficient map between two 2D/3D images (Fig. 1.5.1). This is the core method to find the shift between images. It works by sliding the second image over the reference image and calculating the correlation coefficient at each position. There are many formulas to calculate this coefficient. Here, we use Pearson's coefficient as it has been proven to be one of the most reliable metrics. The method includes low-level implementations for specific cases: 2D or 3D input, using CPU or GPU.

The second block is a method to locate the maximum/minimum point of a correlation-coefficient/cost-function map with sub-pixel accuracy where there are two approaches selected: either a differential approach [R6] or a polynomial fitting approach [R3]. At low-level are implementations to handle different cases: 1D or 2D input, using the differential method or fitting method.

The above blocks are for finding the shift of each pixel using a small window around it. This operation is applied to ~ 2k × 2k pixel. In practice, input data for retrieving a phase-shift image is two stacks of images; each stack is around 20 images (20 speckle-positions); each image has a size of 2k × 2k. Total shape of the input is 2 × 20 × 2k × 2k with the size of ~300MB (16-bit image). As can be seen, many strategies can be used to parallelize workload. Here we find that processing data using chunk-by-chunk of image-rows in one go and calculating the shifts row-by-row in parallel is the most efficient way in term of memory management, performance, and code readability.

As mentioned above, the next building block is a method for finding the shift of each pixel in a chunk of image-rows in parallel. The method includes low-level implementations for different cases: 1D or 2D search, 2D or 3D input, CPU or GPU computing. For GPU, to reduce the overhead of transferring data and compiling functions, the first two blocks are implemented at GPU-kernel level.

The top building block is a method for processing full-size images. It includes many options for processing at the lower-level blocks. The chunk-size option enables the method to run on either small memory or large memory of RAM or GPU. Other top-layer methods listed in the previous section are straightforward to implement either directly or by making use of the methods in the *correlation* module.

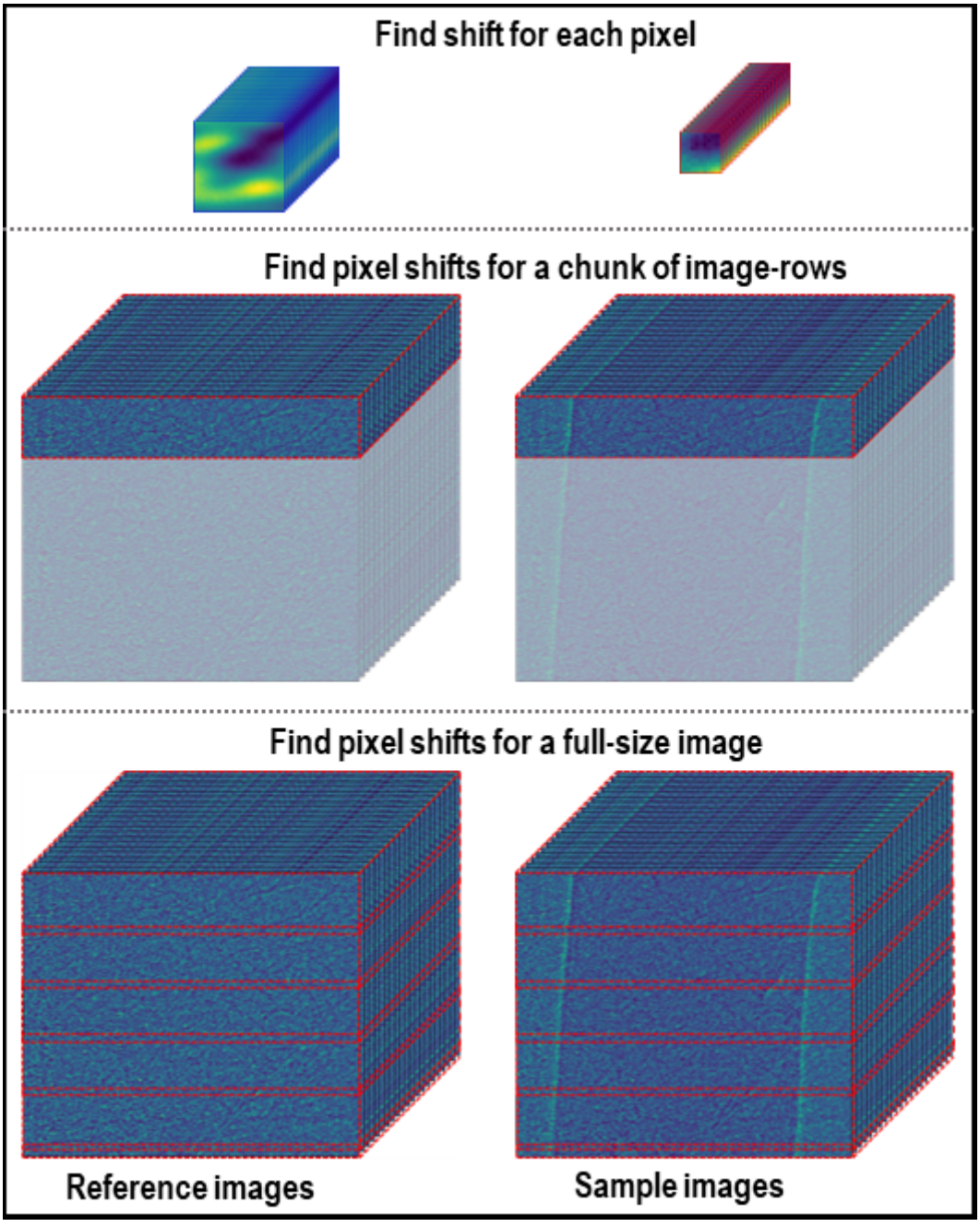


Fig. 1.5.6: Building blocks of the *correlation* module.

Demonstration

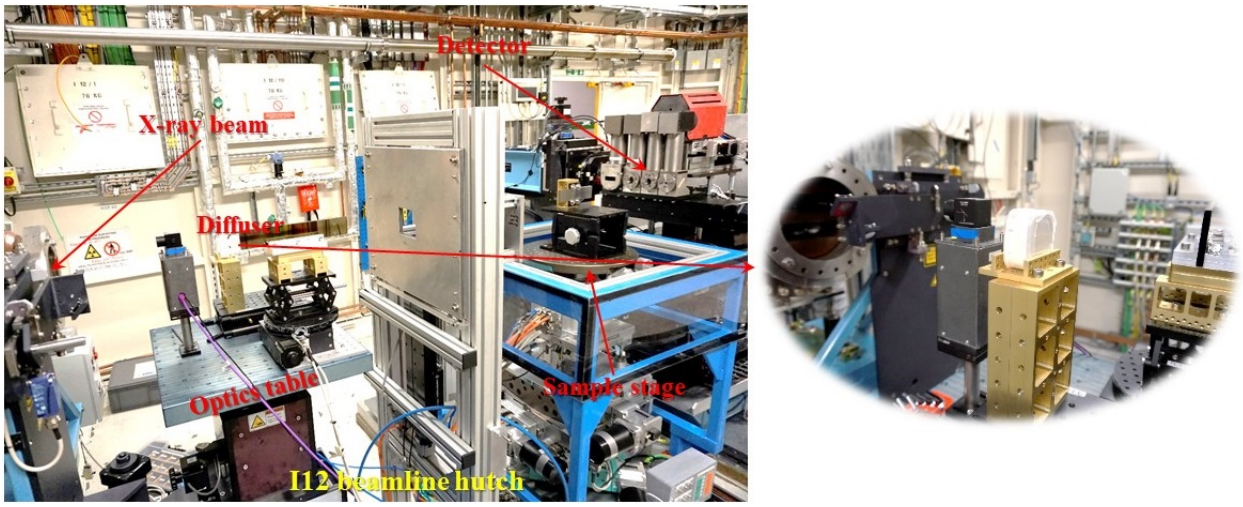


Fig. 1.5.7: Speckle-based tomographic experiment at beamline I12.

Data collected at beamline I12 at Diamond Light Source are used for demonstration. Details of how data were acquired are as follows:

- A box of fine sand was used as a speckle generator and can achieve a visibility of 13% at 53keV X-rays with the detector-sample distance of 2.2m. A detector with the pixel size of 7.9 µm was used. Image-size is 2560 and 2160 in height and width. The speckle-size is around 8 pixels. The intensity of the beam profile is very stable which is an important advantage of beamline I12. The sample is a picrite basaltic rock from Iceland.
- 20 speckle positions following a spiral path with the step of 30 times of the pixel size were used for scanning.
- Speckle images without the sample were acquired at all positions first. Then for each speckle position a tomographic scan of the sample, 1801 projections, was acquired. This strategy ensures that projections at the same angle are not shifted between speckle positions. Due to mechanical error, the diffuser positions were not the same between the first scan (without the sample) and the second scan (with the sample). This problem can be solved by image alignment using free-space areas in each image (Fig. 1.5.8).

The following presents how the data were processed:

- Reference-images for each position are loaded, averaged, normalized (flat-field corrected), aligned, and stacked.
- For each angle of tomographic datasets, projections at different speckle-positions are loaded, normalized, and stacked.
- Phase-shift image is retrieved from two previous image-stacks (Fig. 1.5.9) using a single function. Full options for choosing back-end methods, surface reconstruction methods, and searching parameters are at the API reference page.

Algotom implements three approaches: the correlation-based method using 1D [R24] and 2D [R2] search, and the cost-based approach [R25], known as the UMPA (Unified Modulated Pattern Analysis) method. A summary of computing time for retrieving a single phase-shift image using different options is shown in Fig. 1.5.10 where the window size is 7 and the margin is 10. As can be seen, there is a huge speed-up of computing time if using GPU.

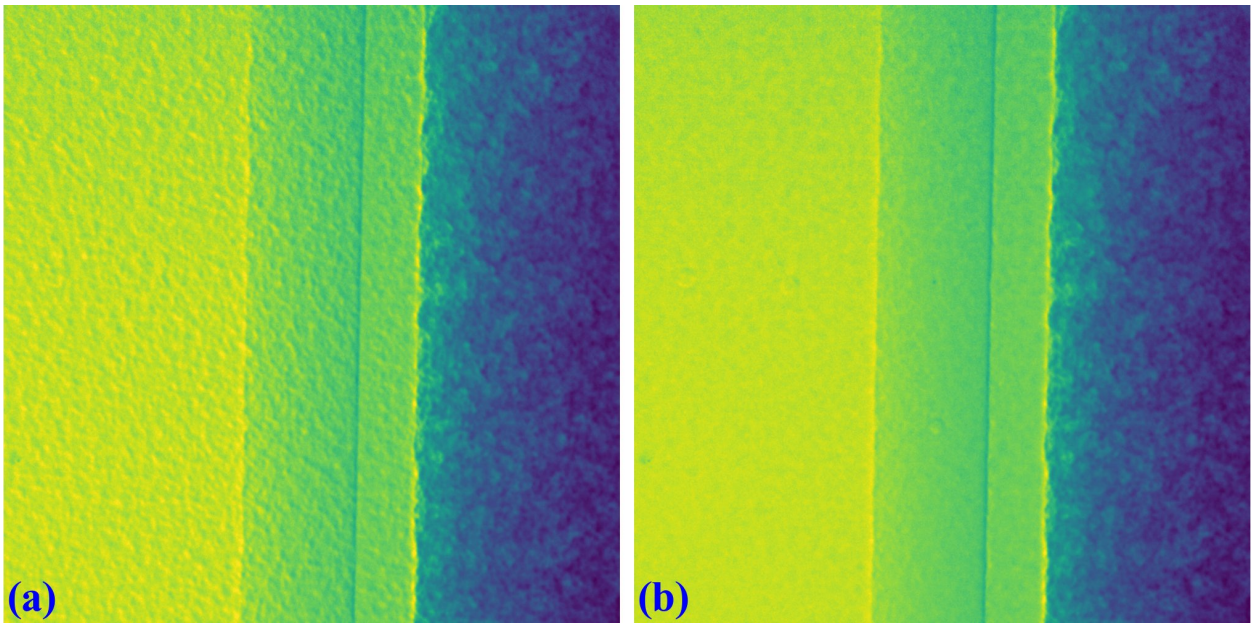


Fig. 1.5.8: Demonstration of the impact of image alignment. (a) Small area of an image which is the result of dividing between speckle-image with sample and without sample. (b) Same as (a) but after image alignment.

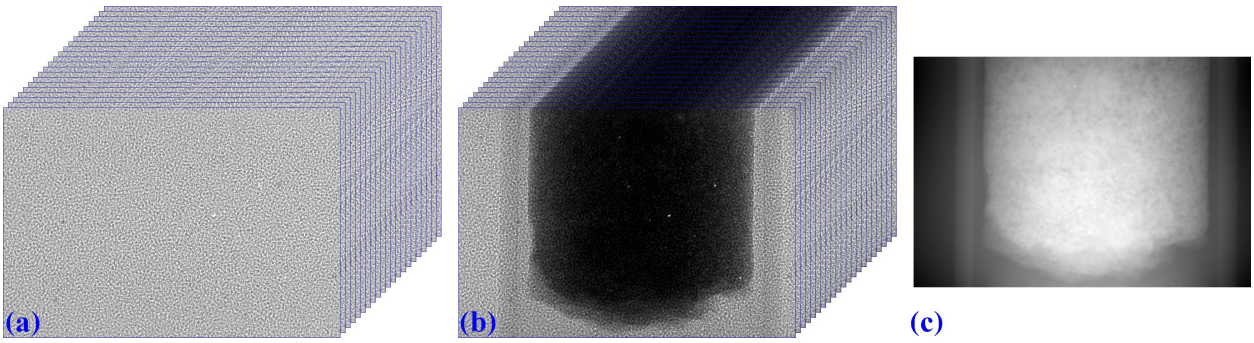


Fig. 1.5.9: Speckle-image stack (a). Sample-image stack (b). Phase-shift image (c) retrieved from (a) and (b).

Comparison of computing time (second) using different approaches

Method	Using the correlation coefficient		Using the cost function
Searching	1D	2D	2D
CPU (32 cores, Intel Xeon Gold 6152)	59	2152	2637
GPU (Nvidia Quadro GV100)	20	44	45

Fig. 1.5.10: Comparison of computing time using different approaches.

For tomographic reconstruction, phase retrieval is applied to all projections then the sinograms are generated for reconstructing slice-by-slice. This step can be manually parallelized for multiple-CPUs or multiple-GPUs to reduce computing time. In practice, users may want to tweak parameters and check the results before running full reconstruction. This can be done by performing phase retrieval on a small area of projection-images.

Fig. 1.5.11 shows reconstructed images in horizontal and vertical direction from the three approaches where ring artifact removal methods [R10, R19] and the FBP reconstruction method were used. There are several interesting findings from the results. Firstly, the 1D-search method gives less-sharp images than other methods but with better contrast and clearer features. There is not much different between the 2D-search method and the UMPA method out of the low-pass component. However, the main advantage of the UMPA approach over the others is that three modes of image can be retrieved at the same time as shown in Fig. 1.5.12. This figure is also a showcase for the speckle-based tomography technique where phase-shift images give better contrast than transmission-signal images (red arrows). The technique reveals interesting features of the sample which are mineral olivine. Because the olivine is a crystal it can enhance dark signal as shown in Fig. 1.5.12 (c,f). Making use of dark-signal images to gain deeper understanding of materials is a very promising application of the technique.

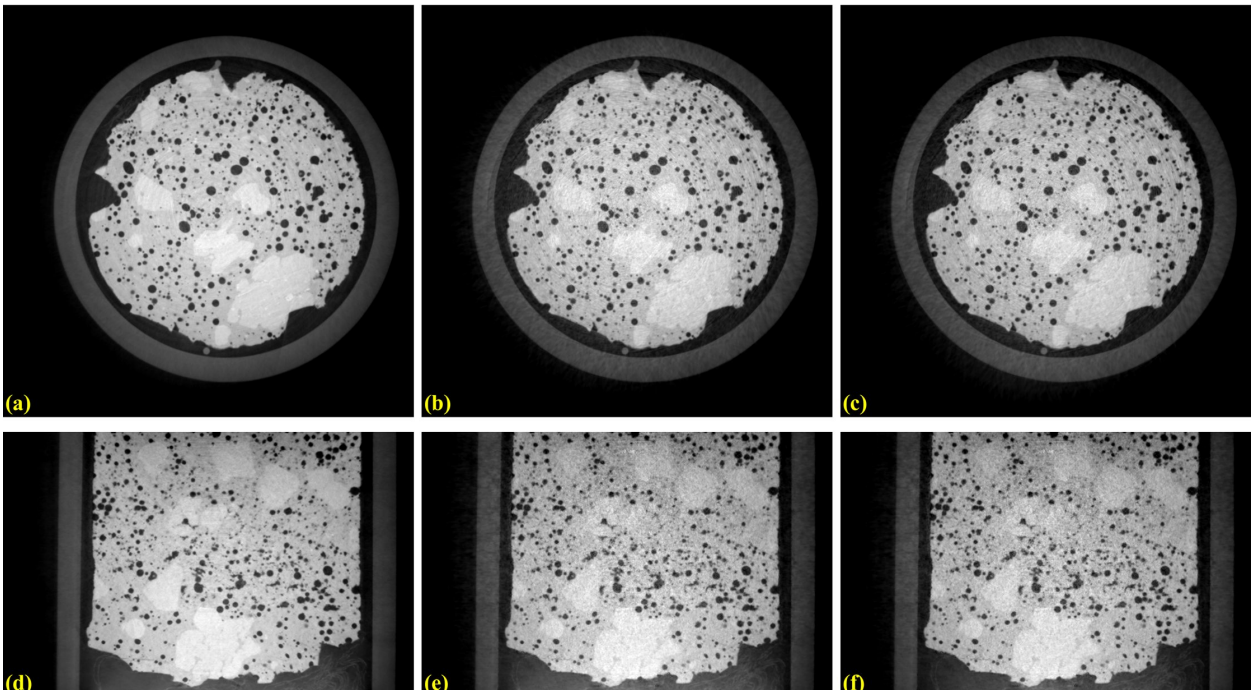


Fig. 1.5.11: Horizontal slice and vertical slice of reconstructed volumes from the 3 approaches: the 1D-search method (a,d); the 2D-search method (b,e); and UMPA (c,f).

Python codes used to process data for this report are at here. Detailed references can be found in [R22].

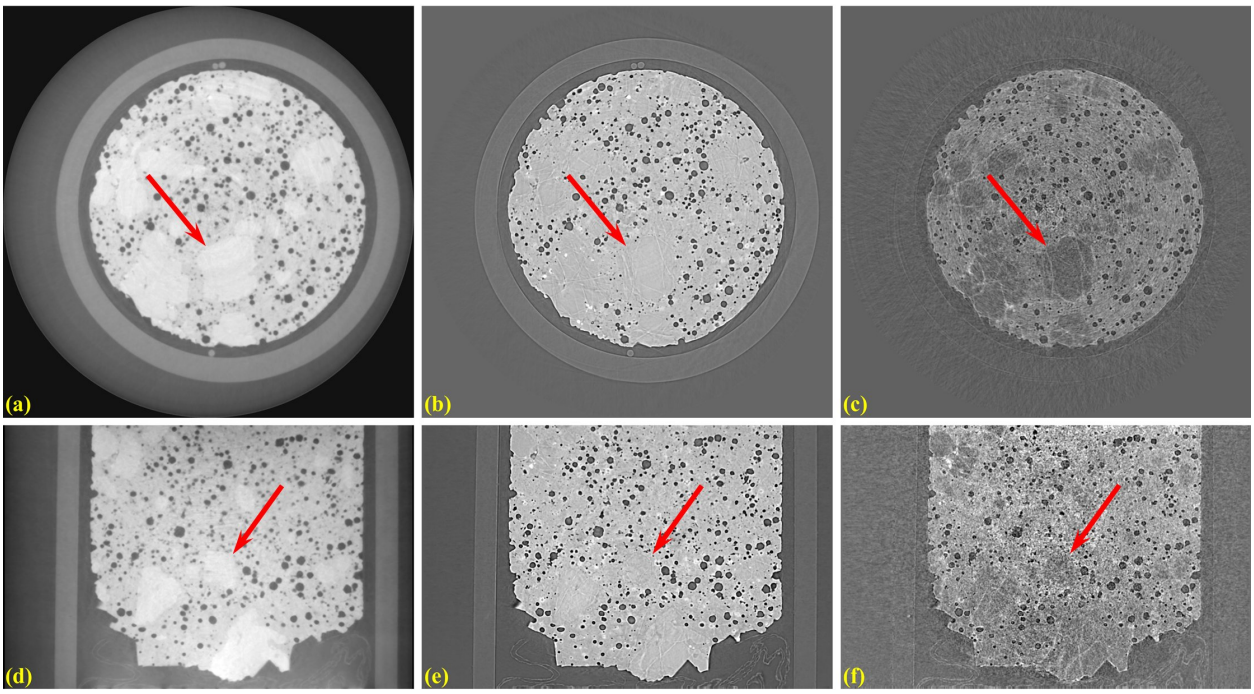


Fig. 1.5.12: Horizontal slice and vertical slice of reconstructed volumes from 3 imaging modes: phase-shift image (a,d); transmission image (b,e); and dark-signal image (c,f).

1.6 Update notes

- 13/05/2021:
  - Publish codes.
- 26/01/2022:
  - Add phase.py module.
  - Add phase-unwrapping methods.
- 20/06/2022:
  - Add correlation.py module.
  - Add methods for speckle-based phase-contrast tomography.
  - Add methods for image alignment.
  - Release version 1.1.
- 27/06/2022:
  - Publish https://algotom.github.io/
- 20/10/2022:
  - Publish implementation of the UMPA method.
- 24/10/2022:
  - Release version 1.2.
- 03/02/2023:

  - Add reslicing 3D-data method. Increase code coverage.

- 25/03/2023:
  - Add upsampling sinogram method.
  - Add method for finding the center of rotation (COR) using the entropy-based metric.
  - Add utility methods for visually finding the COR using: converted 360-degree sinograms and reconstructed slices.
  - Improve reconstruction methods to process multiple-sinograms.
- 30/03/2023:
  - Improve the performance of the reslicing method.
  - Release version 1.3.
- 19/11/2023:
  - Add methods for loading and saving multiple tiff images in parallel.
  - Release version 1.4.

1.7 API Reference

1.7.1 Input-output

algotom.io.converter

Module for converting data type:

- Convert a list of tif files to a hdf/nxs file.
- Extract tif images from a hdf/nxs file.

Functions:

convert_tif_to_hdf(input_path, output_path)	Convert a folder of tif files to a hdf/nxs file.
extract_tif_from_hdf(input_path, ...[, ...])	Extract tif images from a hdf/nxs file.

algotom.io.converter.**convert_tif_to_hdf**(*input_path*, *output_path*, *key_path='entry/data'*, *crop=(0, 0, 0, 0)*, *pattern=None*, ***options*)

Convert a folder of tif files to a hdf/nxs file.

Parameters

- **input_path** (*str*) – Folder path to the tif files.
- **output_path** (*str*) – Path to the hdf/nxs file.
- **key_path** (*str, optional*) – Key path to the dataset.
- **crop** (*tuple of int, optional*) – Crop the images from the edges, i.e. crop = (crop_top, crop_bottom, crop_left, crop_right).
- **pattern** (*str, optional*) – Used to find tif files with names matching the pattern.

- **options** (*dict, optional*) – Add metadata. E.g options={"entry/angles": angles, "entry/energy": 53}.

Returns

str – Path to the hdf/nxs file.

algotom.io.converter.**extract_tif_from_hdf**(*input_path*, *output_path*, *key_path*, *index=(0, -1, 1)*, *axis=0*, *crop=(0, 0, 0, 0)*, *prefix='img'*)

Extract tif images from a hdf/nxs file.

Parameters

- **input_path** (*str*) – Path to the hdf/nxs file.
- **output_path** (*str*) – Output folder.
- **key_path** (*str*) – Key path to the dataset in the hdf/nxs file.
- **index** (*tuple of int or int.*) – Indices of extracted images. A tuple corresponds to (start,stop,step).
- **axis** (*int*) – Axis which the images are extracted.
- **crop** (*tuple of int, optional*) – Crop the images from the edges, i.e. crop = (crop_top, crop_bottom, crop_left, crop_right).
- **prefix** (*str, optional*) – Prefix of names of tif files.

Returns

str – Folder path to the tif files.

algotom.io.loadersaver

Module for I/O tasks:

- Load data from an image file (tif, png, jpeg) or a hdf/nxs file.
- Get information from a hdf/nxs file.
- Search for datasets in a hdf/nxs file.
- Save a 2D array as a tif image or 2D, 3D array to a hdf/nxs file.
- Get file names, make file/folder name.
- Load distortion coefficients from a txt file.
- Get the tree view of a hdf/nxs file.
- Functions for loading stacks of images from multiple datasets, e.g. to be used by speckle-based phase contrast tomography.

Functions:

load_image(file_path)	Load data from an image.
get_hdf_information(file_path[, display])	Get information of datasets in a hdf/nxs file.
find_hdf_key(file_path, pattern[, display])	Find datasets matching the name-pattern in a hdf/nxs file.
load_hdf(file_path, key_path[, return_file_obj])	Load a hdf/nexus dataset as an object.
make_folder(file_path)	Create a folder for saving file if the folder does not exist.
make_file_name(file_path)	Create a new file name to avoid overwriting.
make_folder_name(folder_path[, name_prefix, ...])	Create a new folder name to avoid overwriting.
find_file(path)	Search file
save_image(file_path, mat[, overwrite])	Save a 2D array to an image.
open_hdf_stream(file_path, data_shape[, ...])	Write an array to a hdf/nxs file with options to add metadata.
load_distortion_coefficient(file_path)	Load distortion coefficients from a text file.
save_distortion_coefficient(file_path, ...)	Write distortion coefficients to a text file.
get_hdf_tree(file_path[, output, add_shape, ...])	Get the tree view of a hdf/nxs file.
get_reference_sample_stacks_dls(proj_idx, ...)	A method for multi-position speckle-based phase-contrast tomography to get two stacks of reference images (speckle images) and sample images (at the same rotation angle from each tomographic dataset).
get_reference_sample_stacks(proj_idx, ...[, ...])	Get two stacks of reference images (speckle images) and sample images (at the same rotation angle from each tomographic dataset).
get_tif_stack(file_base[, idx, crop, ...])	Load tif images to a stack.
get_image_stack(idx, paths[, data_key, ...])	To get multiple images with the same index from multiple datasets (tif format or hdf format).
load_image_multiple(list_path[, ncore, prefer])	Load list of images in parallel.
save_image_multiple(list_path, image_stack)	Save an 3D-array to a list of tif images in parallel.

algotom.io.loadersaver.**load_image**(*file_path*)

Load data from an image.

Parameters
file_path (*str*) – Path to the file.

Returns
array_like – 2D array.

algotom.io.loadersaver.**get_hdf_information**(*file_path*, *display=False*)

Get information of datasets in a hdf/nxs file.

Parameters

- **file_path** (*str*) – Path to the file.
- **display** (*bool*) – Print the results onto the screen if True.

Returns

- **list_key** (*str*) – Keys to the datasets.
- **list_shape** (*tuple of int*) – Shapes of the datasets.
- **list_type** (*str*) – Types of the datasets.

algotom.io.loadersaver.**find_hdf_key**(*file_path*, *pattern*, *display=False*)

Find datasets matching the name-pattern in a hdf/nxs file.

Parameters

- **file_path** (*str*) – Path to the file.
- **pattern** (*str*) – Pattern to find the full names of the datasets.
- **display** (*bool*) – Print the results onto the screen if True.

Returns

- **list_key** (*str*) – Keys to the datasets.
- **list_shape** (*tuple of int*) – Shapes of the datasets.
- **list_type** (*str*) – Types of the datasets.

algotom.io.loadersaver.**load_hdf**(*file_path*, *key_path*, *return_file_obj=False*)

Load a hdf/nexus dataset as an object.

Parameters

- **file_path** (*str*) – Path to the file.
- **key_path** (*str*) – Key path to the dataset.
- **return_file_obj** (*bool, optional*)

Returns

objects – hdf-dataset object, and file-object if return_file_obj is True.

algotom.io.loadersaver.**make_folder**(*file_path*)

Create a folder for saving file if the folder does not exist. This is a supplementary function for savers.

Parameters

file_path (*str*) – Path to a file.

algotom.io.loadersaver.**make_file_name**(*file_path*)

Create a new file name to avoid overwriting.

Parameters

file_path (*str*)

Returns

str – Updated file path.

algotom.io.loadersaver.**make_folder_name**(*folder_path*, *name_prefix='Output'*, *zero_prefix=5*)

Create a new folder name to avoid overwriting. E.g: Output_00001, Output_00002…

Parameters

- **folder_path** (*str*) – Path to the parent folder.
- **name_prefix** (*str*) – Name prefix
- **zero_prefix** (*int*) – Number of zeros to be added to file names.

Returns

str – Name of the folder.

algotom.io.loadersaver.**find_file**(*path*)

Search file

Parameters
path (*str*) – Path and pattern to find files.

Returns
str or list of str – List of files.

algotom.io.loadersaver.**save_image**(*file_path*, *mat*, *overwrite=True*)

Save a 2D array to an image.

Parameters

- **file_path** (*str*) – Path to the file.
- **mat** (*int or float*) – 2D array.
- **overwrite** (*bool*) – Overwrite an existing file if True.

Returns
str – Updated file path.

algotom.io.loadersaver.**open_hdf_stream**(*file_path*, *data_shape*, *key_path='entry/data'*, *data_type='float32'*, *overwrite=True*, ***options*)

Write an array to a hdf/nxs file with options to add metadata.

Parameters

- **file_path** (*str*) – Path to the file.
- **data_shape** (*tuple of int*) – Shape of the data.
- **key_path** (*str*) – Key path to the dataset.
- **data_type** (*str*) – Type of data.
- **overwrite** (*bool*) – Overwrite the existing file if True.
- **options** (*dict, optional*) – Add metadata. E.g options={"entry/angles": angles, "entry/energy": 53}.

Returns
object – hdf object.

algotom.io.loadersaver.**load_distortion_coefficient**(*file_path*)

Load distortion coefficients from a text file. The file must use the following format: x_center : float y_center : float factor0 : float factor1 : float ...

Parameters
file_path (*str*) – Path to the file

Returns
tuple of float and list – Tuple of (xcenter, ycenter, list_fact).

algotom.io.loadersaver.**save_distortion_coefficient**(*file_path*, *xcenter*, *ycenter*, *list_fact*, *overwrite=True*)

Write distortion coefficients to a text file.

Parameters

- **file_path** (*str*) – Path to the file.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*float*) – 1D array. Coefficients of the polynomial fit.

- **overwrite** (*bool*) – Overwrite an existing file if True.

Returns
str – Updated file path.

algotom.io.loadersaver.**get_hdf_tree**(*file_path*, *output=None*, *add_shape=True*, *display=True*)

Get the tree view of a hdf/nxs file.

Parameters

- **file_path** (*str*) – Path to the file.
- **output** (*str or None*) – Path to the output file in a text-format file (.txt, .md,...).
- **add_shape** (*bool*) – Including the shape of a dataset to the tree if True.
- **display** (*bool*) – Print the tree onto the screen if True.

Returns
list of string

algotom.io.loadersaver.**get_reference_sample_stacks_dls**(*proj_idx*, *list_path*, *data_key=None*, *image_key=None*, *crop=(0, 0, 0, 0)*, *flat_field=None*, *dark_field=None*, *num_use=None*, *fix_zero_div=True*)

A method for multi-position speckle-based phase-contrast tomography to get two stacks of reference images (speckle images) and sample images (at the same rotation angle from each tomographic dataset).

The method is specific to tomographic datasets acquired at Diamond Light Source (DLS) where projection-images, flat-field images, and dark-field images are in the same 3d array. There is a dataset named "image_key" inside a hdf/nxs file used to distinguish image types.

Parameters

- **proj_idx** (*int*) – Index of a projection-image in a tomographic dataset.
- **list_path** (*list of str*) – List of file paths (hdf/nxs format) to tomographic datasets.
- **data_key** (*str, optional*) – Key to images. Automatically find the key if None.
- **image_key** (*str, list, tuple, ndarray, optional*) – Key to 1d-array dataset for specifying image types. Automatically find the key if None. Can be used to pass the 1d-array manually.
- **crop** (*tuple of int, optional*) – Crop the images from the edges, i.e. crop = (crop_top, crop_bottom, crop_left, crop_right).
- **flat_field** (*ndarray, optional*) – 2D array or None. Used for flat-field correction if not None.
- **dark_field** (*ndarray, optional*) – 2D array or None. Used for dark-field correction if not None.
- **num_use** (*int, optional*) – Number of datasets used for stacking.
- **fix_zero_div** (*bool, optional*) – Correct zeros to avoid zero-division problem down the processing line.

Returns

- **ref_stack** (*ndarray*) – Return if reference-images found. 3D array.
- **sam_stack** (*ndarray*) – 3D array. A stack of sample-images.

algotom.io.loadersaver.**get_reference_sample_stacks**(*proj_idx*, *ref_path*, *sam_path*, *ref_key*, *sam_key*, *crop=(0, 0, 0, 0)*, *flat_field=None*, *dark_field=None*, *num_use=None*, *fix_zero_div=True*)

Get two stacks of reference images (speckle images) and sample images (at the same rotation angle from each tomographic dataset). A method for multi-position speckle-based phase-contrast tomography.

Parameters

- **proj_idx** (*int*) – Index of a projection-image in a tomographic dataset.
- **ref_path** (*list of str*) – List of file paths (hdf/nxs format) to reference-image datasets.
- **sam_path** (*list of str*) – List of file paths (hdf/nxs format) to tomographic datasets.
- **ref_key** (*str*) – Key to a reference-image dataset.
- **sam_key** (*str*) – Key to a projection-image dataset.
- **crop** (*tuple of int, optional*) – Crop the images from the edges, i.e. crop = (crop_top, crop_bottom, crop_left, crop_right).
- **flat_field** (*ndarray, optional*) – 2D array or None. Used for flat-field correction if not None.
- **dark_field** (*ndarray, optional*) – 2D array or None. Used for dark-field correction if not None.
- **num_use** (*int, optional*) – Number of datasets used for stacking.
- **fix_zero_div** (*bool, optional*) – Correct zeros to avoid zero-division problem down the processing line.

Returns

- **ref_stack** (*ndarray*) – 3D array. A stack of reference-images.
- **sam_stack** (*ndarray*) – 3D array. A stack of sample-images.

algotom.io.loadersaver.**get_tif_stack**(*file_base*, *idx=None*, *crop=(0, 0, 0, 0)*, *flat_field=None*, *dark_field=None*, *num_use=None*, *fix_zero_div=True*)

Load tif images to a stack. Supplementary method for 'get_image_stack'.

Parameters

- **file_base** (*str*) – Folder path to tif images.
- **idx** (*int or None*) – Load single or multiple images.
- **crop** (*tuple of int, optional*) – Crop the images from the edges, i.e. crop = (crop_top, crop_bottom, crop_left, crop_right).
- **flat_field** (*ndarray, optional*) – 2D array or None. Used for flat-field correction if not None.
- **dark_field** (*ndarray, optional*) – 2D array or None. Used for dark-field correction if not None.
- **num_use** (*int, optional*) – Number of images used for stacking.
- **fix_zero_div** (*bool, optional*) – Correct zeros to avoid zero-division problem down the processing line.

Returns

img_stack (*ndarray*) – 3D array. A stack of images.

algotom.io.loadersaver.**get_image_stack**(*idx*, *paths*, *data_key=None*, *average=False*, *crop=(0, 0, 0, 0)*, *flat_field=None*, *dark_field=None*, *num_use=None*, *fix_zero_div=True*)

To get multiple images with the same index from multiple datasets (tif format or hdf format). For tif images, if "paths" is a string (not a list) use idx=None to load all images. For getting a stack of images from a single hdf file, use the "load_hdf" method instead.

Parameters

- **idx** (*int or None*) – Index of an image in a dataset. Use None to load all images if only one dataset provided.
- **paths** (*list of str or str*) – List of hdf/nxs file-paths, list of folders of tif-images, or a folder of tif-images.
- **data_key** (*str*) – Requested if input is a hdf/nxs files.
- **average** (*bool, optional*) – Average images in a dataset if True.
- **crop** (*tuple of int, optional*) – Crop the images from the edges, i.e. crop = (crop_top, crop_bottom, crop_left, crop_right).
- **flat_field** (*ndarray, optional*) – 2D array or None. Used for flat-field correction if not None.
- **dark_field** (*ndarray, optional*) – 2D array or None. Used for dark-field correction if not None.
- **num_use** (*int, optional*) – Number of datasets used for stacking.
- **fix_zero_div** (*bool, optional*) – Correct zeros to avoid zero-division problem down the processing line.

Returns

img_stack (*ndarray*) – 3D array. A stack of images.

`algotom.io.loadersaver.`**`load_image_multiple`**`(`*list_path*, *ncore=None*, *prefer='threads'*`)`

Load list of images in parallel.

Parameters

- **list_path** (*str*) – List of file paths.
- **ncore** (*int or None*) – Number of cpu-cores. Automatically selected if None.
- **prefer** (*{"threads", "processes"}*) – Prefer backend for parallel processing.

Returns

array_like – 3D array.

`algotom.io.loadersaver.`**`save_image_multiple`**`(`*list_path*, *image_stack*, *axis=0*, *overwrite=True*, *ncore=None*, *prefer='threads'*, *start_idx=0*`)`

Save an 3D-array to a list of tif images in parallel.

Parameters

- **list_path** (*str*) – List of output paths or a folder path
- **image_stack** (*array_like*) – 3D array.
- **overwrite** (*bool*) – Overwrite an existing file if True.
- **ncore** (*int or None*) – Number of cpu-cores. Automatically selected if None.
- **prefer** (*{"threads", "processes"}*) – Prefer backend for parallel processing.
- **start_idx** (*int*) – Starting index of the output files if input is a folder.

1.7.2 Pre-processing

algotom.prep.calculation

Module of calculation methods in the preprocessing stage:

- Calculating the center-of-rotation (COR) using a 180-degree sinogram.
- Determining the overlap-side and overlap-area between images.
- Calculating the COR in a half-acquisition scan (360-degree scan with offset COR).
- Using the similar technique as above to calculate the COR in a 180-degree scan from two projections.
- Determining the relative translations between images using phase-correlation technique.
- Calculating the COR using phase-correlation technique.

Functions:

make_inverse_double_wedge_mask(height, ...)	Generate a double-wedge binary mask using Eq.
calculate_center_metric(center, sino_180, ...)	Calculate a metric of an estimated center-of-rotation.
coarse_search_cor(sino_180, start, stop[, ...])	Find the center-of-rotation (COR) using integer shifting.
fine_search_cor(sino_180, start, radius, step)	Find the center-of-rotation (COR) using sub-pixel shifting.
downsample_cor(image, dsp_fact0, dsp_fact1)	Downsample an image by averaging.
find_center_vo(sino_180[, start, stop, ...])	Find the center-of-rotation using the method described in Ref.
calculate_curvature(list_metric)	Calculate the curvature of a fitted curve going through the minimum value of a metric list.
correlation_metric(mat1, mat2)	Calculate the correlation metric.
search_overlap(mat1, mat2, win_width, side)	Calculate the correlation metrics between a rectangular region, defined by the window width, on the utmost left/right side of image 2 and the same size region in image 1 where the region is slided across image 1.
find_overlap(mat1, mat2, win_width[, side, ...])	Find the overlap area and overlap side between two images (Ref.
find_overlap_multiple(list_mat, win_width[, ...])	Find the overlap-areas and overlap-sides of a list of images where the overlap side referring to the previous image.
find_center_360(sino_360, win_width[, side, ...])	Find the center-of-rotation (COR) in a 360-degree scan with offset COR use the method presented in Ref.
complex_gradient(mat)	Return complex gradient of a 2D array.
find_shift_based_phase_correlation(mat1, mat2)	Find relative translation in x and y direction between images with haft-pixel accuracy (Ref.
find_center_based_phase_correlation(mat1, mat2)	Find the center-of-rotation (COR) using projection images at 0-degree and 180-degree.
find_center_projection(mat1, mat2[, flip, ...])	Find the center-of-rotation (COR) using projection images at 0-degree and 180-degree based on a method in Ref.
calculate_reconstructable_height(y_start, ...)	Calculate reconstructable height in a helical scan.
calculate_maximum_index(y_start, y_stop, ...)	Calculate the maximum index of a reconstructable slice in a helical scan.

algotom.prep.calculation.**make_inverse_double_wedge_mask**(*height*, *width*, *radius*, *hor_drop=None*, *ver_drop=None*)

Generate a double-wedge binary mask using Eq. (3) in Ref. [1]. Values outside the double-wedge region correspond to 1.0.

Parameters

- **height** (*int*) – Image height.
- **width** (*int*) – Image width.
- **radius** (*int*) – Radius of an object, in pixel unit.
- **hor_drop** (*int or None, optional*) – Number of rows (2 * hor_drop) around the middle of the mask with values set to zeros.
- **ver_drop** (*int or None, optional*) – Number of columns (2 * ver_drop) around the middle of the mask with values set to zeros.

Returns

array_like – 2D binary mask.

References

[1] : https://doi.org/10.1364/OE.22.019078

algotom.prep.calculation.**calculate_center_metric**(*center*, *sino_180*, *sino_flip*, *sino_comp*, *mask*)

Calculate a metric of an estimated center-of-rotation.

Parameters

- **center** (*float*) – Estimated center.
- **sino_180** (*array_like*) – 2D array. 180-degree sinogram.
- **sino_flip** (*array_like*) – 2D array. Flip the 180-degree sinogram in the left/right direction.
- **sino_comp** (*array_like*) – 2D array. Used to fill the gap left by image shifting.
- **mask** (*array_like*) – 2D array. Used to select coefficients in the double-wedge region.

Returns

float – Metric.

algotom.prep.calculation.**coarse_search_cor**(*sino_180*, *start*, *stop*, *ratio=0.5*, *denoise=True*, *ncore=None*, *hor_drop=None*, *ver_drop=None*)

Find the center-of-rotation (COR) using integer shifting.

Parameters

- **sino_180** (*array_like*) – 2D array. 180-degree sinogram.
- **start** (*int*) – Starting point for searching COR.
- **stop** (*int*) – Ending point for searching COR.
- **ratio** (*float*) – Ratio between a sample and the width of the sinogram.
- **denoise** (*bool, optional*) – Apply a smoothing filter.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.

- **hor_drop** (*int or None, optional*) – Refer the method of "make_inverse_double_wedge_mask"
- **ver_drop** (*int or None, optional*) – Refer the method of "make_inverse_double_wedge_mask"

Returns
float – Center of rotation.

algotom.prep.calculation.**fine_search_cor**(*sino_180*, *start*, *radius*, *step*, *ratio=0.5*, *denoise=True*, *ncore=None*, *hor_drop=None*, *ver_drop=None*)

Find the center-of-rotation (COR) using sub-pixel shifting.

Parameters

- **sino_180** (*array_like*) – 2D array. 180-degree sinogram.
- **start** (*float*) – Starting point for searching COR.
- **radius** (*float*) – Searching range: [start - radius; start + radius].
- **step** (*float*) – Searching step.
- **ratio** (*float*) – Ratio between a sample and the width of the sinogram.
- **denoise** (*bool, optional*) – Apply a smoothing filter.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **hor_drop** (*int or None, optional*) – Refer the method of "make_inverse_double_wedge_mask"
- **ver_drop** (*int or None, optional*) – Refer the method of "make_inverse_double_wedge_mask"

Returns
float – Center of rotation.

algotom.prep.calculation.**downsample_cor**(*image*, *dsp_fact0*, *dsp_fact1*)

Downsample an image by averaging.

Parameters

- **image** (*array_like*) – 2D array.
- **dsp_fact0** (*int*) – Downsampling factor along axis 0.
- **dsp_fact1** (*int*) – Downsampling factor along axis 1.

Returns
array_like – 2D array. Downsampled image.

algotom.prep.calculation.**find_center_vo**(*sino_180*, *start=None*, *stop=None*, *step=0.25*, *radius=4*, *ratio=0.5*, *dsp=True*, *ncore=None*, *hor_drop=None*, *ver_drop=None*)

Find the center-of-rotation using the method described in Ref. [1].

Parameters

- **sino_180** (*array_like*) – 2D array. 180-degree sinogram.
- **start** (*float*) – Starting point for searching CoR. Use the value of (width/2 - width/16) if None.
- **stop** (*float*) – Ending point for searching CoR. Use the value of (width/2 + width/16) if None.

- **step** (*float*) – Sub-pixel accuracy of estimated CoR.
- **radius** (*float*) – Searching range with the sub-pixel step.
- **ratio** (*float*) – Ratio between the sample and the width of the sinogram.
- **dsp** (*bool*) – Enable/disable downsampling.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **hor_drop** (*int or None, optional*) – Refer the method of "make_inverse_double_wedge_mask"
- **ver_drop** (*int or None, optional*) – Refer the method of "make_inverse_double_wedge_mask"

Returns
float – Center-of-rotation.

References

[1] : https://doi.org/10.1364/OE.22.019078

`algotom.prep.calculation.`**`calculate_curvature`**(*list_metric*)

Calculate the curvature of a fitted curve going through the minimum value of a metric list.

Parameters
list_metric (*array_like*) – 1D array. List of metrics.

Returns

- **curvature** (*float*) – Quadratic coefficient of the parabola fitting.
- **min_pos** (*float*) – Position of the minimum value with sub-pixel accuracy.

`algotom.prep.calculation.`**`correlation_metric`**(*mat1*, *mat2*)

Calculate the correlation metric. Smaller metric corresponds to better correlation.

Parameters

- **mat1** (*array_like*)
- **mat2** (*array_like*)

Returns
float – Correlation metric.

`algotom.prep.calculation.`**`search_overlap`**(*mat1*, *mat2*, *win_width*, *side*, *denoise=True*, *norm=False*, *use_overlap=False*)

Calculate the correlation metrics between a rectangular region, defined by the window width, on the utmost left/right side of image 2 and the same size region in image 1 where the region is slided across image 1.

Parameters

- **mat1** (*array_like*) – 2D array. Projection image or sinogram image.
- **mat2** (*array_like*) – 2D array. Projection image or sinogram image.
- **win_width** (*int*) – Width of the searching window.
- **side** (*{0, 1}*) – Only two options: 0 or 1. It is used to indicate the overlap side respects to image 1. "0" corresponds to the left side. "1" corresponds to the right side.

- **denoise** (*bool, optional*) – Apply the Gaussian filter if True.
- **norm** (*bool, optional*) – Apply the normalization if True.
- **use_overlap** (*bool, optional*) – Use the combination of images in the overlap area for calculating correlation coefficients if True.

Returns

- **list_metric** (*array_like*) – 1D array. List of the correlation metrics.
- **offset** (*int*) – Initial position of the searching window where the position corresponds to the center of the window.

`algotom.prep.calculation.`**`find_overlap`**(*mat1*, *mat2*, *win_width*, *side=None*, *denoise=True*, *norm=False*, *use_overlap=False*)

Find the overlap area and overlap side between two images (Ref. [1]) where the overlap side referring to the first image.

Parameters

- **mat1** (*array_like*) – 2D array. Projection image or sinogram image.
- **mat2** (*array_like*) – 2D array. Projection image or sinogram image.
- **win_width** (*int*) – Width of the searching window.
- **side** (*{None, 0, 1}, optional*) – Only there options: None, 0, or 1. "None" corresponding to fully automated determination. "0" corresponding to the left side. "1" corresponding to the right side.
- **denoise** (*bool, optional*) – Apply the Gaussian filter if True.
- **norm** (*bool, optional*) – Apply the normalization if True.
- **use_overlap** (*bool, optional*) – Use the combination of images in the overlap area for calculating correlation coefficients if True.

Returns

- **overlap** (*float*) – Width of the overlap area between two images.
- **side** (*int*) – Overlap side between two images.
- **overlap_position** (*float*) – Position of the window in the first image giving the best correlation metric.

References

[1] : https://doi.org/10.1364/OE.418448

`algotom.prep.calculation.`**`find_overlap_multiple`**(*list_mat*, *win_width*, *side=None*, *denoise=True*, *norm=False*, *use_overlap=False*)

Find the overlap-areas and overlap-sides of a list of images where the overlap side referring to the previous image.

Parameters

- **list_mat** (*list of array_like*) – List of 2D array. Projection image or sinogram image.
- **win_width** (*int*) – Width of the searching window.
- **side** (*{None, 0, 1}, optional*) – Only there options: None, 0, or 1. "None" corresponding to fully automated determination. "0" corresponding to the left side. "1" corresponding to the right side.

- **denoise** (*bool, optional*) – Apply the Gaussian filter if True.
- **norm** (*bool, optional*) – Apply the normalization if True.
- **use_overlap** (*bool, optional*) – Use the combination of images in the overlap area for calculating correlation coefficients if True.

Returns

list_overlap (*list of tuple of floats*) – List of [overlap, side, overlap_position]. overlap : Width of the overlap area between two images. side : Overlap side between two images. overlap_position : Position of the window in the first image giving the best correlation metric.

algotom.prep.calculation.**find_center_360**(*sino_360*, *win_width*, *side=None*, *denoise=True*, *norm=False*, *use_overlap=False*)

Find the center-of-rotation (COR) in a 360-degree scan with offset COR use the method presented in Ref. [1].

Parameters

- **sino_360** (*array_like*) – 2D array. 360-degree sinogram.
- **win_width** (*int*) – Window width used for finding the overlap area.
- **side** (*{None, 0, 1}, optional*) – Overlap size. Only there options: None, 0, or 1. "None" corresponding to fully automated determination. "0" corresponding to the left side. "1" corresponding to the right side.
- **denoise** (*bool, optional*) – Apply the Gaussian filter if True.
- **norm** (*bool, optional*) – Apply the normalization if True.
- **use_overlap** (*bool, optional*) – Use the combination of images in the overlap area for calculating correlation coefficients if True.

Returns

- **cor** (*float*) – Center-of-rotation.
- **overlap** (*float*) – Width of the overlap area between two halves of the sinogram.
- **side** (*int*) – Overlap side between two halves of the sinogram.
- **overlap_position** (*float*) – Position of the window in the first image giving the best correlation metric.

References

[1] : https://doi.org/10.1364/OE.418448

algotom.prep.calculation.**complex_gradient**(*mat*)

Return complex gradient of a 2D array.

algotom.prep.calculation.**find_shift_based_phase_correlation**(*mat1*, *mat2*, *gradient=True*)

Find relative translation in x and y direction between images with haft-pixel accuracy (Ref. [1]).

Parameters

- **mat1** (*array_like*) – 2D array. Projection image or sinogram image.
- **mat2** (*array_like*) – 2D array. Projection image or sinogram image.
- **gradient** (*bool, optional*) – Use the complex gradient of the input image for calculation.

Returns

- **ty** (*float*) – Translation in y-direction.
- **tx** (*float*) – Translation in x-direction.

References

[1] : https://doi.org/10.1049/el:20030666

algotom.prep.calculation.**find_center_based_phase_correlation**(*mat1*, *mat2*, *flip=True*, *gradient=True*)

Find the center-of-rotation (COR) using projection images at 0-degree and 180-degree.

Parameters

- **mat1** (*array_like*) – 2D array. Projection image at 0-degree.
- **mat2** (*array_like*) – 2D array. Projection image at 180-degree.
- **flip** (*bool, optional*) – Flip the 180-degree projection in the left-right direction if True.
- **gradient** (*bool, optional*) – Use the complex gradient of the input image for calculation.

Returns

cor (*float*) – Center-of-rotation.

algotom.prep.calculation.**find_center_projection**(*mat1*, *mat2*, *flip=True*, *chunk_height=None*, *start_row=None*, *denoise=True*, *norm=False*, *use_overlap=False*)

Find the center-of-rotation (COR) using projection images at 0-degree and 180-degree based on a method in Ref. [1].

Parameters

- **mat1** (*array_like*) – 2D array. Projection image at 0-degree.
- **mat2** (*array_like*) – 2D array. Projection image at 180-degree.
- **flip** (*bool, optional*) – Flip the 180-degree projection in the left-right direction if True.
- **chunk_height** (*int or float, optional*) – Height of the sub-area of projection images. If a float is given, it must be in the range of [0.0, 1.0].
- **start_row** (*int, optional*) – Starting row used to extract the sub-area.
- **denoise** (*bool, optional*) – Apply the Gaussian filter if True.
- **norm** (*bool, optional*) – Apply the normalization if True.
- **use_overlap** (*bool, optional*) – Use the combination of images in the overlap area for calculating correlation coefficients if True.

Returns

cor (*float*) – Center-of-rotation.

References

[1] : https://doi.org/10.1364/OE.418448

algotom.prep.calculation.**calculate_reconstructable_height**(*y_start*, *y_stop*, *pitch*, *scan_type*)

Calculate reconstructable height in a helical scan.

Parameters

- **y_start** (*float*) – Y-position of the stage at the beginning of the scan.
- **y_stop** (*float*) – Y-position of the stage at the end of the scan.
- **pitch** (*float*) – The distance which the y-stage is translated in one full rotation.
- **scan_type** (*{"180", "360"}*) – One of two options: "180" for generating a 180-degree sinogram or "360" for generating a 360-degree sinogram.

Returns

- **y_s** (*float*) – Starting point of the reconstructable height.
- **y_e** (*float*) – End point of the reconstructable height.

algotom.prep.calculation.**calculate_maximum_index**(*y_start*, *y_stop*, *pitch*, *pixel_size*, *scan_type*)

Calculate the maximum index of a reconstructable slice in a helical scan.

Parameters

- **y_start** (*float*) – Y-position of the stage at the beginning of the scan.
- **y_stop** (*float*) – Y-position of the stage at the end of the scan.
- **pitch** (*float*) – The distance which the y-stage is translated in one full rotation.
- **pixel_size** (*float*) – Pixel size. The unit must be the same as y-position.
- **scan_type** (*{"180", "360"}*) – One of two options: "180" for generating a 180-degree sinogram or "360" for generating a 360-degree sinogram.

Returns

int – Maximum index of reconstructable slices.

algotom.prep.conversion

Module of conversion methods in the preprocessing stage:

- Stitching images.
- Joining images if there is no overlapping.
- Converting a 360-degree sinogram with offset center-of-rotation (COR) to a 180-degree sinogram.
- Extending a 360-degree sinogram with offset COR for direct reconstruction instead of converting it to a 180-degree sinogram.
- Converting a 180-degree sinogram to a 360-sinogram.
- Generating a sinogram from a helical data.

Functions:

make_weight_matrix(mat1, mat2, overlap, side)	Generate a linear-ramp weighting matrix for image stitching.
stitch_image(mat1, mat2, overlap, side[, ...])	Stitch projection images or sinogram images using a linear ramp.
join_image(mat1, mat2, joint_width, side[, ...])	Join projection images or sinogram images.
stitch_image_multiple(list_mat, list_overlap)	Stitch list of projection images or sinogram images using a linear ramp.
join_image_multiple(list_mat, list_joint[, ...])	Join list of projection images or sinogram images.
convert_sinogram_360_to_180(sino_360, cor[, ...])	Convert a 360-degree sinogram to a 180-degree sinogram.
convert_sinogram_180_to_360(sino_180, center)	Convert a 180-degree sinogram to a 360-degree sinogram (Ref.
extend_sinogram(sino_360, cor[, apply_log])	Extend a 360-degree sinogram (with offset center-of-rotation) for later reconstruction (Ref.
generate_sinogram_helical_scan(index, ...[, ...])	Generate a 180-degree/360-degree sinogram from a helical-scan dataset which is a hdf/nxs object (Ref.
generate_full_sinogram_helical_scan(index, ...)	Generate a full sinogram from a helical-scan dataset which is a hdf/nxs object (Ref.

algotom.prep.conversion.**make_weight_matrix**(*mat1*, *mat2*, *overlap*, *side*)

Generate a linear-ramp weighting matrix for image stitching.

Parameters

- **mat1** (*array_like*) – 2D array. Projection image or sinogram image.
- **mat2** (*array_like*) – 2D array. Projection image or sinogram image.
- **overlap** (*int*) – Width of the overlap area between two images.
- **side** (*{0, 1}*) – Only two options: 0 or 1. It is used to indicate the overlap side respects to image 1. "0" corresponds to the left side. "1" corresponds to the right side.

algotom.prep.conversion.**stitch_image**(*mat1*, *mat2*, *overlap*, *side*, *wei_mat1=None*, *wei_mat2=None*, *norm=True*, *total_width=None*)

Stitch projection images or sinogram images using a linear ramp.

Parameters

- **mat1** (*array_like*) – 2D array. Projection image or sinogram image.
- **mat2** (*array_like*) – 2D array. Projection image or sinogram image.
- **overlap** (*float*) – Width of the overlap area between two images.
- **side** (*{0, 1}*) – Only two options: 0 or 1. It is used to indicate the overlap side respects to image 1. "0" corresponds to the left side. "1" corresponds to the right side.
- **wei_mat1** (*array_like, optional*) – Weighting matrix used for image 1.
- **wei_mat2** (*array_like, optional*) – Weighting matrix used for image 2.
- **norm** (*bool, optional*) – Enable/disable normalization before stitching.
- **total_width** (*int, optional*) – Final width of the stitched image.

Returns

array_like – Stitched image.

algotom.prep.conversion.**join_image**(*mat1*, *mat2*, *joint_width*, *side*, *norm=True*, *total_width=None*)

Join projection images or sinogram images. This is useful for fixing the problem of non-overlap between images.

Parameters

- **mat1** (*array_like*) – 2D array. Projection image or sinogram image.
- **mat2** (*array_like*) – 2D array. Projection image or sinogram image.
- **joint_width** (*float*) – Width of the joint area between two images.
- **side** (*{0, 1}*) – Only two options: 0 or 1. It is used to indicate the overlap side respects to image 1. "0" corresponds to the left side. "1" corresponds to the right side.
- **norm** (*bool*) – Enable/disable normalization before joining.
- **total_width** (*int, optional*) – Final width of the joined image.

Returns

array_like – Stitched image.

algotom.prep.conversion.**stitch_image_multiple**(*list_mat*, *list_overlap*, *norm=True*, *total_width=None*)

Stitch list of projection images or sinogram images using a linear ramp.

Parameters

- **list_mat** (*list of array_like*) – List of 2D array. Projection image or sinogram image.
- **list_overlap** (*list of tuple of floats*) – List of [overlap, side]. overlap : Width of the overlap area between two images. side : Overlap side between two images.
- **norm** (*bool, optional*) – Enable/disable normalization before stitching.
- **total_width** (*int, optional*) – Final width of the stitched image.

Returns

array_like – Stitched image.

algotom.prep.conversion.**join_image_multiple**(*list_mat*, *list_joint*, *norm=True*, *total_width=None*)

Join list of projection images or sinogram images. This is useful for fixing the problem of non-overlap between images.

Parameters

- **list_mat** (*list of array_like*) – List of 2D array. Projection image or sinogram image.
- **list_joint** (*list of tuple of floats*) – List of [joint_width, side]. joint_width : Width of the joint area between two images. side : Overlap side between two images.
- **norm** (*bool, optional*) – Enable/disable normalization before stitching.
- **total_width** (*int, optional*) – Final width of the stitched image.

Returns

array_like – Stitched image.

algotom.prep.conversion.**convert_sinogram_360_to_180**(*sino_360*, *cor*, *wei_mat1=None*, *wei_mat2=None*, *norm=True*, *total_width=None*)

Convert a 360-degree sinogram to a 180-degree sinogram.

Parameters

- **sino_360** (*array_like*) – 2D array. 360-degree sinogram.
- **cor** (*float or tuple of float*) – Center-of-rotation or (Overlap_area, overlap_side).

- **wei_mat1** (*array_like, optional*) – Weighting matrix used for the 1st haft of the sinogram.
- **wei_mat2** (*array_like, optional*) – Weighting matrix used for the 2nd haft of the sinogram.
- **norm** (*bool, optional*) – Enable/disable normalization before stitching.
- **total_width** (*int, optional*) – Final width of the stitched image.

Returns

- **sino_stitch** (*array_like*) – Converted sinogram.
- **cor** (*float*) – Updated center-of-rotation referred to the converted sinogram.

algotom.prep.conversion.**convert_sinogram_180_to_360**(*sino_180*, *center*)

Convert a 180-degree sinogram to a 360-degree sinogram (Ref. [1]).

Parameters

- **sino_180** (*array_like*) – 2D array. 180-degree sinogram.
- **center** (*float*) – Center-of-rotation.

Returns

array_like – 360-degree sinogram.

References

[1] : https://doi.org/10.1364/OE.22.019078

algotom.prep.conversion.**extend_sinogram**(*sino_360*, *cor*, *apply_log=True*)

Extend a 360-degree sinogram (with offset center-of-rotation) for later reconstruction (Ref. [1]).

Parameters

- **sino_360** (*array_like*) – 2D array. 360-degree sinogram.
- **cor** (*float or tuple of float*) – Center-of-rotation or (Overlap_area, overlap_side).
- **apply_log** (*bool, optional*) – Apply the logarithm function if True.

Returns

- **sino_pad** (*array_like*) – Extended sinogram.
- **cor** (*float*) – Updated center-of-rotation referred to the converted sinogram.

References

[1] : https://doi.org/10.1364/OE.418448

algotom.prep.conversion.**generate_sinogram_helical_scan**(*index*, *tomo_data*, *num_proj*, *pixel_size*, *y_start*, *y_stop*, *pitch*, *scan_type='180'*, *angles=None*, *flat=None*, *dark=None*, *mask=None*, *crop=(0, 0, 0, 0)*)

Generate a 180-degree/360-degree sinogram from a helical-scan dataset which is a hdf/nxs object (Ref. [1]).

Parameters

- **index** (*int*) – Index of the sinogram.
- **tomo_data** (*hdf object.*) – 3D array.
- **num_proj** (*int*) – Number of projections per 180-degree.

- **pixel_size** (*float*) – Pixel size. The unit must be the same as y-position.
- **y_start** (*float*) – Y-position of the stage at the beginning of the scan.
- **y_stop** (*float*) – Y-position of the stage at the end of the scan.
- **pitch** (*float*) – The distance which the y-stage is translated in one full rotation.
- **scan_type** (*{"180", "360"}*) – One of two options: "180" for generating a 180-degree sinogram or "360" for generating a 360-degree sinogram.
- **angles** (*array_like, optional*) – 1D array. Angles (degree) corresponding to acquired projections.
- **flat** (*array_like, optional*) – Flat-field image used for flat-field correction.
- **dark** (*array_like, optional*) – Dark-field image used for flat-field correction.
- **mask** (*array_like, optional*) – Used for removing streak artifacts caused by blobs in the flat-field image.
- **crop** (*tuple of int, optional*) – Used for cropping images.

Returns

- **sinogram** (*array_like*) – 2D array. 180-degree sinogram or 360-degree sinogram.
- **list_angle** (*array_like*) – 1D array. List of angles corresponding to the generated sinogram.

References

[1] : https://doi.org/10.1364/OE.418448

algotom.prep.conversion.**generate_full_sinogram_helical_scan**(*index*, *tomo_data*, *num_proj*, *pixel_size*, *y_start*, *y_stop*, *pitch*, *scan_type='180'*, *angles=None*, *flat=None*, *dark=None*, *mask=None*, *crop=(0, 0, 0, 0)*)

Generate a full sinogram from a helical-scan dataset which is a hdf/nxs object (Ref. [1]). Full sinogram is all 1D projections of the same slice of a sample staying inside the field of view.

Parameters

- **index** (*int*) – Index of the sinogram.
- **tomo_data** (*hdf object.*) – 3D array.
- **num_proj** (*int*) – Number of projections per 180-degree.
- **pixel_size** (*float*) – Pixel size. The unit must be the same as y-position.
- **y_start** (*float*) – Y-position of the stage at the beginning of the scan.
- **y_stop** (*float*) – Y-position of the stage at the end of the scan.
- **pitch** (*float*) – The distance which the y-stage is translated in one full rotation.
- **scan_type** (*{"180", "360"}*) – Data acquired is the 180-degree type or 360-degree type [1].
- **angles** (*array_like, optional*) – 1D array. Angles (degree) corresponding to acquired projections.
- **flat** (*array_like, optional*) – Flat-field image used for flat-field correction.
- **dark** (*array_like, optional*) – Dark-field image used for flat-field correction.

- **mask** (*array_like, optional*) – Used for removing streak artifacts caused by blobs in the flat-field image.
- **crop** (*tuple of int, optional*) – Used for cropping images.

Returns

- **sinogram** (*array_like*) – 2D array. Full sinogram.
- **list_angle** (*array_like*) – 1D array. List of angles corresponding to the generated sinogram.

References

[1] : https://doi.org/10.1364/OE.418448

algotom.prep.correction

Module of correction methods in the preprocessing stage:

- Flat-field correction.
- Distortion correction.
- MTF deconvolution.
- Tilted sinogram generation.
- Tilted 1D intensity-profile generation.
- Beam hardening correction.
- Sinogram upsampling.

Functions:

flat_field_correction(proj, flat, dark[, ...])	Perform flat-field correction with options to remove zinger artifacts and/or stripe artifacts.
unwarp_projection(proj, xcenter, ycenter, ...)	Apply distortion correction to a projection image using the polynomial backward model (Ref.
unwarp_sinogram(data, index, xcenter, ...)	Unwarp sinogram [:,index.:] of a 3D tomographic dataset or a hdf/nxs object.
unwarp_sinogram_chunk(data, start_index, ...)	Unwarp chunk of sinograms [:, start_index: stop_index, :] of a 3D tomographic dataset or a hdf/nxs object.
mtf_deconvolution(mat, window, pad)	Deconvolve a projection-image using division in the Fourier domain.
generate_tilted_sinogram(data, index, angle, ...)	Generate a tilted sinogram of a 3D tomographic dataset or a hdf/nxs object.
generate_tilted_sinogram_chunk(data, ...)	Generate a chunk of tilted sinograms of a 3D tomographic dataset or a hdf/nxs object.
generate_tilted_profile_line(mat, index, angle)	Generate a tilted horizontal intensity-profile of an image.
generate_tilted_profile_chunk(mat, ...)	Generate a chunk of tilted horizontal intensity-profiles of an image.
non_linear_function(intensity, q, n[, opt])	Function used to define the response curve.
beam_hardening_correction(mat, q, n[, opt])	Correct the grayscale values of a normalized image using a non-linear function.
upsample_sinogram(sinogram, scale[, center, ...])	Upsample a sinogram-image along angular direction based on the double-wedge filter (Ref.

algotom.prep.correction.**flat_field_correction**(*proj*, *flat*, *dark*, *ratio=1.0*, *use_dark=True*, ***options*)

Perform flat-field correction with options to remove zinger artifacts and/or stripe artifacts.

Parameters

- **proj** (*array_like*) – 3D or 2D array. Projection images or a sinogram image.
- **flat** (*array_like*) – 2D or 1D array. Flat-field image or a single row of it.
- **dark** (*array_like*) – 2D or 1D array. Dark-field image or a single row of it.
- **ratio** (*float*) – Ratio between exposure time used for recording projections and exposure time used for recording flat field.
- **use_dark** (*bool*) – Subtracting dark field if True.
- **options** (*dict, optional*) – Apply a zinger removal method and/or ring removal methods. E.g option1={"method": "dezinger", "para1": 0.001, "para2": 1}, option2={"method": "remove_stripe_based_sorting", "para1": 15, "para2": 1}

Returns

array_like – 3D or 2D array. Corrected projections or corrected sinograms.

algotom.prep.correction.**unwarp_projection**(*proj*, *xcenter*, *ycenter*, *list_fact*)

Apply distortion correction to a projection image using the polynomial backward model (Ref. [1]).

Parameters

- **proj** (*array_like*) – 2D array. Projection image.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.

- **list_fact** (*list of float*) – Polynomial coefficients of the backward model.

Returns
array_like – 2D array. Distortion corrected.

References

[1] : https://doi.org/10.1364/OE.23.032859

algotom.prep.correction.**unwarp_sinogram**(*data*, *index*, *xcenter*, *ycenter*, *list_fact*, ***option*)

Unwarp sinogram [:,index.:] of a 3D tomographic dataset or a hdf/nxs object.

Parameters

- **data** (*array_like or hdf object*) – 3D array.
- **index** (*int*) – Index of the sinogram.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*list of float*) – Polynomial coefficients of the backward model.
- **option** (*list or tuple of int*) – To extract subset data along axis 0 from a hdf object. E.g option = (start, stop, step)

Returns
array_like – 2D array. Distortion-corrected sinogram.

algotom.prep.correction.**unwarp_sinogram_chunk**(*data*, *start_index*, *stop_index*, *xcenter*, *ycenter*, *list_fact*, ***option*)

Unwarp chunk of sinograms [:, start_index: stop_index, :] of a 3D tomographic dataset or a hdf/nxs object.

Parameters

- **data** (*array_like or hdf object*) – 3D array.
- **start_index** (*int*) – Starting index of sinograms.
- **stop_index** (*int*) – Stopping index of sinograms.
- **xcenter** (*float*) – Center of distortion in x-direction.
- **ycenter** (*float*) – Center of distortion in y-direction.
- **list_fact** (*list of float*) – Polynomial coefficients of the backward model.
- **option** (*list or tuple of int*) – To extract subset data along axis 0 from a hdf object. E.g option = [start, stop, step]

Returns
array_like – 3D array. Distortion corrected.

algotom.prep.correction.**mtf_deconvolution**(*mat*, *window*, *pad*)

Deconvolve a projection-image using division in the Fourier domain. Window can be determined using the approach in Ref. [1].

Parameters

- **mat** (*array_like*) – 2D array. Projection image.
- **window** (*array_like*) – 2D array. MTF function.
- **pad** (*int*) – Padding width to reduce the side effects of the Fourier transform.

Returns
array_like – 2D array. Deconvolved image.

References

[1] : https://doi.org/10.1117/12.2530324

algotom.prep.correction.**generate_tilted_sinogram**(*data*, *index*, *angle*, ***option*)

Generate a tilted sinogram of a 3D tomographic dataset or a hdf/nxs object.

Parameters

- **data** (*array_like or hdf object*) – 3D array.
- **index** (*int*) – Index of the sinogram.
- **angle** (*float*) – Tilted angle in degree.
- **option** (*list or tuple of int*) – To extract subset data along axis 0 from a hdf object. E.g option = (start, stop, step)

Returns
array_like – 2D array. Tilted sinogram.

algotom.prep.correction.**generate_tilted_sinogram_chunk**(*data*, *start_index*, *stop_index*, *angle*, ***option*)

Generate a chunk of tilted sinograms of a 3D tomographic dataset or a hdf/nxs object.

Parameters

- **data** (*array_like or hdf object*) – 3D array.
- **start_index** (*int*) – Starting index of sinograms.
- **stop_index** (*int*) – Stopping index of sinograms.
- **angle** (*float*) – Tilted angle in degree.
- **option** (*list or tuple of int*) – To extract subset data along axis 0 from a hdf object. E.g option = (start, stop, step)

Returns
array_like – 3D array. Chunk of tilted sinograms.

algotom.prep.correction.**generate_tilted_profile_line**(*mat*, *index*, *angle*)

Generate a tilted horizontal intensity-profile of an image.

Parameters

- **mat** (*array_like*) – 2D array.
- **index** (*int*) – Index of the line.
- **angle** (*float*) – Tilted angle in degree.

Returns
array_like – 1D array.

algotom.prep.correction.**generate_tilted_profile_chunk**(*mat*, *start_index*, *stop_index*, *angle*)

Generate a chunk of tilted horizontal intensity-profiles of an image.

Parameters

- **mat** (*array_like*) – 2D array.

- **start_index** (*int*) – Starting index of lines.
- **stop_index** (*int*) – Stopping index of lines.
- **angle** (*float*) – Tilted angle in degree.

Returns
array_like – 2D array.

algotom.prep.correction.**non_linear_function**(*intensity*, *q*, *n*, *opt=True*)

Function used to define the response curve.

Parameters

- **intensity** (*float*) – Values stay in the range of [0; 1]
- **q** (*float*) – Positive number.
- **n** (*float*) – Positive number. Must larger than 1.
- **opt** (*bool*) – True: Curve more to values closer to 1.0. False: Curve more to values closer to 0.0

Returns
float

algotom.prep.correction.**beam_hardening_correction**(*mat*, *q*, *n*, *opt=True*)

Correct the grayscale values of a normalized image using a non-linear function.

Parameters

- **mat** (*array_like*) – Normalized projection image or sinogram image.
- **q** (*float*) – Positive number. Recommended range [0.005, 50].
- **n** (*float*) – Positive number. Must larger than 1.
- **opt** (*bool*) – True: Curve towards 0.0. False: Curve towards 1.0.

Returns
array_like – Corrected image.

algotom.prep.correction.**upsample_sinogram**(*sinogram*, *scale*, *center=0*, *sino_type='180'*, *iteration=1*, *pad=50*)

Upsample a sinogram-image along angular direction based on the double-wedge filter (Ref. [1]).

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **scale** (*int*) – Upscale 2n_x time. E.g. 2, 4, 6.
- **center** (*float, optional*) – Center-of-rotation. No need for a 360-sinogram.
- **sino_type** (*{"180", "360"}*) – Sinogram type : 180-degree or 360-degree.
- **iteration** (*int, optional*) – Number of iteration for the double-wedge filter.
- **pad** (*int, optional*) – Padding width for FFT.

Returns
array_like – Upsampled sinogram.

algotom.prep.filtering

Module of filtering methods in the preprocessing stage:

- Fresnel filter (denoising or low-pass filter), a simplified version of the well-known Paganin's filter.
- Double-wedge filter.

Functions:

make_fresnel_window(height, width, ratio, dim)	Create a low pass window based on the Fresnel propagator.
fresnel_filter(mat, ratio[, dim, window, ...])	Apply a low-pass filter based on the Fresnel propagator to an image (Ref.
make_double_wedge_mask(height, width, radius)	Generate a double-wedge binary mask using Eq.
double_wedge_filter(sinogram[, center, ...])	Apply double-wedge filter to a sinogram image (Ref.

algotom.prep.filtering.**make_fresnel_window**(*height*, *width*, *ratio*, *dim*)

Create a low pass window based on the Fresnel propagator. It is used to denoise a projection image (dim=2) or a sinogram image (dim=1).

Parameters

- **height** (*int*) – Image height
- **width** (*int*) – Image width
- **ratio** (*float*) – To define the shape of the window.
- **dim** (*{1, 2}*) – Use "1" if working on a sinogram image and "2" if working on a projection image.

Returns

array_like – 2D array.

algotom.prep.filtering.**fresnel_filter**(*mat*, *ratio*, *dim=1*, *window=None*, *pad=150*, *apply_log=True*)

Apply a low-pass filter based on the Fresnel propagator to an image (Ref. [1]). It can be used for improving the contrast of an image. It's simpler than the well-known Paganin's filter (Ref. [2]).

Parameters

- **mat** (*array_like*) – 2D array. Projection image or sinogram image.
- **ratio** (*float*) – Define the shape of the window. Larger is more smoothing.
- **dim** (*{1, 2}*) – Use "1" if working on a sinogram image and "2" if working on a projection image.
- **window** (*array_like, optional*) – Window for deconvolution.
- **pad** (*int*) – Padding width.
- **apply_log** (*bool, optional*) – Apply the logarithm function to the sinogram before filtering.

Returns

array_like – 2D array. Filtered image.

References

[1] : https://doi.org/10.1364/OE.418448

[2] : https://tinyurl.com/2f8nv875

algotom.prep.filtering.**make_double_wedge_mask**(*height*, *width*, *radius*)

Generate a double-wedge binary mask using Eq. (3) in Ref. [1]. Values outside the double-wedge region correspond to 0.0.

Parameters

- **height** (*int*) – Image height.
- **width** (*int*) – Image width.
- **radius** (*int*) – Radius of an object, in pixel unit.

Returns

array_like – 2D binary mask.

References

[1] : https://doi.org/10.1364/OE.22.019078

algotom.prep.filtering.**double_wedge_filter**(*sinogram*, *center=0*, *sino_type='180'*, *iteration=5*, *mask=None*, *ratio=1.0*, *pad=250*)

Apply double-wedge filter to a sinogram image (Ref. [1]).

Parameters

- **sinogram** (*array_like*) – 2D array. 180-degree sinogram or 360-degree sinogram.
- **center** (*float, optional*) – Center-of-rotation. No need for a 360-sinogram.
- **sino_type** (*{"180", "360"}*) – Sinogram type : 180-degree or 360-degree.
- **iteration** (*int*) – Number of iteration.
- **mask** (*array_like, optional*) – Double-wedge binary mask.
- **ratio** (*float, optional*) – Define the cut-off angle of the double-wedge filter.
- **pad** (*int*) – Padding width.

Returns

array_like – 2D array. Filtered sinogram.

References

[1] : https://doi.org/10.1364/OE.418448

algotom.prep.removal

Module of removal methods in the preprocessing stage:

- Many methods for removing stripe artifact in a sinogram (<-> ring artifact in a reconstructed image).
- A zinger removal method.
- Blob removal methods.

Functions:

remove_stripe_based_sorting(sinogram[, ...])	Remove stripe artifacts in a sinogram using the sorting technique, algorithm 3 in Ref.
remove_stripe_based_filtering(sinogram[, ...])	Remove stripe artifacts in a sinogram using the filtering technique, algorithm 2 in Ref.
remove_stripe_based_fitting(sinogram[, ...])	Remove stripe artifacts in a sinogram using the fitting technique, algorithm 1 in Ref.
remove_large_stripe(sinogram[, snr, size, ...])	Remove large stripe artifacts in a sinogram, algorithm 5 in Ref.
remove_dead_stripe(sinogram[, snr, size, ...])	Remove unresponsive or fluctuating stripe artifacts in a sinogram, algorithm 6 in Ref.
remove_all_stripe(sinogram[, snr, la_size, ...])	Remove all types of stripe artifacts in a sinogram by combining algorithm 6, 5, 4, and 3 in Ref.
remove_stripe_based_2d_filtering_sorting(...)	Remove stripes using a 2D low-pass filter and the sorting-based technique, algorithm in section 3.3.4 in Ref.
remove_stripe_based_normalization(sinogram)	Remove stripes using the method in Ref.
remove_stripe_based_regularization(sinogram)	Remove stripes using the method in Ref.
remove_stripe_based_fft(sinogram[, u, n, v, ...])	Remove stripes using the method in Ref.
remove_stripe_based_wavelet_fft(sinogram[, ...])	Remove stripes using the method in Ref.
remove_stripe_based_interpolation(sinogram)	Combination of algorithm 4, 5, and 6 in Ref.
check_zinger_size(mat, max_size)	Check if the size of a zinger is smaller than a given size.
select_zinger(mat, max_size)	Select zingers smaller than a certain size.
remove_zinger(mat, threshold[, size, check_size])	Remove zinger using the method in Ref.
generate_blob_mask(flat, size, snr)	Generate a binary mask of blobs from a flat-field image (Ref.
remove_blob_1d(sino_1d, mask_1d)	Remove blobs in one row of a sinogram, e.g.
remove_blob(mat, mask)	Remove blobs in an image.

algotom.prep.removal.**remove_stripe_based_sorting**(*sinogram*, *size=21*, *dim=1*, ***options*)

Remove stripe artifacts in a sinogram using the sorting technique, algorithm 3 in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **size** (*int*) – Window size of the median filter.
- **dim** (*{1, 2}, optional*) – Dimension of the window.
- **options** (*dict, optional*) – Use another smoothing filter rather than the median filter. E.g. options={"method": "gaussian_filter", "para1": (1,21)}

Returns
array_like – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1364/OE.26.028396

algotom.prep.removal.**remove_stripe_based_filtering**(*sinogram*, *sigma=3*, *size=21*, *dim=1*, *sort=True*, ***options*)

Remove stripe artifacts in a sinogram using the filtering technique, algorithm 2 in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image
- **sigma** (*int*) – Sigma of the Gaussian window used to separate the low-pass and high-pass components of the intensity profile of each column.
- **size** (*int*) – Window size of the median filter.
- **dim** (*{1, 2}, optional*) – Dimension of the window.
- **sort** (*bool, optional*) – Apply sorting if True.
- **options** (*dict, optional*) – Use another smoothing filter rather than the median filter. E.g. options={"method": "gaussian_filter", "para1": (1,21))}.

Returns
array_like – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1364/OE.26.028396

algotom.prep.removal.**remove_stripe_based_fitting**(*sinogram*, *order=2*, *sigma=10*, *sort=False*, *num_chunk=1*, ***options*)

Remove stripe artifacts in a sinogram using the fitting technique, algorithm 1 in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image
- **order** (*int*) – Polynomial fit order.
- **sigma** (*int*) – Sigma of the Gaussian window in the x-direction. Smaller is stronger.
- **sort** (*bool, optional*) – Apply sorting if True.
- **num_chunk** (*int*) – Number of chunks of rows to apply the fitting.
- **options** (*dict, optional*) – Use another smoothing filter rather than the Fourier gaussian filter. E.g. options={"method": "gaussian_filter", "para1": (1,21))}.

Returns
array_like – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1364/OE.26.028396

algotom.prep.removal.**remove_large_stripe**(*sinogram*, *snr=3.0*, *size=51*, *drop_ratio=0.1*, *norm=True*, ***options*)

Remove large stripe artifacts in a sinogram, algorithm 5 in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image
- **snr** (*float*) – Ratio (>1.0) for stripe detection. Greater is less sensitive.
- **size** (*int*) – Window size of the median filter.
- **drop_ratio** (*float, optional*) – Ratio of pixels to be dropped, which is used to reduce the possibility of the false detection of stripes.
- **norm** (*bool, optional*) – Apply normalization if True.
- **options** (*dict, optional*) – Use another smoothing filter rather than the median filter. E.g. options={"method": "gaussian_filter", "para1": (1,21))}.

Returns

array_like – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1364/OE.26.028396

algotom.prep.removal.**remove_dead_stripe**(*sinogram*, *snr=3.0*, *size=51*, *residual=True*, *smooth_strength=10*)

Remove unresponsive or fluctuating stripe artifacts in a sinogram, algorithm 6 in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **snr** (*float*) – Ratio (>1.0) for stripe detection. Greater is less sensitive.
- **size** (*int*) – Window size of the median filter.
- **residual** (*bool, optional*) – Removing residual stripes if True.
- **smooth_strength** (*int, optional*) – Window size of the uniform filter used to detect stripes.

Returns

ndarray – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1364/OE.26.028396

algotom.prep.removal.**remove_all_stripe**(*sinogram*, *snr=3.0*, *la_size=51*, *sm_size=21*, *drop_ratio=0.1*, *dim=1*, ***options*)

Remove all types of stripe artifacts in a sinogram by combining algorithm 6, 5, 4, and 3 in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **snr** (*float*) – Ratio (>1.0) for stripe detection. 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.
- **drop_ratio** (*float, optional*) – Ratio of pixels to be dropped, which is used to reduce the possibility of the false detection of stripes.
- **dim** (*{1, 2}, optional*) – Dimension of the window.
- **options** (*dict, optional*) – Use another smoothing filter rather than the median filter. E.g. options={"method": "gaussian_filter", "para1": (1,21))}

Returns

array_like – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1364/OE.26.028396

algotom.prep.removal.**remove_stripe_based_2d_filtering_sorting**(*sinogram*, *sigma=3*, *size=21*, *dim=1*, ***options*)

Remove stripes using a 2D low-pass filter and the sorting-based technique, algorithm in section 3.3.4 in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **sigma** (*int*) – Sigma of the Gaussian window.
- **size** (*int*) – Window size of the median filter.
- **dim** (*{1, 2}, optional*) – Dimension of the window.

Returns

array_like – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1117/12.2530324

algotom.prep.removal.**remove_stripe_based_normalization**(*sinogram*, *sigma=15*, *num_chunk=1*, *sort=True*, ***options*)

Remove stripes using the method in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **sigma** (*int*) – Sigma of the Gaussian window.
- **num_chunk** (*int*) – Number of chunks of rows.
- **sort** (*bool, optional*) – Apply sorting (Ref. [2]) if True.
- **options** (*dict, optional*) – Use another smoothing 1D-filter rather than the Gaussian filter. E.g. options={"method": "median_filter", "para1": 21)}.

Returns

array_like – 2D array. Stripe-removed sinogram.

References

[1] : https://www.mcs.anl.gov/research/projects/X-ray-cmt/rivers/tutorial.html

[2] : https://doi.org/10.1364/OE.26.028396

algotom.prep.removal.**remove_stripe_based_regularization**(*sinogram*, *alpha=0.0005*, *num_chunk=1*, *apply_log=True*, *sort=True*)

Remove stripes using the method in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **alpha** (*float*) – Regularization parameter, e.g. 0.0005. Smaller is stronger.
- **num_chunk** (*int*) – Number of chunks of rows.
- **apply_log** (*bool*) – Apply the logarithm function to the sinogram if True.
- **sort** (*bool, optional*) – Apply sorting (Ref. [2]) if True.

Returns

array_like – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1016/j.aml.2010.08.022

[2] : https://doi.org/10.1364/OE.26.028396

algotom.prep.removal.**remove_stripe_based_fft**(*sinogram*, *u=20*, *n=8*, *v=1*, *sort=False*)

Remove stripes using the method in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **u** (*int*) – Cutoff frequency.

- **n** (*int*) – Filter order.
- **v** (*int*) – Number of rows (* 2) to be applied the filter.
- **sort** (*bool, optional*) – Apply sorting (Ref. [2]) if True.

Returns
ndarray – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1063/1.1149043

[2] : https://doi.org/10.1364/OE.26.028396

algotom.prep.removal.**remove_stripe_based_wavelet_fft**(*sinogram*, *level=5*, *size=1*, *wavelet_name='db9'*, *window_name='gaussian'*, *sort=False*, ***options*)

Remove stripes using the method in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **level** (*int*) – Wavelet decomposition level.
- **size** (*int*) – Damping parameter. Larger is stronger.
- **wavelet_name** (*str*) – Name of a wavelet. Search pywavelets API for a full list.
- **window_name** (*str*) – High-pass window. Two options: "gaussian" or "butter".
- **sort** (*bool, optional*) – Apply sorting (Ref. [2]) if True.
- **options** (*dict, optional*) – Use another smoothing filter rather than the fft-gaussian-filter. E.g. options={"method": "gaussian_filter", "para1": (1,11))}

Returns
array_like – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1364/OE.17.008567

[2] : https://doi.org/10.1364/OE.26.028396

algotom.prep.removal.**remove_stripe_based_interpolation**(*sinogram*, *snr=3.0*, *size=51*, *drop_ratio=0.1*, *norm=True*, *kind='linear'*, ***options*)

Combination of algorithm 4, 5, and 6 in Ref. [1]. Angular direction is along the axis 0.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image
- **snr** (*float*) – Ratio (>1.0) for stripe detection. Greater is less sensitive.
- **size** (*int*) – Window size of the median filter used to detect stripes.
- **drop_ratio** (*float, optional*) – Ratio of pixels to be dropped, which is used to reduce the possibility of the false detection of stripes.
- **norm** (*bool, optional*) – Apply normalization if True.

- **kind** (*{'linear', 'cubic', 'quintic'}, optional*) – The kind of spline interpolation to use. Default is 'linear'.
- **options** (*dict, optional*) – Use another smoothing filter rather than the median filter. E.g. options={"method": "gaussian_filter", "para1": (1,21))}

Returns
array_like – 2D array. Stripe-removed sinogram.

References

[1] : https://doi.org/10.1364/OE.26.028396

`algotom.prep.removal.`**`check_zinger_size`**(*mat*, *max_size*)

Check if the size of a zinger is smaller than a given size.

Parameters

- **mat** (*array_like*) – 2D array.
- **max_size** (*int*) – Maximum size.

Returns
bool

`algotom.prep.removal.`**`select_zinger`**(*mat*, *max_size*)

Select zingers smaller than a certain size.

Parameters

- **mat** (*array_like*) – 2D array.
- **max_size** (*int*) – Maximum size in pixel.

Returns
array_like – 2D binary array.

`algotom.prep.removal.`**`remove_zinger`**(*mat*, *threshold*, *size=2*, *check_size=False*)

Remove zinger using the method in Ref. [1], working on a projection image or sinogram image.

Parameters

- **mat** (*array_like*) – 2D array. Projection image or sinogram image.
- **threshold** (*float*) – Threshold to segment zingers. Smaller is more sensitive. Recommended range [0.05, 0.1].
- **size** (*int*) – Size of a zinger.
- **check_size** (*bool*) – Enable/disable size checking before removal.

Returns
array_like – 2D array. Zinger-removed image.

References

[1] : https://doi.org/10.1364/OE.418448

algotom.prep.removal.**generate_blob_mask**(*flat*, *size*, *snr*)

Generate a binary mask of blobs from a flat-field image (Ref. [1]).

Parameters

- **flat** (*array_like*) – 2D array. Flat-field image.
- **size** (*float*) – Estimated size of the largest blob.
- **snr** (*float*) – Ratio used to segment blobs.

Returns

array_like – 2D array. Binary mask.

References

[1] : https://doi.org/10.1364/OE.418448

algotom.prep.removal.**remove_blob_1d**(*sino_1d*, *mask_1d*)

Remove blobs in one row of a sinogram, e.g. for a helical scan as shown in Ref. [1].

Parameters

- **sino_1d** (*array_like*) – 1D array. A row of a sinogram.
- **mask_1d** (*array_like*) – 1D binary mask.

Returns

array_like – 1D array.

Notes

The method is used to remove streak artifacts caused by blobs in a sinogram generated from a helical-scan data [1].

References

[1] : https://doi.org/10.1364/OE.418448

algotom.prep.removal.**remove_blob**(*mat*, *mask*)

Remove blobs in an image.

Parameters

- **mat** (*array_like*) – 2D array. Projection image or sinogram image.
- **mask** (*array_like*) – 2D binary mask.

Returns

array_like – 2D array.

algotom.prep.phase

Module for phase contrast imaging:

- Unwrap phase images.
- Generate a quality map, weight mask.
- Reconstruct surface from gradient images.
- **Methods for speckle-based phase-contrast imaging.**
  - Find shifts between two stacks of images.
  - Find shifts between sample-images.
  - Align between two stacks of images.
  - Retrieve phase image.
  - Generate transmission-signal and dark-signal images.

Functions:

unwrap_phase_based_cosine_transform(mat[, ...])	Unwrap a phase image using the cosine transform as described in Ref.
unwrap_phase_based_fft(mat[, win_for, win_back])	Unwrap a phase image using the Fourier transform as described in Ref.
unwrap_phase_iterative_fft(mat[, iteration, ...])	Unwrap a phase image using an iterative FFT-based method as described in Ref.
get_quality_map(mat, size)	Generate a quality map using the phase derivative variance (PDV) as described in Ref.
get_weight_mask(mat[, snr])	Generate a binary weight-mask based on a provided quality map.
reconstruct_surface_from_gradient_FC_method(...)	Reconstruct a surface from the gradients in x and y-direction using the Frankot-Chellappa method (Ref.
reconstruct_surface_from_gradient_SCS_method(...)	Reconstruct a surface from the gradients in x and y-direction using the Simchony-Chellappa-Shao method (Ref.
find_shift_between_image_stacks(ref_stack, ...)	Find shifts between each pair of two image-stacks.
find_shift_between_sample_images(ref_stack, ...)	Find shifts between sample-images in a stack against the first sample-image.
align_image_stacks(ref_stack, sam_stack, ...)	Align each pair of two image-stacks using provided reference-sample shifts with an option to correct the shifts between sample-images.
retrieve_phase_based_speckle_tracking(...[, ...])	Retrieve the phase image from two stacks of speckle-images and sample-images where the shift of each pixel is determined using a correlation-based technique (Ref.
get_transmission_dark_field_signal(...[, ...])	Get the transmission-signal image and dark-signal image from two stacks of speckle-images and sample-images for correlation-based methods.

algotom.prep.phase.**get_quality_map**(*mat*, *size*)

Generate a quality map using the phase derivative variance (PDV) as described in Ref. [1].

Parameters

- **mat** (*array_like*) – 2D array.
- **size** (*int*) – Window size. e.g. size=5.

Returns
array_like – 2D array.

References

[1]
[Dennis Ghiglia and Mark Pritt, "Two-dimensional Phase Unwrapping:] Theory, Algorithms, and Software", Wiley, New York,1998.

`algotom.prep.phase.get_weight_mask`(*mat*, *snr=1.5*)

Generate a binary weight-mask based on a provided quality map. Threshold value is calculated based on Algorithm 4 in Ref. [1].

Parameters

- **mat** (*array_like*) – 2D array. e.g. a quality map.
- **snr** (*float*) – Ratio used to calculate the threshold value. Greater is less sensitive.

Returns
array_like – 2D binary array.

References

[1] : https://doi.org/10.1364/OE.26.028396

`algotom.prep.phase.unwrap_phase_based_cosine_transform`(*mat*, *window=None*)

Unwrap a phase image using the cosine transform as described in Ref. [1].

Parameters

- **mat** (*array_like*) – 2D array. Wrapped phase-image in the range of [-Pi; Pi].
- **window** (*array_like*) – 2D array. Window is used for the cosine transform. Generated if None.

Returns
array_like – 2D array. Unwrapped phase-image.

References

[1] : https://doi.org/10.1364/JOSAA.11.000107

`algotom.prep.phase.unwrap_phase_based_fft`(*mat*, *win_for=None*, *win_back=None*)

Unwrap a phase image using the Fourier transform as described in Ref. [1].

Parameters

- **mat** (*array_like*) – 2D array. Wrapped phase-image in the range of [-Pi; Pi].
- **win_for** (*array_like*) – 2D array. FFT-window for the forward transform. Generated if None.
- **win_back** (*array_like*) – 2D array. FFT-window for the backward transform. Making sure there are no zero-values. Generated if None.

Returns
array_like – 2D array. Unwrapped phase-image.

References

[1] : https://doi.org/10.1109/36.297989

algotom.prep.phase.**unwrap_phase_iterative_fft**(*mat*, *iteration=4*, *win_for=None*, *win_back=None*, *weight_map=None*)

Unwrap a phase image using an iterative FFT-based method as described in Ref. [1].

Parameters

- **mat** (*array_like*) – 2D array. Wrapped phase-image in the range of [-Pi; Pi].
- **iteration** (*int*) – Number of iteration.
- **win_for** (*array_like*) – 2D array. FFT-window for the forward transform. Generated if None.
- **win_back** (*array_like*) – 2D array. FFT-window for the backward transform. Making sure there are no zero-values. Generated if None.
- **weight_map** (*array_like*) – 2D array. Using a weight map if provided.

Returns
array_like – 2D array. Unwrapped phase-image.

References

[1] : https://doi.org/10.1364/AO.56.007079

algotom.prep.phase.**reconstruct_surface_from_gradient_FC_method**(*grad_x*, *grad_y*, *correct_negative=True*, *window=None*)

Reconstruct a surface from the gradients in x and y-direction using the Frankot-Chellappa method (Ref. [1]). Note that the DC-component (average value of an image) of the reconstructed image is unidentified because the DC-component of the FFT-window is zero.

Parameters

- **grad_x** (*array_like*) – 2D array. Gradient in x-direction.
- **grad_y** (*array_like*) – 2D array. Gradient in y-direction.
- **correct_negative** (*bool, optional*) – Correct negative offset if True.
- **window** (*list of array_like*) – list of three 2D-arrays. Spatial frequencies in x, y, and the window for the Fourier transform. Generated if None.

Returns
array_like – 2D array. Reconstructed surface.

References

[1] : https://doi.org/10.1109/34.3909

algotom.prep.phase.**reconstruct_surface_from_gradient_SCS_method**(*grad_x*, *grad_y*, *correct_negative=True*, *window=None*, *pad=0*, *pad_mode='linear_ramp'*)

Reconstruct a surface from the gradients in x and y-direction using the Simchony-Chellappa-Shao method (Ref. [1]). Note that the DC-component (average value of an image) of the reconstructed image is unidentified because the DC-component of the FFT-window is zero.

Parameters

- **grad_x** (*array_like*) – 2D array. Gradient in x-direction.
- **grad_y** (*array_like*) – 2D array. Gradient in y-direction.
- **correct_negative** (*bool, optional*) – Correct negative offset if True.
- **window** (*list of array_like*) – List of three 2D-arrays. Spatial frequencies in x, y, and the window for the Fourier transform. Generated if None.
- **pad** (*int*) – Padding width.
- **pad_mode** (*str*) – Padding method. Full list can be found at numpy_pad documentation.

Returns

array_like – 2D array. Reconstructed surface.

References

[1] : https://doi.org/10.1109/34.55103

algotom.prep.phase.**find_shift_between_image_stacks**(*ref_stack*, *sam_stack*, *win_size*, *margin*, *list_ij*, *global_value='mixed'*, *gpu=False*, *block=32*, *sub_pixel=True*, *method='diff'*, *size=3*, *ncore=None*, *norm=False*)

Find shifts between each pair of two image-stacks. Can be used to align reference-images and sample-images in speckle-based imaging technique. The method finds the shift between two images by finding local shifts between small areas of the images given by a list of points.

Parameters

- **ref_stack** (*array_like*) – 3D array. Reference images.
- **sam_stack** (*array_like*) – 3D array. Sample images.
- **win_size** (*int*) – To define the size of the area around a selected pixel of the sample image.
- **margin** (*int*) – To define the size of the area of the reference image for searching, i.e. size = 2 * margin + win_size.
- **list_ij** (*list of lists of int*) – List of indices of points used for local search. Accept the value of [i_index, j_index] for a single point or [[i_index0, i_index1,...], [j_index0, j_index1,...]] for multiple points.
- **global_value** (*{"median", "mean", "mixed"}*) – Method for calculating the global value from local values.
- **gpu** (*bool, optional*) – Use GPU for computing if True.

- **block** (*int*) – Size of a GPU block. E.g. 16, 32, 64, ...
- **sub_pixel** (*bool, optional*) – Enable sub-pixel location.
- **method** (*{"diff", "poly_fit"}*) – Method for finding 1d sub-pixel position. Two options: a differential method or a polynomial method.
- **size** (*int*) – Window size around the integer location of the maximum value used for sub-pixel searching.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **norm** (*bool, optional*) – Normalize the input images if True.

Returns

array_like – List of [[x_shift0, y_shift0], [x_shift1, y_shift1],...]. The shift of each image in the second stacks against each image in the first stack.

algotom.prep.phase.**find_shift_between_sample_images**(*ref_stack*, *sam_stack*, *sr_shifts*, *win_size*, *margin*, *list_ij*, *global_value='median'*, *gpu=False*, *block=32*, *sub_pixel=True*, *method='diff'*, *size=3*, *ncore=None*, *norm=False*)

Find shifts between sample-images in a stack against the first sample-image. It is used to align sample-images of the same rotation-angle from multiple tomographic datasets. Reference-images are used for normalization before finding the shifts.

Parameters

- **ref_stack** (*array_like*) – 3D array. Reference images.
- **sam_stack** (*array_like*) – 3D array. Sample images.
- **sr_shifts** (*array_like*) – List of shifts between each pair of reference-images and sample-images.
- **win_size** (*int*) – To define the size of the area around a selected pixel of the sample image.
- **margin** (*int*) – To define the size of the area of the reference image for searching, i.e. size = 2 * margin + win_size.
- **list_ij** (*list of lists of int*) – List of indices of points used for local search. Accept the value of [i_index, j_index] for a single point or [[i_index0, i_index1,...], [j_index0, j_index1,...]] for multiple points.
- **global_value** (*{"median", "mean", "mixed"}*) – Method for calculating the global value from local values.
- **gpu** (*bool, optional*) – Use GPU for computing if True.
- **block** (*int*) – Size of a GPU block. E.g. 16, 32, 64, ...
- **sub_pixel** (*bool, optional*) – Enable sub-pixel location.
- **method** (*{"diff", "poly_fit"}*) – Method for finding 1d sub-pixel position. Two options: a differential method or a polynomial method.
- **size** (*int*) – Window size around the integer location of the maximum value used for sub-pixel searching.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **norm** (*bool, optional*) – Normalize the input images if True.

Returns

array_like – List of [[0.0, 0.0], [x_shift1, y_shift1],...]. For convenient usage, the shift of the first image in the stack with itself, [0.0, 0.0], is added to the result.

algotom.prep.phase.**align_image_stacks**(*ref_stack*, *sam_stack*, *sr_shifts*, *sam_shifts=None*, *mode='reflect'*)

Align each pair of two image-stacks using provided reference-sample shifts with an option to correct the shifts between sample-images.

Parameters

- **ref_stack** (*array_like*) – 3D array. Reference images.
- **sam_stack** (*array_like*) – 3D array. Sample images.
- **sr_shifts** (*array_like*) – List of shifts between each pair of reference-images and sample-images. Each value is the shift of the second image against the first image.
- **sam_shifts** (*array_like, optional*) – List of shifts between each sample-image and the first sample-image.
- **mode** (*{'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional*) – Method to fill up empty areas caused by shifting the images.

Returns

- **ref_stack** (*array_like*) – 3D array. Aligned reference-images.
- **sam_stack** (*array_like*) – 3D array. Aligned sample-images.

algotom.prep.phase.**get_transmission_dark_field_signal**(*ref_stack*, *sam_stack*, *x_shifts*, *y_shifts*, *win_size*, *margin=None*, *ncore=None*)

Get the transmission-signal image and dark-signal image from two stacks of speckle-images and sample-images for correlation-based methods.

Parameters

- **ref_stack** (*array_like*) – 3D array. Reference images (speckle images).
- **sam_stack** (*array_like*) – 3D array. Sample images.
- **x_shifts** (*array_like*) – x-shift image.
- **y_shifts** (*array_like*) – y-shift image.
- **win_size** (*int*) – Window size used for calculating signals.
- **margin** (*int or None*) – Margin value used for calculating signals.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.

Returns

- **trans** (*array_like*) – Transmission-signal image
- **dark** (*array_like*) – Dark-signal image

algotom.prep.phase.**retrieve_phase_based_speckle_tracking**(*ref_stack*, *sam_stack*, *find_shift='correl'*, *filter_name='hamming'*, *dark_signal=False*, *dim=1*, *win_size=7*, *margin=10*, *method='diff'*, *size=3*, *gpu=False*, *block=(16, 16)*, *ncore=None*, *norm=True*, *norm_global=False*, *chunk_size=100*, *surf_method='SCS'*, *correct_negative=True*, *window=None*, *pad=100*, *pad_mode='linear_ramp'*, *return_shift=False*)

Retrieve the phase image from two stacks of speckle-images and sample-images where the shift of each pixel is determined using a correlation-based technique (Ref. [1-2]) or a cost-function-based method (Ref. [3]). Results can be an image, a list of 3 images, or a list of 5 images.

Parameters

- **ref_stack** (*array_like*) – 3D array. Reference images (speckle images).
- **sam_stack** (*array_like*) – 3D array. Sample images.
- **find_shift** (*{"correl", "umpa"}*) – To select the back-end method for finding shifts. Using a correlation-based method (Ref. [1-2]) or a cost-based method (Ref. [3]).
- **filter_name** (*{None, "hann", "bartlett", "blackman", "hamming", "nuttall", "parzen", "triang"}*) – To select a smoothing filter.
- **dark_signal** (*bool*) – Return both dark-signal image and transmission-signal image if True
- **dim** (*{1, 2}*) – To find the shifts (in x and y) separately (1D) or together (2D).
- **win_size** (*int*) – Size of local areas in the sample image for finding shifts.
- **margin** (*int*) – To define the searching range of the sample images in finding the shifts compared to the reference images.
- **method** (*{"diff", "poly_fit"}*) – Method for finding sub-pixel shift. Two options: a differential method (Ref. [4]) or a polynomial method (Ref. [5]). The "poly_fit" option is not available if using GPU.
- **size** (*int*) – Window size around the integer location of the maximum value used for sub-pixel location. Adjustable if using the polynomial method.
- **gpu** (*{False, True, "hybrid"}*) – Use GPU for computing if True or in "hybrid" mode.
- **block** (*tuple of two integer-values, optional*) – Size of a GPU block. E.g. (8, 8), (16, 16), (32, 32), ...
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **norm** (*bool, optional*) – Normalizing the inputs if True.
- **norm_global** (*bool, optional*) – Normalize by using the full size of the inputs if True.
- **chunk_size** (*int or None*) – Size of each chunk extracted along the height of the image.
- **surf_method** (*{"SCS", "FC"}*) – Select method for surface reconstruction: "SCS" (Ref. [6]) or "FC" (Ref. [7])
- **correct_negative** (*bool, optional*) – Correct negative offset if True.
- **window** (*list of array_like*) – List of three 2D-arrays. Spatial frequencies in x, y, and the window in the Fourier space for the surface reconstruction method. Generated if None.

- **pad** (*int*) – Padding-width used for the "SCS" method.
- **pad_mode** (*str*) – Padding-method used for the "SCS" method. Full list can be found at numpy_pad documentation.
- **return_shift** (*bool, optional*) – Return a list of 3 arrays: x-shifts, y-shifts, and phase image if True. The shifts can be used to determine transmission-signal and dark-signal image.

Returns

- **phase** (*array_like*) – Phase image. If dark_signal is False and return_shifts is False.
- **phase, trans, dark** (*list of array_like*) – Phase image, transmission image, and dark-signal image. If dark_signal is True and return_shifts is False.
- **x_shifts, y_shifts, phase** (*list of array_like*) – x-shift image and y-shift image. If dark_signal is False and return_shifts is True.
- **x_shifts, y_shifts, phase, trans, dark** (*list of array_like*) – x-shift image, y-shift image, phase image, transmission image, and dark-signal image. If dark_signal is True and return_shifts is True.

References

[1] : https://doi.org/10.1038/srep08762

[2] : https://doi.org/10.1103/PhysRevApplied.5.044014

[3] : https://doi.org/10.1103/PhysRevLett.118.203903

[4] : https://doi.org/10.48550/arXiv.0712.4289

[5] : https://doi.org/10.1088/0957-0233/17/6/045

[6] : https://doi.org/10.1109/34.55103

[7] : https://doi.org/10.1109/34.3909

1.7.3 Reconstruction

`algotom.rec.reconstruction`

Module of FFT-based reconstruction methods in the reconstruction stage:

- Filtered back-projection (FBP) method for GPU (using numba and cuda) and CPU.
- Direct Fourier inversion (DFI) method.
- Wrapper for Astra-Toolbox reconstruction methods (optional)
- Wrapper for Tomopy-gridrec reconstruction method (optional)
- Center-of-rotation determination using slice metrics.

Functions:

fbp_reconstruction(sinogram, center[, ...])	Apply the FBP (filtered back-projection) reconstruction method to a sinogram-image or a chunk of sinogram-images.
dfi_reconstruction(sinogram, center[, ...])	Apply the DFI (direct Fourier inversion) reconstruction method (Ref.
gridrec_reconstruction(sinogram, center[, ...])	Apply the gridrec method to a sinogram-image or a chunk of sinogram-images.
astra_reconstruction(sinogram, center[, ...])	Wrapper of reconstruction methods implemented in the astra toolbox package.
find_center_based_slice_metric(sinogram, ...)	Find the center-of-rotation (COR) using metrics of reconstructed slices at different CORs.

algotom.rec.reconstruction.**make_smoothing_window**(*filter_name*, *width*)

Make a 1d smoothing window.

Parameters

- **filter_name** (*{"hann", "bartlett", "blackman", "hamming", "nuttall", "parzen", "triang"}*) – Window function used for filtering.
- **width** (*int*) – Width of the window.

Returns

array_like – 1D array.

algotom.rec.reconstruction.**make_2d_ramp_window**(*height*, *width*, *filter_name=None*)

Make the 2d ramp window (in the Fourier space) by repeating the 1d ramp window with the option of adding a smoothing window.

Parameters

- **height** (*int*) – Height of the window.
- **width** (*int*) – Width of the window.
- **filter_name** (*{None, "hann", "bartlett", "blackman", "hamming", "nuttall", "parzen", "triang"}*) – Name of a smoothing window used.

Returns

complex ndarray – 2D array.

algotom.rec.reconstruction.**apply_ramp_filter**(*sinogram*, *ramp_win=None*, *filter_name=None*, *pad=None*, *pad_mode='edge'*)

Apply the ramp filter to a sinogram with the option of adding a smoothing filter.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **ramp_win** (*complex ndarray or None*) – Ramp window in the Fourier space.
- **filter_name** (*{None, "hann", "bartlett", "blackman", "hamming", "nuttall", "parzen", "triang"}*) – Name of a smoothing window used.
- **pad** (*int or None*) – To apply padding before the FFT. The value is set to 10% of the image width if None is given.
- **pad_mode** (*str*) – Padding method. Full list can be found at numpy_pad documentation.

Returns
array_like – Filtered sinogram.

algotom.rec.reconstruction.**back_projection_gpu**(*recon*, *sinogram*, *angles*, *xlist*, *center*, *sino_height*, *sino_width*)

Implement the back-projection algorithm using GPU.

Parameters

- **recon** (*array_like*) – Square array of zeros. Initialized reconstruction-image.
- **sinogram** (*array_like*) – 2D array. (Filtered) sinogram image.
- **angles** (*array_like*) – 1D array. Angles (radian) corresponding to the sinogram.
- **xlist** (*array_like*) – 1D array. Distances of the integration lines to the image center.
- **center** (*float*) – Center of rotation.
- **sino_height** (*int*) – Height of the sinogram image.
- **sino_width** (*int*) – Width of the sinogram image.

Returns
recon (*array_like*) – Note that this is the GPU kernel function, i.e. no need of "return".

algotom.rec.reconstruction.**back_projection_gpu_chunk**(*recons*, *sinograms*, *angles*, *xlist*, *center*, *sino_height*, *sino_width*, *num_sino*)

Implement the back-projection algorithm for a chunk of sinograms using GPU. Axis of a sinogram/slice in the 3D array is 1.

Parameters

- **recons** (*array_like*) – 3D array of zeros. Initialized reconstruction-images.
- **sinograms** (*array_like*) – 3D array. (Filtered) sinogram images.
- **angles** (*array_like*) – 1D array. Angles (radian) corresponding to a sinogram.
- **xlist** (*array_like*) – 1D array. Distances of the integration lines to the image center.
- **center** (*float*) – Center of rotation.
- **sino_height** (*int*) – Height of the sinogram image.
- **sino_width** (*int*) – Width of the sinogram image.
- **num_sino** (*int*) – Number of sinograms.

Returns
recons (*array_like*) – Reconstructed images.

algotom.rec.reconstruction.**back_projection_cpu**(*sinogram*, *angles*, *xlist*, *center*)

Implement the back-projection algorithm using CPU.

Parameters

- **sinogram** (*array_like*) – 2D array. (Filtered) sinogram image.
- **angles** (*array_like*) – 1D array. Angles (radian) corresponding to the sinogram.
- **xlist** (*array_like*) – 1D array. Distances of the integration lines to the image center.
- **center** (*float*) – Center of rotation.

Returns
recon (*array_like*) – Square array. Reconstructed image.

algotom.rec.reconstruction.**fbp_reconstruction**(*sinogram*, *center*, *angles=None*, *ratio=1.0*, *ramp_win=None*, *filter_name='hann'*, *pad=None*, *pad_mode='edge'*, *apply_log=True*, *gpu=True*, *block=(16, 16)*, *ncore=None*)

Apply the FBP (filtered back-projection) reconstruction method to a sinogram-image or a chunk of sinogram-images. Angular axis is 0. If input is 3D array, the slicing axis of sinograms must be 1, e.g. data[:, index, :].

Parameters

- **sinogram** (*array_like*) – 2D/3D array. Sinogram image.
- **center** (*float*) – Center of rotation.
- **angles** (*array_like, optional*) – 1D array. List of angles (in radian) corresponding to the sinogram.
- **ratio** (*float, optional*) – To apply a circle mask to the reconstructed image.
- **ramp_win** (*complex ndarray, optional*) – Ramp window in the Fourier space. Generated if None.
- **filter_name** (*{None, "hann", "bartlett", "blackman", "hamming", "nuttall", "parzen", "triang"}*) – Apply a smoothing filter.
- **pad** (*int, optional*) – To apply padding before the FFT. The value is set to 10% of the image width if None is given.
- **pad_mode** (*str, optional*) – Padding method. Full list can be found at numpy_pad documentation.
- **apply_log** (*bool, optional*) – Apply the logarithm function to the sinogram before reconstruction.
- **gpu** (*bool, optional*) – Use GPU for computing if True.
- **block** (*tuple of two integer-values, optional*) – Size of a GPU block. E.g. (8, 8), (16, 16), (32, 32), ...
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.

Returns

array_like – Square array. Reconstructed image.

algotom.rec.reconstruction.**generate_mapping_coordinate**(*width_sino*, *height_sino*, *width_rec*, *height_rec*)

Calculate coordinates in the sinogram space from coordinates in the reconstruction space (in the Fourier domain). They are used for the DFI (direct Fourier inversion) reconstruction method.

Parameters

- **width_sino** (*int*) – Width of a sinogram image.
- **height_sino** (*int*) – Height of a sinogram image.
- **width_rec** (*int*) – Width of a reconstruction image.
- **height_rec** (*int*) – Height of a reconstruction image.

Returns

- **r_mat** (*array_like*) – 2D array. Broadcast of the r-coordinates.
- **theta_mat** (*array_like*) – 2D array. Broadcast of the theta-coordinates.

algotom.rec.reconstruction.**dfi_reconstruction**(*sinogram*, *center*, *angles=None*, *ratio=1.0*, *filter_name='hann'*, *pad_rate=0.25*, *pad_mode='edge'*, *apply_log=True*, *ncore=None*)

Apply the DFI (direct Fourier inversion) reconstruction method (Ref. [1]) to a sinogram-image or a chunk of sinogram-images. Angular axis is 0. If input is 3D array, the slicing axis of sinograms must be 1, e.g. data[:, index, :].

Parameters

- **sinogram** (*array_like*) – 2D/3D array. Sinogram image.
- **center** (*float*) – Center of rotation.
- **angles** (*array_like*) – 1D array. List of angles (in radian) corresponding to the sinogram.
- **ratio** (*float*) – To apply a circle mask to the reconstructed image.
- **filter_name** (*{None, "hann", "bartlett", "blackman", "hamming", "nuttall", "parzen", "triang"}*) – Apply a smoothing filter.
- **pad_rate** (*float*) – To apply padding before the FFT. The padding width equals to (pad_rate * image_width).
- **pad_mode** (*str*) – Padding method. Full list can be found at numpy_pad documentation.
- **apply_log** (*bool*) – Apply the logarithm function to the sinogram before reconstruction.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.

Returns

array_like – Square array. Reconstructed image.

References

[1] : https://doi.org/10.1071/PH560198

algotom.rec.reconstruction.**gridrec_reconstruction**(*sinogram*, *center*, *angles=None*, *ratio=1.0*, *filter_name='shepp'*, *apply_log=True*, *pad=100*, *filter_par=0.9*, *ncore=1*)

Apply the gridrec method to a sinogram-image or a chunk of sinogram-images. Angular axis is 0. If input is 3D array, the slicing axis of sinograms must be 1, e.g. data[:, index, :]. This is the wrapper of the gridrec method implemented in the Tomopy package: https://tomopy.readthedocs.io/en/latest/api/tomopy.recon.algorithm.html. Users must install Tomopy before using this function.

Parameters

- **sinogram** (*array_like*) – 2D/3D array. Sinogram image.
- **center** (*float*) – Center of rotation.
- **angles** (*array_like*) – 1D array. List of angles (radian) corresponding to the sinogram.
- **ratio** (*float*) – To apply a circle mask to the reconstructed image.
- **filter_name** (*str or None*) – Apply a smoothing filter. Full list is at: https://github.com/tomopy/tomopy/blob/master/source/tomopy/recon/algorithm.py
- **apply_log** (*bool*) – Apply the logarithm function to the sinogram before reconstruction.
- **pad** (*bool or int*) – Apply edge padding to the nearest power of 2.

- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.

Returns
array_like – Square array.

algotom.rec.reconstruction.**astra_reconstruction**(*sinogram*, *center*, *angles=None*, *ratio=1.0*, *method='FBP_CUDA'*, *num_iter=1*, *filter_name='hann'*, *pad=None*, *apply_log=True*, *ncore=1*)

Wrapper of reconstruction methods implemented in the astra toolbox package. https://www.astra-toolbox.com/docs/algs/index.html Users must install Astra Toolbox before using this function. Apply the method to a sinogram-image or a chunk of sinogram-images. Angular axis is 0. If input is 3D array, the slicing axis of sinograms must be 1, e.g. data[:, index, :]

Parameters

- **sinogram** (*array_like*) – 2D/3D array. Sinogram image.
- **center** (*float*) – Center of rotation.
- **angles** (*array_like*) – 1D array. List of angles (radian) corresponding to the sinogram.
- **ratio** (*float*) – To apply a circle mask to the reconstructed image.
- **method** (*str*) – Reconstruction algorithms. For CPU: 'FBP', 'SIRT', 'SART', 'ART', and 'CGLS'. For GPU: 'FBP_CUDA', 'SIRT_CUDA', 'SART_CUDA', and 'CGLS_CUDA'.
- **num_iter** (*int*) – Number of iterations if using iteration methods.
- **filter_name** (*str or None*) – Apply filter if using FBP method. Options: 'ram-lak', 'hamming', 'hann', 'lanczos', 'kaiser', 'parzen',...
- **pad** (*int*) – Padding to reduce the side effect of FFT.
- **apply_log** (*bool*) – Apply the logarithm function to the sinogram before reconstruction.

Returns
array_like – Square array.

algotom.rec.reconstruction.**find_center_based_slice_metric**(*sinogram*, *start*, *stop*, *step=0.5*, *radius=2*, *zoom=0.5*, *method='dfi'*, *gpu=False*, *angles=None*, *ratio=1.0*, *filter_name='hann'*, *apply_log=False*, *ncore=None*, *sigma=3*, *metric_function=None*, ***kwargs*)

Find the center-of-rotation (COR) using metrics of reconstructed slices at different CORs. The entropy of histogram (Ref. [1]) is used by default if the metric-function is set to None. If customized metrics are used, the minimum value must be corresponding to the best center.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **start** (*float*) – Starting point for searching CoR.
- **stop** (*float*) – Ending point for searching CoR.
- **step** (*float*) – Sub-pixel searching step.
- **radius** (*float*) – Searching range with the sub-pixel step.
- **zoom** (*float*) – To resize the sinogram for fast coarse-searching. For example, 0.5 <=> reduce the size of the image by half.

- **method** (*{"dfi", "gridrec", "fbp", "astra"}*) – To select a backend method for reconstruction.
- **gpu** (*bool, optional*) – Use GPU for computing if True.
- **angles** (*array_like, optional*) – 1D array. List of angles (in radian) corresponding to the sinogram.
- **ratio** (*float, optional*) – To apply a circle mask to the reconstructed image.
- **filter_name** (*{None, "hann", "bartlett", "blackman", "hamming", "nuttall", "parzen", "triang"}*) – Apply a smoothing filter before reconstruction.
- **apply_log** (*bool, optional*) – Apply the logarithm function to the sinogram before reconstruction.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **sigma** (*int*) – Denoising the sinogram before reconstruction. Should be set to 0 for noise-free data (simulation).
- **metric_function** (*obj*) – To apply a customized function for calculating metric going with keyword arguments.

Returns

float – Center-of-rotation.

References

[1] : https://doi.org/10.1364/JOSAA.23.001048

1.7.4 Post-processing

`algotom.post.postprocessing`

Module of methods in the postprocessing stage:

- Get statistical information of reconstructed images or a dataset.
- Downsample 2D, 3D array, or a dataset.
- Rescale 2D, 3D array or a dataset to 8-bit or 16-bit data-type.
- Reslice 3D array or a dataset (hdf/nxs file or tif images).
- Removing ring artifacts in a reconstructed image by transform back and forth between the polar coordinates and the Cartesian coordinates.

Functions:

get_statistical_information(mat[, ...])	Get statistical information of an image.
get_statistical_information_dataset(input_)	Get statical information of a dataset.
downsample(mat, cell_size[, method])	Downsample an image.
downsample_dataset(input_, output, cell_size)	Downsample a dataset.
rescale(mat[, nbit, minmax])	Rescale a 32-bit array to 16-bit/8-bit data.
rescale_dataset(input_, output[, nbit, ...])	Rescale a dataset to 8-bit or 16-bit data-type.
reslice_dataset(input_, output[, axis, ...])	Reslice a 3d dataset.
remove_ring_based_fft(mat[, u, n, v, sort])	Remove ring artifacts in the reconstructed image by combining the polar transform and the fft-based method.
remove_ring_based_wavelet_fft(mat[, level, ...])	Remove ring artifacts in a reconstructed image by combining the polar transform and the wavelet-fft-based method (Ref.

algotom.post.postprocessing.**get_statistical_information**(*mat*, *percentile=(0, 100)*, *denoise=False*)

Get statistical information of an image.

Parameters

- **mat** (*array_like*) – 2D array. Projection image, sinogram image, or reconstructed image.
- **percentile** (*tuple of floats*) – Tuple of (min_percentile, max_percentile) to compute. Must be between 0 and 100 inclusive.
- **denoise** (*bool, optional*) – Enable/disable denoising before extracting statistical information.

Returns

- **gmin** (*float*) – The minimum value of the data array.
- **gmax** (*float*) – The maximum value of the data array.
- **min_percent** (*float*) – The first computed percentile of the data array.
- **max_percent** (*tuple of floats*) – The last computed percentile of the data array.
- **mean** (*float*) – The mean of the data array.
- **median** (*float*) – The median of the data array.
- **variance** (*float*) – The variance of the data array.

algotom.post.postprocessing.**get_statistical_information_dataset**(*input_*, *percentile=(0, 100)*, *skip=5*, *denoise=False*, *key_path=None*, *crop=(0, 0, 0, 0, 0, 0)*)

Get statical information of a dataset. This can be a folder of tif files, a hdf file, or a 3D array.

Parameters

- **input_** (*str, hdf file, or array_like*) – It can be a folder path to tif files, a hdf file, or a 3D array.
- **percentile** (*tuple of floats*) – Tuple of (min_percentile, max_percentile) to compute. Must be between 0 and 100 inclusive.
- **skip** (*int*) – Skipping step of reading input.
- **denoise** (*bool, optional*) – Enable/disable denoising before extracting statistical information.
- **key_path** (*str, optional*) – Key path to the dataset if input is a hdf file.

- **crop** (*tuple of int, optional*) – Crop 3D data from the edges, i.e. crop = (crop_depth1, crop_depth2, crop_height1, crop_height2, crop_width1, crop_width2).

Returns

- **gmin** (*float*) – The global minimum value of the data array.
- **gmax** (*float*) – The global maximum value of the data array.
- **min_percent** (*float*) – The global min of the first computed percentile of the data array.
- **max_percent** (*tuple of floats*) – The global min of the last computed percentile of the data array.
- **mean** (*float*) – The mean of the data array.
- **median** (*float*) – The median of the data array.
- **variance** (*float*) – The mean of the variance of the data array.

algotom.post.postprocessing.**downsample**(*mat*, *cell_size*, *method='mean'*)

Downsample an image.

Parameters

- **mat** (*array_like*) – 2D array.
- **cell_size** (*int or tuple of int*) – Window size along axes used for grouping pixels.
- **method** (*{"mean", "median", "max", "min"}*) – Downsampling method.

Returns

array_like – Downsampled image.

algotom.post.postprocessing.**rescale**(*mat*, *nbit=16*, *minmax=None*)

Rescale a 32-bit array to 16-bit/8-bit data.

Parameters

- **mat** (*array_like*)
- **nbit** (*{8,16}*) – Rescaled data-type: 8-bit or 16-bit.
- **minmax** (*tuple of float, or None*) – Minimum and maximum values used for rescaling.

Returns

array_like – Rescaled array.

algotom.post.postprocessing.**downsample_dataset**(*input_*, *output*, *cell_size*, *method='mean'*, *key_path=None*, *rescaling=False*, *nbit=16*, *minmax=None*, *skip=None*, *crop=(0, 0, 0, 0, 0, 0)*, *overwrite=False*)

Downsample a dataset. Input can be a folder of tif files, a hdf file, or a 3D array.

Parameters

- **input_** (*str, array_like*) – It can be a folder path to tif files, a hdf file, or a 3D array.
- **output** (*str, None*) – It can be a folder path, a hdf file path, or None (memory consuming).
- **cell_size** (*int or tuple of int*) – Window size along axes used for grouping pixels.
- **method** (*{"mean", "median", "max", "min"}*) – Downsampling method.
- **key_path** (*str, optional*) – Key path to the dataset if the input is a hdf file.
- **rescaling** (*bool*) – Rescale dataset if True.

- **nbit** (*{8,16}*) – If rescaling is True, select data-type: 8-bit or 16-bit.
- **minmax** (*tuple of float, or None*) – Minimum and maximum values used for rescaling if True.
- **skip** (*int or None*) – Skipping step of images used for getting statistical information if rescaling is True and input is 32-bit data.
- **crop** (*tuple of int, optional*) – Crop 3D data from the edges, i.e. crop = (crop_depth1, crop_depth2, crop_height1, crop_height2, crop_width1, crop_width2).
- **overwrite** (*bool*) – Overwrite an existing file/folder if True.

Returns

array_like or None – If output is None, returning a 3D array.

algotom.post.postprocessing.**rescale_dataset**(*input_*, *output*, *nbit=16*, *minmax=None*, *skip=None*, *key_path=None*, *crop=(0, 0, 0, 0, 0, 0)*, *overwrite=False*)

Rescale a dataset to 8-bit or 16-bit data-type. The dataset can be a folder of tif files, a hdf file, or a 3D array.

Parameters

- **input_** (*str, array_like*) – It can be a folder path to tif files, a hdf file, or 3D array.
- **output** (*str, None*) – It can be a folder path, a hdf file path, or None (memory consuming).
- **nbit** (*{8,16,32}*) – Select rescaled data-type: 8-bit/16-bit. 32 is for cropping data only.
- **minmax** (*tuple of float, or None*) – Minimum and maximum values used for rescaling. They are calculated if None is given.
- **skip** (*int or None*) – Skipping step of images used for getting statistical information.
- **key_path** (*str, optional*) – Key path to the dataset if the input is a hdf file.
- **crop** (*tuple of int, optional*) – Crop 3D data from the edges, i.e. crop = (crop_depth1, crop_depth2, crop_height1, crop_height2, crop_width1, crop_width2).
- **overwrite** (*bool*) – Overwrite an existing file/folder if True.

Returns

array_like or None – If output is None, returning an 3D array.

algotom.post.postprocessing.**reslice_dataset**(*input_*, *output*, *axis=1*, *key_path=None*, *rescaling=False*, *nbit=16*, *minmax=None*, *skip=None*, *rotate=0.0*, *chunk=16*, *mode='constant'*, *crop=(0, 0, 0, 0, 0, 0)*, *ncore=None*, *show_progress=True*, *overwrite=False*)

Reslice a 3d dataset. Input can be a folder of tif files or a hdf file.

Parameters

- **input_** (*str, array_like*) – It can be a folder path to tif files or a hdf file.
- **output** (*str*) – It can be a folder path (for generated tif-files) or a hdf file-path.
- **axis** (*{1,2}*) – Slicing axis. This axis becomes the 0-axis of the output.
- **key_path** (*str, optional*) – Key path to the dataset if the input is a hdf file.
- **rescaling** (*bool*) – Rescale dataset if True.
- **nbit** (*{8,16}*) – If rescaling is True, select data-type: 8-bit or 16-bit.
- **minmax** (*tuple of float, or None*) – Minimum and maximum values used for rescaling if True.

- **skip** (*int or None*) – Skipping step of images used for getting statistical information if rescaling is True and input is 32-bit data.
- **rotate** (*float*) – Rotate image (degree). Positive direction is counterclockwise.
- **chunk** (*int*) – Number of images to be loaded/saved in one go to reduce IO overhead.
- **mode** (*{'reflect', 'grid-mirror', 'constant', 'grid-constant', 'nearest', 'mirror', 'grid-wrap', 'wrap'}*) – Select how the input array is extended beyond its boundaries.
- **crop** (*tuple of int, optional*) – Crop 3D data from the edges, i.e. crop = (crop_depth1, crop_depth2, crop_height1, crop_height2, crop_width1, crop_width2). Cropping is done before reslicing.
- **ncore** (*int or None*) – Number of cpu-cores. Automatically selected if None.
- **show_progress** (*bool*) – Show the progress of reslicing data if True.
- **overwrite** (*bool*) – Overwrite an existing file/folder if True.

Returns

array_like or None – If output is None, returning a 3D array.

`algotom.post.postprocessing.`**`remove_ring_based_fft`**(*mat*, *u=20*, *n=8*, *v=1*, *sort=False*)

Remove ring artifacts in the reconstructed image by combining the polar transform and the fft-based method.

Parameters

- **mat** (*array_like*) – Square array. Reconstructed image
- **u** (*int*) – Cutoff frequency.
- **n** (*int*) – Filter order.
- **v** (*int*) – Number of rows (* 2) to be applied the filter.
- **sort** (*bool, optional*) – Apply sorting (Ref. [2]) if True.

Returns

array_like – Ring-removed image.

References

[1] : https://doi.org/10.1063/1.1149043

[2] : https://doi.org/10.1364/OE.26.028396

`algotom.post.postprocessing.`**`remove_ring_based_wavelet_fft`**(*mat*, *level=5*, *size=1*, *wavelet_name='db9'*, *sort=False*)

Remove ring artifacts in a reconstructed image by combining the polar transform and the wavelet-fft-based method (Ref. [1]).

Parameters

- **mat** (*array_like*) – Square array. Reconstructed image
- **level** (*int*) – Wavelet decomposition level.
- **size** (*int*) – Damping parameter. Larger is stronger.
- **wavelet_name** (*str*) – Name of a wavelet. Search pywavelets API for a full list.
- **sort** (*bool, optional*) – Apply sorting (Ref. [2]) if True.

Returns
array_like – Ring-removed image.

References

[1] : https://doi.org/10.1364/OE.17.008567

[2] : https://doi.org/10.1364/OE.26.028396

1.7.5 Utilities

algotom.util.calibration

Module of calibration methods:

- Correcting the non-uniform background of an image.
- Binarizing an image.
- Calculating the distance between two point-like objects segmented from two images. Useful for determining pixel-size in helical scans.
- Find the tilt and roll of a parallel-beam tomography system given coordinates of a point-like object scanned in the range of [0, 360] degrees.

Functions:

normalize_background(mat[, size])	Correct a non-uniform background of an image using the median filter.
normalize_background_based_fft(mat[, sigma, ...])	Correct a non-uniform background of an image using a Fourier Gaussian filter.
invert_dot_contrast(mat)	Invert the contrast of a 2D binary array to make sure that a dot is white.
calculate_threshold(mat[, bgr])	Calculate threshold value based on Algorithm 4 in Ref.
binarize_image(mat[, threshold, bgr, norm, ...])	Binarize an image.
get_dot_size(mat[, size_opt])	Get size of binary dots given the option.
check_dot_size(mat, min_size, max_size)	Check if the size of a dot is in a range.
select_dot_based_size(mat, dot_size[, ratio])	Select dots having a certain size.
calculate_distance(mat1, mat2[, size_opt, ...])	Calculate the distance between two point-like objects segmented from two images.
fit_points_to_ellipse(x, y)	Fit an ellipse to a set of points.
find_tilt_roll(x, y[, method])	Find the tilt and roll of a parallel-beam tomography system given coordinates of a point-like object scanned in the range of [0, 360] degrees.

algotom.util.calibration.**normalize_background**(*mat*, *size=51*)

Correct a non-uniform background of an image using the median filter.

Parameters

- **mat** (*array_like*) – 2D array.
- **size** (*int*) – Size of the median filter.

Returns
array_like – 2D array. Corrected image.

algotom.util.calibration.**normalize_background_based_fft**(*mat*, *sigma=5*, *pad=None*, *mode='reflect'*)

Correct a non-uniform background of an image using a Fourier Gaussian filter.

Parameters

- **mat** (*array_like*) – 2D array.
- **sigma** (*int*) – Sigma of the Gaussian.
- **pad** (*int*) – Padding for the Fourier transform.
- **mode** (*str, list of str, or tuple of str*) – Padding method. One of options : 'reflect', 'edge', 'constant'. Full list is at: https://numpy.org/doc/stable/reference/generated/numpy.pad.html

Returns
array_like – 2D array. Corrected image.

algotom.util.calibration.**invert_dot_contrast**(*mat*)

Invert the contrast of a 2D binary array to make sure that a dot is white.

Parameters
mat (*array_like*) – 2D binary array.

Returns
array_like – 2D array.

algotom.util.calibration.**calculate_threshold**(*mat*, *bgr='bright'*)

Calculate threshold value based on Algorithm 4 in Ref. [1].

Parameters

- **mat** (*array_like*) – 2D array.
- **bgr** (*{"bright", "dark"}*) – To indicate the brightness of the background against image features.

Returns
float – Threshold value.

References

[1] : https://doi.org/10.1364/OE.26.028396

algotom.util.calibration.**binarize_image**(*mat*, *threshold=None*, *bgr='bright'*, *norm=False*, *denoise=True*, *invert=True*)

Binarize an image.

Parameters

- **mat** (*array_like*) – 2D array.
- **threshold** (*float, optional*) – Threshold value for binarization. Automatically calculated using Algorithm 4 in Ref. [1] if None.
- **bgr** (*{"bright", "dark"}*) – To indicate the brightness of the background against image features.
- **norm** (*bool, optional*) – Apply normalization if True.
- **denoise** (*bool, optional*) – Apply denoising if True.

- **invert** (*bool, optional*) – Invert the contrast if needed.

Returns
array_like – 2D binary array.

References

[1] : https://doi.org/10.1364/OE.26.028396

algotom.util.calibration.**get_dot_size**(*mat*, *size_opt='max'*)

Get size of binary dots given the option.

Parameters

- **mat** (*array_like*) – 2D binary array.
- **size_opt** (*{"max", "min", "median", "mean"}*) – Select options.

Returns
dot_size (*float*) – Size of the dot.

algotom.util.calibration.**check_dot_size**(*mat*, *min_size*, *max_size*)

Check if the size of a dot is in a range.

Parameters

- **mat** (*array_like*) – 2D array.
- **min_size** (*float*) – Minimum size.
- **max_size** (*float*) – Maximum size.

Returns
bool

algotom.util.calibration.**select_dot_based_size**(*mat*, *dot_size*, *ratio=0.01*)

Select dots having a certain size.

Parameters

- **mat** (*array_like*) – 2D array.
- **dot_size** (*float*) – Size of the standard dot.
- **ratio** (*float*) – Used to calculate the acceptable range. [dot_size - ratio*dot_size; dot_size + ratio*dot_size]

Returns
array_like – 2D array. Selected dots.

algotom.util.calibration.**calculate_distance**(*mat1*, *mat2*, *size_opt='max'*, *threshold=None*, *bgr='bright'*, *norm=False*, *denoise=True*, *invert=True*)

Calculate the distance between two point-like objects segmented from two images. Useful for measuring pixel-size in helical scans (Ref. [1]).

Parameters

- **mat1** (*array_like*) – 2D array.
- **mat2** (*array_like*) – 2D array.
- **size_opt** (*{"max", "min", "median", "mean"}*) – Options to select binary objects based on their size.

- **threshold** (*float, optional*) – Threshold value for binarization. Automatically calculated using Algorithm 4 in Ref. [2] if None.
- **bgr** (*{"bright", "dark"}*) – To indicate the brightness of the background against image features.
- **norm** (*bool, optional*) – Apply normalization if True.
- **denoise** (*bool, optional*) – Apply denoising if True.
- **invert** (*bool, optional*) – Invert the contrast if needed.

References

[1] : https://doi.org/10.1364/OE.418448

[2] : https://doi.org/10.1364/OE.26.028396

`algotom.util.calibration.`**`fit_points_to_ellipse`**(*x*, *y*)

Fit an ellipse to a set of points.

Parameters

- **x** (*ndarray*) – x-coordinates of the points.
- **y** (*ndarray*) – y-coordinates of the points.

Returns

- **roll_angle** (*float*) – Rotation angle of the ellipse in degree.
- **a_major** (*float*) – Length of the major axis.
- **b_minor** (*float*) – Length of the minor axis.
- **xc** (*float*) – x-coordinate of the ellipse center.
- **yc** (*float*) – y-coordinate of the ellipse center.

`algotom.util.calibration.`**`find_tilt_roll_based_linear_fit`**(*x*, *y*)

Find the tilt and roll of a parallel-beam tomography system given coordinates of a point-like object scanned in the range of [0, 360] degrees. Uses a linear-fit-based approach [1].

Parameters

- **x** (*ndarray*) – x-coordinates of the points.
- **y** (*ndarray*) – y-coordinates of the points.

Returns

- **tilt** (*float*) – Tilt angle in degree.
- **roll** (*float*) – Roll angle in degree.

References

[1] : https://doi.org/10.1098/rsta.2014.0398

algotom.util.calibration.**find_tilt_roll_based_ellipse_fit**(*x*, *y*)

Find the tilt and roll of a parallel-beam tomography system given coordinates of a point-like object scanned in the range of [0, 360] degrees. Uses an ellipse-fit-based approach.

Parameters

- **x** (*ndarray*) – x-coordinates of the points.
- **y** (*ndarray*) – y-coordinates of the points.

Returns

- **tilt** (*float*) – Tilt angle in degree.
- **roll** (*float*) – Roll angle in degree.

algotom.util.calibration.**find_tilt_roll**(*x*, *y*, *method='ellipse'*)

Find the tilt and roll of a parallel-beam tomography system given coordinates of a point-like object scanned in the range of [0, 360] degrees.

Parameters

- **x** (*ndarray*) – x-coordinates of the points.
- **y** (*ndarray*) – y-coordinates of the points.
- **method** (*{"linear", "ellipse"}*) – Method for finding tilt and roll.

Returns

- **tilt** (*float*) – Tilt angle in degree.
- **roll** (*float*) – Roll angle in degree.

algotom.util.simulation

Module of simulation methods:

- Methods for designing a customized 2D phantom.
- Method for calculating a sinogram of a phantom based on the Fourier slice theorem.
- Methods for adding artifacts to a simulated sinogram.

Functions:

make_elliptic_mask(size, center, length, angle)	Create an elliptic mask.
make_rectangular_mask(size, center, length, ...)	Create a rectangular mask.
make_triangular_mask(size, center, length, angle)	Create an isosceles triangle mask.
make_line_target(size)	Create line patterns for testing the resolution of a reconstructed image.
make_face_phantom(size)	Create a face phantom for testing reconstruction methods.
make_sinogram(mat, angles[, pad_rate, pad_mode])	Create a sinogram (series of 1D projections) from a 2D image based on the Fourier slice theorem (Ref.
add_noise(mat[, noise_ratio])	Add Gaussian noise to an image.
add_stripe_artifact(sinogram, size, position)	Add stripe artifacts to a sinogram.
convert_to_Xray_image(sinogram[, global_max])	Convert a simulated sinogram to an equivalent X-ray image.
add_background_fluctuation(sinogram[, ...])	Fluctuate the background of a sinogram image using a Gaussian profile beam.

algotom.util.simulation.**make_elliptic_mask**(*size*, *center*, *length*, *angle*)

Create an elliptic mask.

Parameters

- **size** (*int*) – Size of a square array.
- **center** (*float or tuple of float*) – Ellipse center.
- **length** (*float or tuple of float*) – Lengths of ellipse axes.
- **angle** (*float*) – Rotation angle (Degree) of the ellipse.

Returns

array_like – Square array.

algotom.util.simulation.**make_rectangular_mask**(*size*, *center*, *length*, *angle*)

Create a rectangular mask.

Parameters

- **size** (*int*) – Size of a square array.
- **center** (*float or tuple of float*) – Center of the mask.
- **length** (*float or tuple of float*) – Lengths of the rectangular mask.
- **angle** (*float*) – Rotation angle (Degree) of the mask.

Returns

array_like – Square array.

algotom.util.simulation.**make_triangular_mask**(*size*, *center*, *length*, *angle*)

Create an isosceles triangle mask.

Parameters

- **size** (*int*) – Size of a square array.
- **center** (*float or tuple of float*) – Center of the mask.
- **length** (*float or tuple of float*) – Lengths of the mask.

- **angle** (*float*) – Rotation angle (Degree) of the mask.

Returns
array_like – Square array.

algotom.util.simulation.**make_line_target**(*size*)

Create line patterns for testing the resolution of a reconstructed image.

Parameters
size (*int*) – Size of a square array.

Returns
array_like – Square array.

algotom.util.simulation.**make_face_phantom**(*size*)

Create a face phantom for testing reconstruction methods.

Parameters
size (*int*) – Size of a square array.

Returns
array_like – Square array.

algotom.util.simulation.**make_sinogram**(*mat*, *angles*, *pad_rate=0.5*, *pad_mode='edge'*)

Create a sinogram (series of 1D projections) from a 2D image based on the Fourier slice theorem (Ref. [1]).

Parameters

- **mat** (*array_like*) – Square array.
- **angles** (*array_like*) – 1D array. List of angles (in radian) for projecting.
- **pad_rate** (*float*) – To apply padding before the FFT. The padding width equals to (pad_rate * image_width).
- **pad_mode** (*str*) – Padding method. Full list can be found at numpy_pad documentation.

References

[1] : https://doi.org/10.1071/PH560198

algotom.util.simulation.**add_noise**(*mat*, *noise_ratio=0.1*)

Add Gaussian noise to an image.

Parameters

- **mat** (*array_like*) – 2D array
- **noise_ratio** (*float*) – Ratio between the noise level and the mean of the array.

Returns
array_like

algotom.util.simulation.**add_stripe_artifact**(*sinogram*, *size*, *position*, *strength_ratio=0.2*, *stripe_type='partial'*)

Add stripe artifacts to a sinogram.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **size** (*int*) – Size of stripe artifact.

- **position** (*int*) – Position of the stripe.
- **strength_ratio** (*float*) – To define the strength of the artifact. The value is in the range of [0.0, 1.0].
- **stripe_type** (*{"partial", "full", "dead", "fluctuating"}*) – Type of stripe as classified in Ref. [1].

Returns
array_like

References

[1] : https://doi.org/10.1364/OE.26.028396

algotom.util.simulation.**convert_to_Xray_image**(*sinogram*, *global_max=None*)

Convert a simulated sinogram to an equivalent X-ray image.

Parameters

- **sinogram** (*array_like*) – 2D array.
- **global_max** (*float*) – Maximum value used for normalizing array values to stay in the range of [0.0, 1.0].

Returns
array_like

algotom.util.simulation.**add_background_fluctuation**(*sinogram*, *strength_ratio=0.2*)

Fluctuate the background of a sinogram image using a Gaussian profile beam.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **strength_ratio** (*float*) – To define the strength of the variation. The value is in the range of [0.0, 1.0].

Returns
array_like

algotom.util.utility

Module of utility methods:

- Methods for parallel computing, geometric transformation, masking.
- **Methods for customizing stripe/ring removal methods**
  - sort_forward
  - sort_backward
  - separate_frequency_component
  - generate_fitted_image
  - detect_stripe
  - calculate_regularization_coefficient
  - make_2d_butterworth_window
  - make_2d_damping_window

  - apply_wavelet_decomposition
  - apply_wavelet_reconstruction
  - apply_filter_to_wavelet_component
  - interpolate_inside_stripe
  - transform_slice_forward
  - transform_slice_backward
- **Customized smoothing filters:**
  - apply_gaussian_filter (in the Fourier space)
  - apply_regularization_filter
- **Methods for grid scans:**
  - detect_sample
  - fix_non_sample_areas
  - locate_slice
  - locate_slice_chunk
- **Methods for speckle-based tomography**
  - generate_spiral_positions
- **Methods for finding the center of rotation by visual inspection.**
  - find_center_visual_sinograms
  - find_center_visual_slices

Functions:

apply_method_to_multiple_sinograms(data, ...)	Apply a processing method (in "filtering", "removal", and "reconstruction" module) to multiple sinograms or multiple slices in parallel.
mapping(mat, x_mat, y_mat[, order, mode])	Apply a geometric transformation to a 2D array
make_circle_mask(width, ratio)	Create a circle mask.
sort_forward(mat[, axis])	Sort gray-scales of an image along an axis.
sort_backward(mat, mat_index[, axis])	Sort gray-scales of an image using an index array provided.
separate_frequency_component(mat[, axis, window])	Separate low and high frequency components of an image along an axis.
generate_fitted_image(mat, order[, axis, ...])	Apply a polynomial fitting along an axis of an image.
detect_stripe(list_data, snr)	Locate stripe positions using Algorithm 4 in Ref.
calculate_regularization_coefficient(width, ...)	Calculate coefficients used for the regularization-based method.
make_2d_butterworth_window(width, height, u, ...)	Create a 2d window from the 1D Butterworth window.
make_2d_damping_window(width, height, size)	Make 2D damping window from a list of 1D window for a Fourier-space filter, i.e. a high-pass filter.
apply_wavelet_decomposition(mat, wavelet_name)	Apply 2D wavelet decomposition.
apply_wavelet_reconstruction(data, wavelet_name)	Apply 2D wavelet reconstruction.
apply_filter_to_wavelet_component(data[, ...])	Apply a filter to a component of the wavelet decomposition of an image.
interpolate_inside_stripe(mat, list_mask[, kind])	Interpolate gray-scales inside vertical stripes of an image.
rectangular_from_polar(width_reg, ...)	Generate coordinates of a rectangular grid from polar coordinates.
polar_from_rectangular(width_pol, ...)	Generate polar coordinates from grid coordinates.
transform_slice_forward(mat[, coord_mat])	Transform a reconstructed image into polar coordinates.
transform_slice_backward(mat[, coord_mat])	Transform a reconstructed image in polar coordinates back to rectangular coordinates.
make_2d_gaussian_window(height, width, ...)	Create a 2D Gaussian window.
apply_gaussian_filter(mat, sigma_x, sigma_y)	Filtering an image in the Fourier domain using a 2D Gaussian window.
apply_regularization_filter(mat, alpha[, ...])	Apply a regularization filter using the method in Ref.
transform_1d_window_to_2d(win_1d)	Transform a 1d-window to 2d-window.
detect_sample(sinogram[, sino_type])	To check if there is a sample in a sinogram using the "double-wedge" property of the Fourier transform of the sinogram (Ref.
fix_non_sample_areas(overlap_metadata[, ...])	Used to fix overlap values of grid-cells without sample by copying from its neighbours.
locate_slice(slice_idx, height, overlap_metadata)	Locate slice indices in grid-rows given a slice index of the reconstruction data as a whole.
locate_slice_chunk(slice_start, slice_stop, ...)	Locate slice indices in grid-rows given slice indices of the reconstruction data as a whole.
generate_spiral_positions(step, num_pos, ...)	Generate Fermat spiral positions.
find_center_visual_sinograms(sino_180, ...)	For visually finding the center-of-rotation (COR) using converted 360-degree sinograms from a 180-degree sinogram at different CORs (Ref.
find_center_visual_slices(sinogram, output, ...)	For visually finding the center-of-rotation (COR) using reconstructed slices at different CORs.

algotom.util.utility.**apply_method_to_multiple_sinograms**(*data*, *method*, *para*, *ncore=None*, *prefer='threads'*)

Apply a processing method (in "filtering", "removal", and "reconstruction" module) to multiple sinograms or multiple slices in parallel.

Parameters

- **data** (*array_like or hdf object*) – 3D array data where sinograms/slices are extracted along axis 1, e.g [:, i, :].
- **method** (*str*) – Name of a method. e.g. "remove_stripe_based_sorting".
- **para** (*list*) – Parameters of the method. e.g. [21, 1]
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **prefer** (*{"threads", "processes"}*) – Prefer backend for parallel processing.

Returns

array_like – Same axis-definition as the input.

algotom.util.utility.**mapping**(*mat*, *x_mat*, *y_mat*, *order=1*, *mode='reflect'*)

Apply a geometric transformation to a 2D array

Parameters

- **mat** (*array_like*) – 2D array.
- **x_mat** (*array_like*) – 2D array of the x-coordinates.
- **y_mat** (*array_like*) – 2D array of the y-coordinates.
- **order** (*int, optional*) – The order of the spline interpolation, default is 1. The order has to be in the range 0-5.
- **mode** (*{'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional*) – The mode parameter determines how the input array is extended beyond its boundaries. Default is 'reflect'.

Returns

array_like – 2D array.

algotom.util.utility.**make_circle_mask**(*width*, *ratio*)

Create a circle mask.

Parameters

- **width** (*int*) – Width of a square array.
- **ratio** (*float*) – Ratio between the diameter of the mask and the width of the array.

Returns

array_like – Square array.

algotom.util.utility.**sort_forward**(*mat*, *axis=0*)

Sort gray-scales of an image along an axis. e.g. axis=0 is to sort along each column.

Parameters

- **mat** (*array_like*) – 2D array.
- **axis** (*int*) – Axis along which to sort.

Returns

- **mat_sort** (*array_like*) – 2D array. Sorted image.

- **mat_index** (*array_like*) – 2D array. Index array used for sorting backward.

algotom.util.utility.**sort_backward**(*mat*, *mat_index*, *axis=0*)

Sort gray-scales of an image using an index array provided. e.g. axis=0 is to sort each column.

Parameters

- **mat** (*array_like*) – 2D array.
- **mat_index** (*array_like*) – 2D array. Index array used for sorting.
- **axis** (*int*) – Axis along which to sort.

Returns

mat_sort (*array_like*) – 2D array. Sorted image.

algotom.util.utility.**separate_frequency_component**(*mat*, *axis=0*, *window=None*)

Separate low and high frequency components of an image along an axis. e.g. axis=0 is to apply the separation to each column.

Parameters

- **mat** (*array_like*) – 2D array.
- **axis** (*int*) – Axis along which to apply the filter.
- **window** (*array_like or dict*) – 1D array or a dictionary which given the name of a window in the scipy_window list and its parameters (without window-length). E.g window={"name": "gaussian", "sigma": 5}

Returns

- **mat_low** (*array_like*) – 2D array. Low-frequency image.
- **mat_high** (*array_like*) – 2D array. High-frequency image.

algotom.util.utility.**generate_fitted_image**(*mat*, *order*, *axis=0*, *num_chunk=1*)

Apply a polynomial fitting along an axis of an image. e.g. axis=0 is to apply the fitting to each column.

Parameters

- **mat** (*array_like*) – 2D array.
- **order** (*int*) – Order of the polynomial used to fit.
- **axis** (*int*) – Axis along which to apply the filter.
- **num_chunk** (*int*) – Number of chunks of rows or columns to apply the fitting.

Returns

mat_fit (*array_like*)

algotom.util.utility.**detect_stripe**(*list_data*, *snr*)

Locate stripe positions using Algorithm 4 in Ref. [1]

Parameters

- **list_data** (*array_like*) – 1D array. Normalized data.
- **snr** (*float*) – Ratio (>1.0) for stripe detection. Greater is less sensitive.

Returns

array_like – 1D binary mask.

References

[1] : https://doi.org/10.1364/OE.26.028396

algotom.util.utility.**calculate_regularization_coefficient**(*width*, *alpha*)

Calculate coefficients used for the regularization-based method. Eq. (7) in Ref. [1].

Parameters

- **width** (*int*) – Width of a square array.
- **alpha** (*float*) – Regularization parameter.

Returns

float – Square array.

References

[1] : https://doi.org/10.1016/j.aml.2010.08.022

algotom.util.utility.**make_2d_butterworth_window**(*width*, *height*, *u*, *v*, *n*)

Create a 2d window from the 1D Butterworth window.

Parameters

- **height** (*int*) – Height of the window.
- **width** (*int*) – Width of the window.
- **u** (*int*) – Cutoff frequency.
- **n** (*int*) – Filter order.
- **v** (*int*) – Number of rows (= 2*v) around the height middle are the 1D Butterworth windows.

Returns

array_like – 2D array.

algotom.util.utility.**make_2d_damping_window**(*width*, *height*, *size*, *window_name='gaussian'*)

Make 2D damping window from a list of 1D window for a Fourier-space filter, i.e. a high-pass filter.

Parameters

- **height** (*int*) – Height of the window.
- **width** (*int*) – Width of the window.
- **size** (*int*) – Sigma of a Gaussian window or cutoff frequency of a Butterworth window.
- **window_name** (*str, optional*) – Two options: "gaussian" or "butter".

Returns

array_like – 2D array of the window.

algotom.util.utility.**apply_wavelet_decomposition**(*mat*, *wavelet_name*, *level=None*)

Apply 2D wavelet decomposition.

Parameters

- **mat** (*array_like*) – 2D array.
- **wavelet_name** (*str*) – Name of a wavelet. E.g. "db5"
- **level** (*int, optional*) – Decomposition level. It is constrained to return an array with a minimum size of larger than 16 pixels.

Returns

list – The first element is an 2D-array, next elements are tuples of three 2D-arrays. i.e [mat_n, (cH_level_n, cV_level_n, cD_level_n), ..., (cH_level_1, cV_level_1, cD_level_1)]

algotom.util.utility.**apply_wavelet_reconstruction**(*data*, *wavelet_name*, *ignore_level=None*)

Apply 2D wavelet reconstruction.

Parameters

- **data** (*list or tuple*) – The first element is an 2D-array, next elements are tuples of three 2D-arrays. i.e [mat_n, (cH_level_n, cV_level_n, cD_level_n), ..., (cH_level_1, cV_level_1, cD_level_1)].
- **wavelet_name** (*str*) – Name of a wavelet. E.g. "db5"
- **ignore_level** (*int, optional*) – Decomposition level to be ignored for reconstruction.

Returns

array_like – 2D array. Note that the sizes of the array are always even numbers.

algotom.util.utility.**check_level**(*level*, *n_level*)

Supplementary method for the method of "apply_filter_to_wavelet_component". To check if the provided level is in the right format.

algotom.util.utility.**apply_filter_to_wavelet_component**(*data*, *level=None*, *order=1*, *method=None*, *para=None*)

Apply a filter to a component of the wavelet decomposition of an image.

Parameters

- **data** (*list or tuple*) – The first element is an 2D-array, next elements are tuples of three 2D-arrays. i.e [mat_n, (cH_level_n, cV_level_n, cD_level_n), ..., (cH_level_1, cV_level_1, cD_level_1)].
- **level** (*int, list of int, or None*) – Decomposition level to be applied the filter.
- **order** (*{0, 1, 2}*) – Specify which component in a tuple, (cH_level_n, cV_level_n, cD_level_n), to be filtered.
- **method** (*str*) – Name of the filter in the namespace. E.g. method="gaussian_filter"
- **para** (*list or tuple*) – Parameters of the filter. E.g para=[(1,11)]

Returns

list or tuple – The first element is an 2D-array, next elements are tuples of three 2D-arrays. i.e [mat_n, (cH_level_n, cV_level_n, cD_level_n), ..., (cH_level_1, cV_level_1, cD_level_1)].

algotom.util.utility.**interpolate_inside_stripe**(*mat*, *list_mask*, *kind='linear'*)

Interpolate gray-scales inside vertical stripes of an image. Stripe locations given by a binary 1D-mask.

Parameters

- **mat** (*array_like*) – 2D array.
- **list_mask** (*array_like*) – 1D array. Must equal the width of an image.
- **kind** (*{'linear', 'cubic', 'quintic'}, optional*) – The kind of spline interpolation to use. Default is 'linear'.

Returns

array_like

algotom.util.utility.**rectangular_from_polar**(*width_reg*, *height_reg*, *width_pol*, *height_pol*)

Generate coordinates of a rectangular grid from polar coordinates.

Parameters

- **width_reg** (*int*) – Width of an image in the Cartesian coordinate system.
- **height_reg** (*int*) – Height of an image in the Cartesian coordinate system.
- **width_pol** (*int*) – Width of an image in the polar coordinate system.
- **height_pol** (*int*) – Height of an image in the polar coordinate system.

Returns

- **x_mat** (*array_like*) – 2D array. Broadcast of the x-coordinates.
- **y_mat** (*array_like*) – 2D array. Broadcast of the y-coordinates.

algotom.util.utility.**polar_from_rectangular**(*width_pol*, *height_pol*, *width_reg*, *height_reg*)

Generate polar coordinates from grid coordinates.

Parameters

- **width_pol** (*int*) – Width of an image in the polar coordinate system.
- **height_pol** (*int*) – Height of an image in the polar coordinate system.
- **width_reg** (*int*) – Width of an image in the Cartesian coordinate system.
- **height_reg** (*int*) – Height of an image in the Cartesian coordinate system.

Returns

- **r_mat** (*array_like*) – 2D array. Broadcast of the r-coordinates.
- **theta_mat** (*array_like*) – 2D array. Broadcast of the theta-coordinates.

algotom.util.utility.**transform_slice_forward**(*mat*, *coord_mat=None*)

Transform a reconstructed image into polar coordinates.

Parameters

- **mat** (*array_like*) – Square array. Reconstructed image.
- **coord_mat** (*tuple of array_like, optional*) – (Square array of x-coordinates , square array of y-coordinates) or generated if None.

Returns

array_like – Transformed image.

algotom.util.utility.**transform_slice_backward**(*mat*, *coord_mat=None*)

Transform a reconstructed image in polar coordinates back to rectangular coordinates.

Parameters

- **mat** (*array_like*) – Square array. Reconstructed image in polar coordinates.
- **coord_mat** (*tuple of array_like, optional*) – (Square array of r-coordinates , square array of theta-coordinates) or generated if None.

Returns

array_like – Transformed image.

algotom.util.utility.**make_2d_gaussian_window**(*height*, *width*, *sigma_x*, *sigma_y*)

Create a 2D Gaussian window.

Parameters

- **height** (*int*) – Height of the image.
- **width** (*int*) – Width of the image.
- **sigma_x** (*int*) – Sigma in the x-direction.
- **sigma_y** (*int*) – Sigma in the y-direction.

Returns

array_like – 2D array.

algotom.util.utility.**apply_gaussian_filter**(*mat*, *sigma_x*, *sigma_y*, *pad=None*, *mode=None*)

Filtering an image in the Fourier domain using a 2D Gaussian window. Smaller is stronger.

Parameters

- **mat** (*array_like*) – 2D array.
- **sigma_x** (*int*) – Sigma in the x-direction.
- **sigma_y** (*int*) – Sigma in the y-direction.
- **pad** (*int or None*) – Padding for the Fourier transform.
- **mode** (*str, list of str, or tuple of str*) – Padding method. One of options : 'reflect', 'edge', 'constant'. Full list is at: https://numpy.org/doc/stable/reference/generated/numpy.pad.html

Returns

array_like – 2D array. Filtered image.

algotom.util.utility.**apply_1d_regularizer**(*list_data*, *sijmat*)

Supplementary method for the method of "apply_regularization_filter". To apply a regularizer to an 1D-array.

algotom.util.utility.**apply_regularization_filter**(*mat*, *alpha*, *axis=1*, *ncore=None*)

Apply a regularization filter using the method in Ref. [1]. Note that it's computationally costly.

Parameters

- **mat** (*array_like*) – 2D array
- **alpha** (*float*) – Regularization parameter, e.g. 0.001. Smaller is stronger.
- **axis** (*int*) – Axis along which to apply the filter.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.

Returns

array_like – 2D array. Smoothed image.

References

[1] : https://doi.org/10.1016/j.aml.2010.08.022

algotom.util.utility.**transform_1d_window_to_2d**(*win_1d*)

Transform a 1d-window to 2d-window. Useful for designing a Fourier filter.

Parameters
win_1d (*array_like*) – 1D array.

Returns
win_2d (*array_like*) – Square array, a 2D version of the 1d-window.

algotom.util.utility.**detect_sample**(*sinogram*, *sino_type='180'*)

To check if there is a sample in a sinogram using the "double-wedge" property of the Fourier transform of the sinogram (Ref. [1]).

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image
- **sino_type** (*{"180", "360"}*) – Sinogram type : 180-degree or 360-degree.

Returns
bool – True if there is a sample.

References

[1] : https://doi.org/10.1364/OE.418448

algotom.util.utility.**fix_non_sample_areas**(*overlap_metadata*, *direction='horizontal'*)

Used to fix overlap values of grid-cells without sample by copying from its neighbours. Input is a 3d-array of overlapping values for each grid cell. For example, to a 5 x 3 (n_row x n_column) grid scans, the shape for overlapping values in the horizontal direction is: 5 x 2 (n_column - 1) x 2 (overlap, side). The shape for overlapping values in the vertical direction is: 4 (n_row - 1) x 3 x 2 (overlap, side). The order of calculating overlapping values in a grid is left-to-right, top-to-bottom.

Parameters

- **overlap_metadata** (*array_like*) – A matrix of overlap values of each grid-cell where each element is a list of [overlap, side].
- **direction** (*{"horizontal", "vertical"}*) – Direction of overlapping calculation.

Returns
metadata (*array_like*)

algotom.util.utility.**locate_slice**(*slice_idx*, *height*, *overlap_metadata*)

Locate slice indices in grid-rows given a slice index of the reconstruction data as a whole.

Parameters

- **slice_idx** (*int*) – Slice index of full reconstruction data.
- **height** (*int*) – Height of a projection image of each grid-cell.
- **overlap_metadata** (*array_like*) – A matrix of overlap values of each grid-row where each element is a list of [overlap, side]. Used to stitch the grid-data along the row-direction.

Returns

list of int and float – If the slice is not in the overlapping area between two grid-rows, the result is a list of [grid_row_index, slice_index, weight_factor]. If the slice is in the overlapping area between two grid-rows, the result is a list of [[grid_row_index_0, slice_index_0, weight_factor_0], [grid_row_index_1, slice_index_1, weight_factor_1]]

algotom.util.utility.**locate_slice_chunk**(*slice_start*, *slice_stop*, *height*, *overlap_metadata*)

Locate slice indices in grid-rows given slice indices of the reconstruction data as a whole.

Parameters

- **slice_start** (*int*) – Starting index of full reconstruction data.
- **slice_stop** (*int*) – Stopping index of full reconstruction data.
- **height** (*int*) – Height of a projection image of each grid-cell.
- **overlap_metadata** (*array_like*) – A matrix of overlap values of each grid-row where each element is a list of [overlap, side]. Used to stitch the grid-data along the row-direction.

Returns

list of list of int and float – List of results for each slice index. If a slice is not in the overlapping area between two grid-rows, the result is a list of [grid_row_index, slice_index, weight_factor]. If a slice is in the overlapping area between two grid-rows, the result is a list of [[grid_row_index_0, slice_index_0, weight_factor_0], [grid_row_index_1, slice_index_1, weight_factor_1]].

algotom.util.utility.**generate_spiral_positions**(*step*, *num_pos*, *height*, *width*, *spiral_shape=1.0*)

Generate Fermat spiral positions. Unit is pixel.

Parameters

- **step** (*int*) – Step size in pixel. ~ 20-> 40
- **num_pos** (*int*) – Number of positions.
- **height** (*int*) – Height of the field of view (in pixel).
- **width** (*int*) – Width of the field of view (in pixel).
- **spiral_shape** (*float, optional*) – To define the spiral shape.

Returns

array_like – 2D array. List of (x,y) positions

algotom.util.utility.**find_center_visual_sinograms**(*sino_180*, *output*, *start*, *stop*, *step=1*, *zoom=1.0*, *display=False*)

For visually finding the center-of-rotation (COR) using converted 360-degree sinograms from a 180-degree sinogram at different CORs (Ref. [1]).

Parameters

- **sino_180** (*array_like*) – 2D array. 180-degree sinogram.
- **output** (*str*) – Base folder for saving converted 360-degree sinograms.
- **start** (*float*) – Starting point for searching CoR.
- **stop** (*float*) – Ending point for searching CoR.
- **step** (*float*) – Searching step.
- **zoom** (*float*) – To resize output images. For example, 0.5 <=> reduce the size of output images by half.
- **display** (*bool*) – Print the output if True.

Returns
str – Folder path to tif images.

References

[1] : https://doi.org/10.1364/OE.22.019078

algotom.util.utility.**find_center_visual_slices**(*sinogram*, *output*, *start*, *stop*, *step=1*, *zoom=1.0*, *method='dfi'*, *gpu=False*, *angles=None*, *ratio=1.0*, *filter_name='hann'*, *apply_log=True*, *ncore=None*, *display=False*)

For visually finding the center-of-rotation (COR) using reconstructed slices at different CORs.

Parameters

- **sinogram** (*array_like*) – 2D array. Sinogram image.
- **output** (*str*) – Base folder for saving reconstructed slices.
- **start** (*float*) – Starting point for searching CoR.
- **stop** (*float*) – Ending point for searching CoR.
- **step** (*float*) – Searching step.
- **zoom** (*float*) – To resize input and output images. For example, 0.5 <=> reduce the size of images by half.
- **method** (*{"dfi", "gridrec", "fbp", "astra"}*) – To select a backend method for reconstruction.
- **gpu** (*bool, optional*) – Use GPU for computing if True.
- **angles** (*array_like, optional*) – 1D array. List of angles (in radian) corresponding to the sinogram.
- **ratio** (*float, optional*) – To apply a circle mask to the reconstructed image.
- **filter_name** (*{None, "hann", "bartlett", "blackman", "hamming", "nuttall", "parzen", "triang"}*) – Apply a smoothing filter.
- **apply_log** (*bool, optional*) – Apply the logarithm function to the sinogram before reconstruction.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **display** (*bool*) – Print the output if True.

Returns
str – Folder path to tif images.

algotom.util.correlation

Module of correlation-based methods for finding shifts between images or stacks of images. The methods are designed to be flexible to:

- Run on multicore CPU or GPU.
- Use small/large RAM or small/large GPU memory.
- Work with small/large size of data.
- Find shifts locally or globally.

Functions:

normalize_image(mat)	Normalize an image.
generate_correlation_map(ref_mat, mat[, ...])	Generate the correlation map (Pearson coefficients) between two images by shifting the second image over the reference image.
locate_peak(mat[, sub_pixel, method, dim, ...])	Locate the position of the maximum value of a 2d-array with sub-pixel accuracy.
find_shift_based_correlation_map(ref_mat, mat)	Find the relative translations of the second image against the first image using the correlation map generated by sliding the 2nd image over the 1st one.
find_local_shifts(ref_mat, mat[, dim, ...])	To find local shifts (in x and y direction) between two images by selecting a small area/volume of the second image and sliding over a slightly larger area/volume of the reference image.
find_global_shift_based_local_shifts(...[, ...])	Find global shift between two images based on finding local shifts.
find_local_shifts_umpa(ref_mat, mat[, ...])	To find local shifts (in x and y direction) of each pixel between two 3d-images by selecting a small volume of the second image and sliding over a slightly larger volume of the reference image.

algotom.util.correlation.**normalize_image**(*mat*)

Normalize an image.

Parameters
mat (*array_like*) – 2D or 3D array.

Returns
array_like – 2D or 3D array. Normalized image.

algotom.util.correlation.**generate_correlation_map**(*ref_mat*, *mat*, *gpu=False*, *block=(16, 16)*)

Generate the correlation map (Pearson coefficients) between two images by shifting the second image over the reference image.

Parameters

- **ref_mat** (*array_like*) – 2D or 3D array. The reference image (e.g. with height0 x width0).
- **mat** (*array_like*) – 2D or 3D array. The second image (e.g. with height1 x width1). If 3D, the size of the first dimension (i.e. depth) must be the same as the reference image.
- **gpu** (*bool, optional*) – Use GPU for computing if True.
- **block** (*tuple of two integer-values, optional*) – Size of a GPU block. E.g. (4,4), (8, 8), ...

Returns
array_like – 2D array with the size of (height0-height1+1) x (width0-width1+1).

algotom.util.correlation.**locate_peak**(*mat*, *sub_pixel=True*, *method='diff'*, *dim=2*, *size=3*, *max_peak=True*)

Locate the position of the maximum value of a 2d-array with sub-pixel accuracy.

Parameters

- **mat** (*array_like*) – 2D array.
- **sub_pixel** (*bool, optional*) – Enable sub-pixel location.
- **method** (*{"diff", "poly_fit"}*) – Method for finding sub-pixel shift. Two options: a differential method (Ref. [1]) or a polynomial method (Ref. [2]).
- **dim** (*{1, 2}*) – Searching dimension for sub-pixel location.
- **size** (*int*) – Window size around the integer location of the maximum value used for sub-pixel searching.
- **max_peak** (*bool, optional*) – Used to locate the minimum value if False.

Returns
list of two floats – Sub-pixel position (x, y), i.e. (column, row), of the maximum value.

References

[1] : https://doi.org/10.48550/arXiv.0712.4289

[2] : https://doi.org/10.1088/0957-0233/17/6/045

algotom.util.correlation.**find_shift_based_correlation_map**(*ref_mat*, *mat*, *margin=10*, *axis=None*, *sub_pixel=True*, *method='diff'*, *dim=2*, *size=3*, *gpu=False*, *block=(16, 16)*)

Find the relative translations of the second image against the first image using the correlation map generated by sliding the 2nd image over the 1st one. If the inputs are 3d-arrays, the size of the first axis must be the same.

Parameters

- **ref_mat** (*array_like*) – 2D or 3D array. Reference image.
- **mat** (*array_like*) – 2D or 3D array. The second image. If 3D, the size of the first dimension (i.e. depth) must be the same as the reference image.
- **margin** (*int, optional*) – If the second image and the first image are the same size, the second image will be cropped with the margin amount from the edges before sliding. Basically, this value defines the sliding range.
- **axis** (*{0, 1, None}*) – To select the axis for sliding. If the inputs are 3d-arrays, 0 and 1 corresponding to axis-1 and axis-2 of a 3d-array.
- **sub_pixel** (*bool, optional*) – Enable sub-pixel location.
- **method** (*{"diff", "poly_fit"}*) – Method for finding sub-pixel shift. Two options: a differential method (Ref. [1]) or a polynomial method (Ref. [2]).
- **dim** (*{1, 2}*) – Searching dimension for sub-pixel location.
- **size** (*int*) – Window size around the integer location of the maximum value used for sub-pixel searching.
- **gpu** (*bool, optional*) – Use GPU for computing if True.

- **block** (*tuple of two integer-values, optional*) – Size of a GPU block. E.g. (4,4), (8, 8), ...

Returns

list of 2 floats – The shifts in x and y-direction of the second image referred to the middle of the reference image.

References

[1] : https://doi.org/10.48550/arXiv.0712.4289

[2] : https://doi.org/10.1088/0957-0233/17/6/045

algotom.util.correlation.**find_local_shifts**(*ref_mat*, *mat*, *dim=1*, *win_size=7*, *margin=10*, *method='diff'*, *size=3*, *gpu=False*, *block=(16, 16)*, *ncore=None*, *norm=True*, *norm_global=False*, *chunk_size=100*)

To find local shifts (in x and y direction) between two images by selecting a small area/volume of the second image and sliding over a slightly larger area/volume of the reference image.

Parameters

- **ref_mat** (*array_like*) – 2D/3D array, can be a numpy array or hdf object. Reference image.
- **mat** (*array_like*) – 2D/3D array, can be a numpy array or hdf object. The second image, must be the same size as the reference image.
- **dim** (*{1, 2}*) – To find the shifts (in x and y) separately or together.
- **win_size** (*int*) – Size of local areas in the second image.
- **margin** (*int*) – To define the sliding range of the second image.
- **method** (*{"diff", "poly_fit"}*) – Method for finding sub-pixel shift. Two options: a differential method (Ref. [1]) or a polynomial method (Ref. [2]). The "poly_fit" option is not available if using GPU.
- **size** (*int*) – Window size around the integer location of the maximum value used for sub-pixel location. Adjustable if using the polynomial method.
- **gpu** (*{False, True, "hybrid"}*) – Use GPU for computing if True or in "hybrid" mode.
- **block** (*tuple of two integer-values, optional*) – Size of a GPU block. E.g. (8, 8), (16, 16), (32, 32), ...
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **norm** (*bool, optional*) – Normalizing the inputs if True.
- **norm_global** (*bool, optional*) – Normalize by using the full size of the inputs if True.
- **chunk_size** (*int or None*) – Size of each chunk extracted along the height of the image.

Returns

list of two 2d-arrays – x-shift array and y-shift array. Zeros at the outer area of the size of (margin + win_size // 2).

References

[1] : https://doi.org/10.48550/arXiv.0712.4289

[2] : https://doi.org/10.1088/0957-0233/17/6/045

algotom.util.correlation.**find_global_shift_based_local_shifts**(*ref_mat*, *mat*, *win_size*, *margin*, *list_ij=None*, *num_point=None*, *global_value='mixed'*, *gpu=False*, *block=32*, *sub_pixel=True*, *method='diff'*, *size=3*, *ncore=None*, *norm=False*, *return_list=False*)

Find global shift between two images based on finding local shifts.

Parameters

- **ref_mat** (*array_like*) – 2D array. Reference image.
- **mat** (*array_like*) – 2D array. The 2nd image. Must be the same size as the reference image.
- **win_size** (*int*) – To define the size of the area around the selected pixel of the 2nd image. E.g. 41, 61, ..
- **margin** (*int*) – To define the searching range (in pixel) for finding shift. E.g. 20, 30,...
- **list_ij** (*list of lists of int or None*) – List of indices of points used for local search. Accept the value of [i_index, j_index] for a single point or [[i_index0, i_index1,...], [j_index0, j_index1,...]] for multiple points. Automatically generated if None.
- **num_point** (*int or None*) – Number of points used for local search if list_ij is None.
- **global_value** (*{"median", "mean", "mixed"}*) – Method for calculating the global value from local values.
- **gpu** (*bool, optional*) – Use GPU for computing if True. If win_size and image size is large (e.g. > 201 x 2k x 2k), using CPU may be better.
- **block** (*int, optional*) – Size of a GPU block if using GPU. E.g. 16, 32, 64, ...
- **sub_pixel** (*bool, optional*) – Enable sub-pixel location.
- **method** (*{"diff", "poly_fit"}*) – Method for finding sub-pixel shift. Two options: a differential method (Ref. [1]) or a polynomial method (Ref. [2]). The "poly_fit" option is not available if using GPU.
- **size** (*int, optional*) – Window size around the integer location of the maximum value used for sub-pixel searching.
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **norm** (*bool, optional*) – Normalize the input images if True.
- **return_list** (*bool*) – Return all local values if True.

Returns

- *float or list of float* – Shift in x-direction. Return a list of float if return_list is True.
- *float or list of float* – Shift in y-direction. Return a list of float if return_list is True.

References

[1] : https://doi.org/10.48550/arXiv.0712.4289

[2] : https://doi.org/10.1088/0957-0233/17/6/045

algotom.util.correlation.**find_local_shifts_umpa**(*ref_mat*, *mat*, *win_size=7*, *margin=10*, *method='diff'*, *size=3*, *gpu=True*, *block=(16, 16)*, *ncore=None*, *chunk_size=100*, *filter_name='hamming'*, *dark_signal=False*)

To find local shifts (in x and y direction) of each pixel between two 3d-images by selecting a small volume of the second image and sliding over a slightly larger volume of the reference image. The cost function uses the formula in Ref. [1], known as UMPA.

Parameters

- **ref_mat** (*array_like*) – 3D array, can be a numpy array or hdf object. Reference image.
- **mat** (*array_like*) – 3D array, can be a numpy array or hdf object. The second image, must be the same size as the reference image.
- **win_size** (*int*) – Size of local areas in the second image.
- **margin** (*int*) – To define the sliding range of the second image.
- **method** (*{"diff", "poly_fit"}*) – Method for finding sub-pixel shift. Two options: a differential method (Ref. [2]) or a polynomial method (Ref. [3]). The "poly_fit" option is not available if using GPU.
- **size** (*int*) – Window size around the integer location of the maximum value used for sub-pixel location. Adjustable if using the polynomial method.
- **gpu** (*bool*) – Use GPU for computing if True.
- **block** (*tuple of two integer-values, optional*) – Size of a GPU block. E.g. (8, 8), (16, 16), (32, 32), ...
- **ncore** (*int or None*) – Number of cpu-cores used for computing. Automatically selected if None.
- **chunk_size** (*int or None*) – Size of each chunk extracted along the height of the image. Use to avoid the out of memory problem.
- **filter_name** (*{None, "hann", "bartlett", "blackman", "hamming", "nuttall", "parzen", "triang"}*) – To select a smoothing filter.
- **dark_signal** (*bool*) – Return both dark-signal image and transmission-signal image if True

Returns

list of two 2d-arrays or four 2d-arrays – x-shift image and y-shift image. Zeros at the outer area of the size of (margin + win_size // 2). And/or dark-signal image and transmission-signal image If the 'dark_signal' option is True.

References

[1] : https://doi.org/10.1103/PhysRevLett.118.203903

[2] : https://doi.org/10.48550/arXiv.0712.4289

[3] : https://doi.org/10.1088/0957-0233/17/6/045

1.8 Credits

1.8.1 Citations

If Algotom is useful for your project, citing the following article [C1] is very much appreciated.

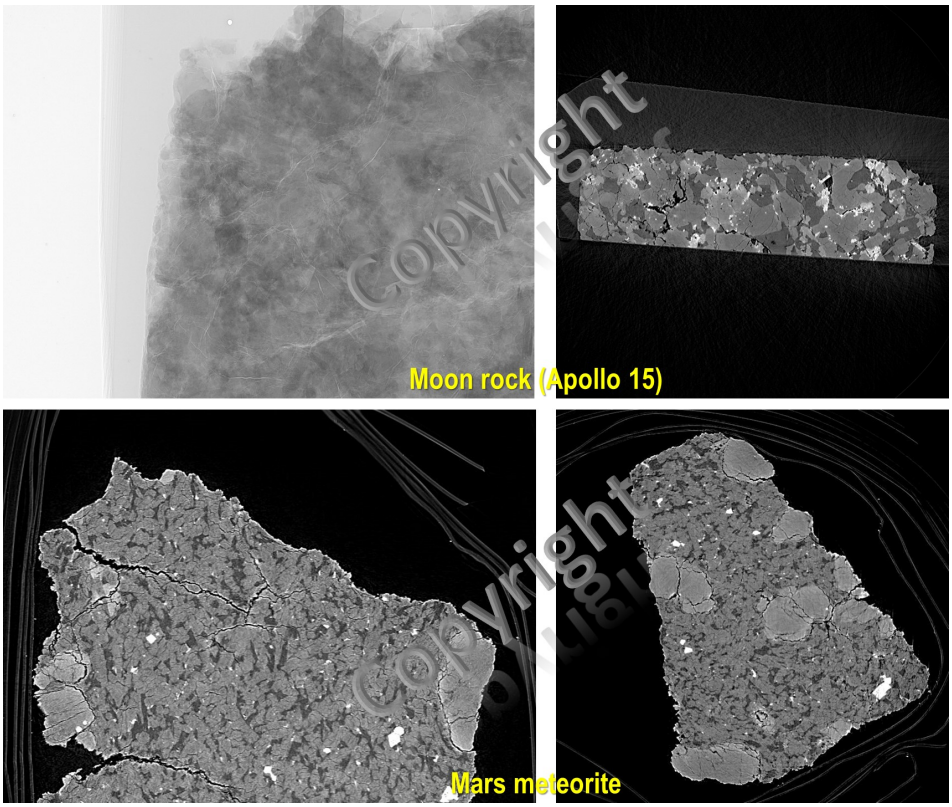
Algorithms, methods, or techniques implemented in a scientific software package are crucial for its success. This is the same for Algotom. Acknowledging algorithms you use through Algotom is also very much appreciated.

1.8.2 References

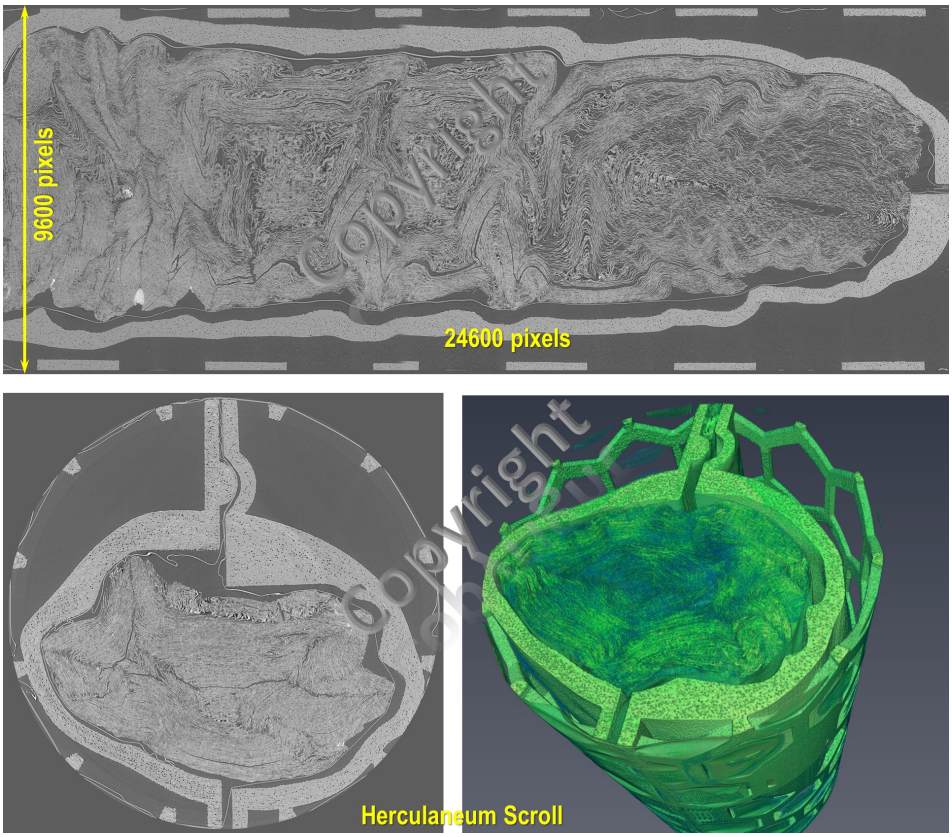
1.9 Highlights

Algotom was used for some experiments featured on media:

- Scanning Moon rocks and Martian meteorites using helical scans with offset rotation-axis. Featured on Reuters.



- Scanning Herculaneum Scrolls using grid scans with offset rotation-axis respect to the grid's FOV. Featured on BBC. The latest updates on the scroll's reading progress are here.



- Scanning Little Foot fossil using two-camera detector with offset rotation-axis. Featured on BBC.

1.10 Quick links

- How to set up Python workspace for coding and installing libraries:
  - Section 4.1.
  - Section 1.1.
- How to read/write or explore hdf/nxs/h5 files:
  - Section 4.2.1.
  - Section 1.2.1
  - Broh5 software.
  - API of loading a hdf file.
  - API of writing to a hdf file.
  - Script for exploring a hdf file.
- How to read/write tiff images:
  - Section 1.2.2.
  - Section 4.2.2.
  - Loader/saver API.
- How to process standard tomography data:
  - Workflow:
    - Section 1.4.
    - Section 4.5.
  - Command line interface scripts: few-slices reconstruction, full reconstruction, data reduction:
    - Common data processing workflow.
  - Scripts:
    - Reconstructing a few slices of a standard scan.
    - Full size reconstruction of a standard scan.
- How to process half-acquisition tomography (360-degree scanning with offset center) data:
  - Demo script.
  - Section 1.4.8.2.
- How to apply distortion correction:
  - Reconstruct a scan with distortion correction.
  - Use Discorpy for finding distortion coefficients.
- How to choose ring artifact removal methods:
  - Section 4.4.
  - Section 4.5.4.
  - Sarepy documentation about ring artifacts in tomography.
- How to find center of rotation (rotation axis):

  - Section 4.5.3.
- How to process time-series tomography data:
  - Demo scripts
- How to process grid-scanning tomography data (tiled scans):
  - Reconstruct a few slices.
  - Full reconstruction: step 1.
  - Full reconstruction: step 2.
  - Full reconstruction: step 3.
- How to process helical tomography data:
  - Demo script.
- How to perform data reduction of reconstructed volume (cropping, rescaling, downsampling, reslicing,...):
  - Section 4.5.8.
  - Command line interface script.
- How to process speckle-based phase-contrast tomography data:
  - Section 5.1.
  - Demo scripts.
- How to correct tilted tomography data:
  - Demo script 1.
  - Demo script 2.
  - Tomography alignment tutorial.
- How to automate the workflow:
  - Section 4.5.7.
  - Utility scripts.
- How tomography works:
  - Section 1.3.
  - Section 1.6.
- How to generate simulated data:
  - Simulation module.
  - Demo script.
- How to customize ring-artifact removal methods:
  - Section 4.3.
- Tools for finding image shift, stitching images:
  - Correlation module.
  - Conversion module.
- Tools for phase unwrapping, phase retrieval:
  - Phase module.

  - Section 5.1.3.

- Datasets for testing algorithms:
  - Zenodo.
  - Tomobank.
- Parallel processing, GPU programming, and high-performance computing with Numba:
  - Section 1.5.
  - GPU programming.
  - Compiling python code.
- Common mistakes and useful tips:
  - Section 4.5.9.

BIBLIOGRAPHY

[C1] Nghia T. Vo, Robert C. Atwood, Michael Drakopoulos, and Thomas Connolley. Data processing methods and data acquisition for samples larger than the field of view in parallel-beam tomography. *Opt. Express*, 29(12):17849–17874, Jun 2021. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-29-12-17849, doi:10.1364/OE.418448.

[R1] V. Argyriou and T. Vlachos. Estimation of sub-pixel motion using gradient cross-correlation. *Electronics Letters*, 39:980–982, June 2003. URL: https://digital-library.theiet.org/content/journals/10.1049/el_20030666, doi:10.1049/el:20030666.

[R2] Sebastien Berujon and Eric Ziegler. X-ray multimodal tomography using speckle-vector tracking. *Phys. Rev. Applied*, 5:044014, 2016. doi:10.1103/PhysRevApplied.5.044014.

[R3] Pan Bing, Xie Hui-min, Xu Bo-qin, and Dai Fu-long. Performance of sub-pixel registration algorithms in digital image correlation. *Measurement Science and Technology*, 17(6):1615–1621, may 2006. URL: https://doi.org/10.1088/0957-0233/17/6/045, doi:10.1088/0957-0233/17/6/045.

[R4] RN Bracewell. Strip integration in radio astronomy. *Australian Journal of Physics*, 9(2):198 – 217, 1956. URL: https://www.publish.csiro.au/ph/PH560198, doi:10.1071/PH560198.

[R5] Tilman Donath, Felix Beckmann, and Andreas Schreyer. Automated determination of the center of rotation in tomography data. *J. Opt. Soc. Am. A*, 23(5):1048–1057, May 2006. URL: https://opg.optica.org/josaa/abstract.cfm?URI=josaa-23-5-1048, doi:10.1364/JOSAA.23.001048.

[R6] G. H. Fisher and B. T. Welsch. Flct: a fast, efficient method for performing local correlation tracking. 2007. URL: https://arxiv.org/abs/0712.4289, doi:10.48550/ARXIV.0712.4289.

[R7] R.T. Frankot and R. Chellappa. A method for enforcing integrability in shape from shading algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4):439–451, 1988. doi:10.1109/34.3909.

[R8] Dennis Ghiglia and Mark Pritt. *Two-dimensional Phase Unwrapping: Theory, Algorithms, and Software*. Wiley, New York, 1998.

[R9] Doğa Gürsoy, Francesco De Carlo, Xianghui Xiao, and Chris Jacobsen. TomoPy: a framework for the analysis of synchrotron tomographic data. *Journal of Synchrotron Radiation*, 21(5):1188–1193, 2014. doi:10.1107/S1600577514013939.

[R10] Beat Münch, Pavel Trtik, Federica Marone, and Marco Stampanoni. Stripe and ring artifact removal with combined wavelet — fourier filtering. *Opt. Express*, 17(10):8567–8591, May 2009. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-17-10-8567, doi:10.1364/OE.17.008567.

[R11] Juan Martinez-Carranza, Konstantinos Falaggis, and Tomasz Kozacki. Fast and accurate phase-unwrapping algorithm based on the transport of intensity equation. *Appl. Opt.*, 56(25):7079–7088, 2017. doi:10.1364/AO.56.007079.

[R12] M.D. Pritt and J.S. Shipman. Least-squares two-dimensional phase unwrapping using fft's. *IEEE Transactions on Geoscience and Remote Sensing*, 32(3):706–708, 1994. doi:10.1109/36.297989.

[R13] G. N. Ramachandran and A. V. Lakshminarayanan. Three-dimensional reconstruction from radiographs and electron micrographs: application of convolutions instead of fourier transforms. *Proceedings of the National Academy of Sciences*, 68(9):2236–2240, 1971. URL: https://www.pnas.org/content/68/9/2236, doi:10.1073/pnas.68.9.2236.

[R14] Carsten Raven. Numerical removal of ring artifacts in microtomography. *Review of Scientific Instruments*, 69(8):2978–2980, 1998. URL: https://aip.scitation.org/doi/10.1063/1.1149043, doi:10.1063/1.1149043.

[R15] T. Simchony, R. Chellappa, and M. Shao. Direct analytical methods for solving poisson equations in computer vision problems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(5):435–446, 1990. doi:10.1109/34.55103.

[R16] Sofya Titarenko, Philip J. Withers, and Anatoly Yagola. An analytical formula for ring artefact suppression in x-ray tomography. *Applied Mathematics Letters*, 23(12):1489–1495, 2010. URL: https://www.sciencedirect.com/science/article/pii/S089396591000282X, doi:https://doi.org/10.1016/j.aml.2010.08.022.

[R17] Wim van Aarle, Willem Jan Palenstijn, Jeroen Cant, Eline Janssens, Folkert Bleichrodt, Andrei Dabravolski, Jan De Beenhouwer, K. Joost Batenburg, and Jan Sijbers. Fast and flexible x-ray tomography using the astra toolbox. *Opt. Express*, 24(22):25129–25147, 2016. doi:10.1364/OE.24.025129.

[R18] Nghia T. Vo, Robert C. Atwood, and Michael Drakopoulos. Radial lens distortion correction with sub-pixel accuracy for x-ray micro-tomography. *Opt. Express*, 23(25):32859–32868, Dec 2015. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-23-25-32859, doi:10.1364/OE.23.032859.

[R19] Nghia T. Vo, Robert C. Atwood, and Michael Drakopoulos. Superior techniques for eliminating ring artifacts in x-ray micro-tomography. *Opt. Express*, 26(22):28396–28412, Oct 2018. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-26-22-28396, doi:10.1364/OE.26.028396.

[R20] Nghia T. Vo, Robert C. Atwood, and Michael Drakopoulos. Preprocessing techniques for removing artifacts in synchrotron-based tomographic images. In Bert Müller and Ge Wang, editors, *Developments in X-Ray Tomography XII*, volume 11113, 309 – 328. International Society for Optics and Photonics, SPIE, 2019. doi:10.1117/12.2530324.

[R21] Nghia T. Vo, Michael Drakopoulos, Robert C. Atwood, and Christina Reinhard. Reliable method for calculating the center of rotation in parallel-beam tomography. *Opt. Express*, 22(16):19078–19086, Aug 2014. URL: http://www.opticsexpress.org/abstract.cfm?URI=oe-22-16-19078, doi:10.1364/OE.22.019078.

[R22] Nghia T. Vo, Hongchang Wang, Lingfei Hu, Tunhe Zhou, Marie-Christine Zdora, Hans Deyhle, Robert C. Atwood, and Michael Drakopoulos. Practical implementations of speckle-based phase-retrieval methods in python and gpu for tomography. In *Developments in X-Ray Tomography XIV*, volume 12242, 122420E. International Society for Optics and Photonics, SPIE, 2022. doi:10.1117/12.2636834.

[R23] H. Wang, S. Berujon, J. Herzen, R. Atwood, D. Laundy, A. Hipp, and K. Sawhney. X-ray phase contrast tomography by tracking near field speckle. *Scientific Repport*, 5:8762, 2015. doi:10.1038/srep08762.

[R24] Hongchang Wang and Kawal Sawhney. Hard x-ray omnidirectional differential phase and dark-field imaging. *Proceedings of the National Academy of Sciences*, 2021. doi:10.1073/pnas.2022319118.

[R25] Marie-Christine Zdora, Pierre Thibault, Tunhe Zhou, Frieder J. Koch, Jenny Romell, Simone Sala, Arndt Last, Christoph Rau, and Irene Zanette. X-ray phase-contrast imaging and metrology through unified modulated pattern analysis. *Phys. Rev. Lett.*, 118:203903, 2017. doi:10.1103/PhysRevLett.118.203903.

PYTHON MODULE INDEX

a

INDEX

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U