02_intro_sandstone_parallel_roi_tmp

August 19, 2025

[1]:
```
import warnings
warnings.simplefilter('error', RuntimeWarning)
```

[2]:
```
# -*- coding: utf-8 -*-
#  Copyright 2021 - 2022 United Kingdom Research and Innovation
#  Copyright 2021 - 2022 The University of Manchester
#  Copyright 2021 - 2022 Technical University of Denmark
#
#  Licensed under the Apache License, Version 2.0 (the "License");
#  you may not use this file except in compliance with the License.
#  You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
#  Unless required by applicable law or agreed to in writing, software
#  distributed under the License is distributed on an "AS IS" BASIS,
#  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#  See the License for the specific language governing permissions and
#  limitations under the License.
#
#   Authored by:    Jakob S. Jørgensen (DTU)
#                   Gemma Fardell (UKRI-STFC)
#                   Laura Murgatroyd (UKRI-STFC)
#                   Hannah Robarts (UKRI-STFC)
```

1 Sandstone 2D parallel-beam data reconstruction demo

This exercise walks through the steps needed to load in, preprocess and reconstruct by FBP a 2D parallel-beam data set from a synchrotron of a sandstone sample. Learning objectives are: - Load and investigate a real data set. - Determine geometric information of the data and set up CIL data structures. - Apply CIL processors to pre-process the data, including normalisation, negative log, region-of-interest and centre of rotation correction. - Compute FBP reconstruction using CIL and compare with reconstruction provided.

This example requires the data in `small.zip` from: https://zenodo.org/record/4912435 : - https://zenodo.org/record/4912435/files/small.zip

If running locally please download the data and update the `datapath` variable below.

[3]:
```
datapath = "/mnt/share/materials/SIRF/Fully3D/CIL/SandStone/"
```

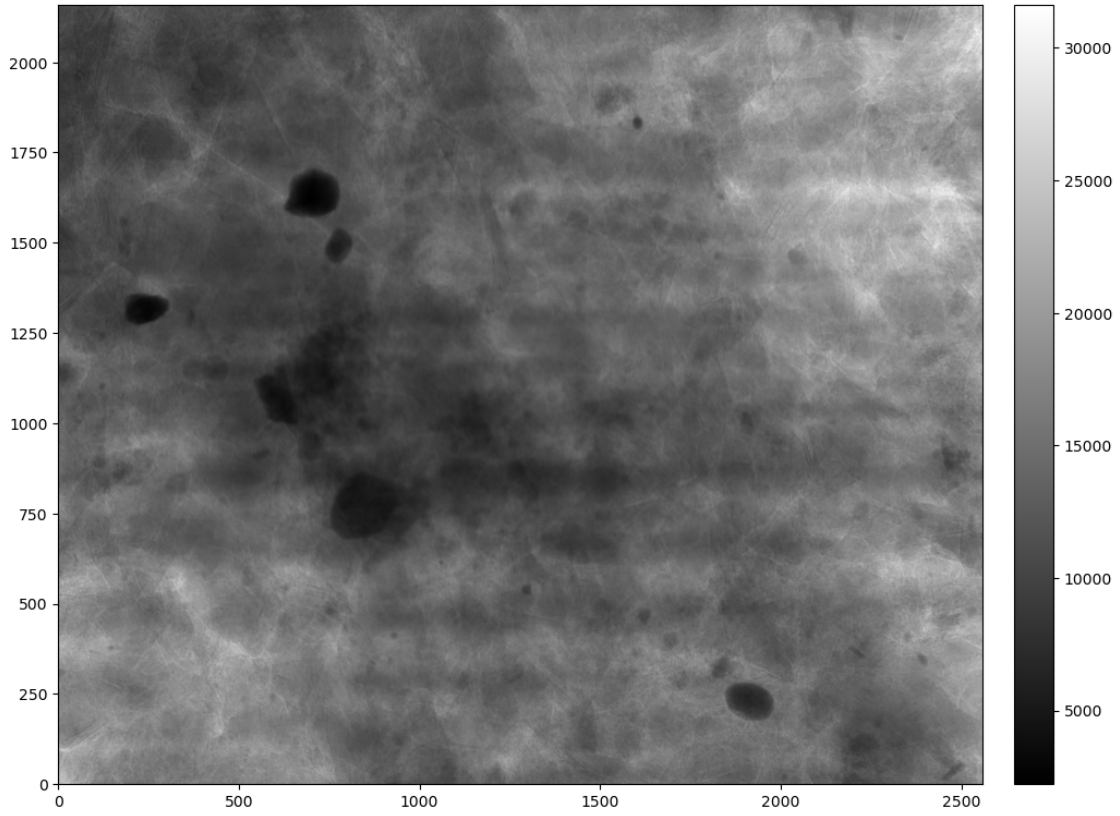
[4]:
```
# Import all CIL components needed
from cil.framework import ImageData, ImageGeometry
from cil.framework import AcquisitionGeometry, AcquisitionData

# CIL Processors
from cil.processors import CentreOfRotationCorrector, Slicer,␣
 ↪TransmissionAbsorptionConverter, Normaliser, Padder

# CIL display tools
from cil.utilities.display import show2D, show_geometry

# CIL FBP
from cil.recon import FBP

# From CIL ASTRA plugin
from cil.plugins.astra import ProjectionOperator

# All external imports
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
import os
```

Switch on printing of more info in some of the methods including `CentreOfRotationCorrector`:

[5]:
```
import logging
logging.basicConfig(level=logging.WARNING)
cil_log_level = logging.getLogger('cil.processors')
cil_log_level.setLevel(logging.INFO)
```

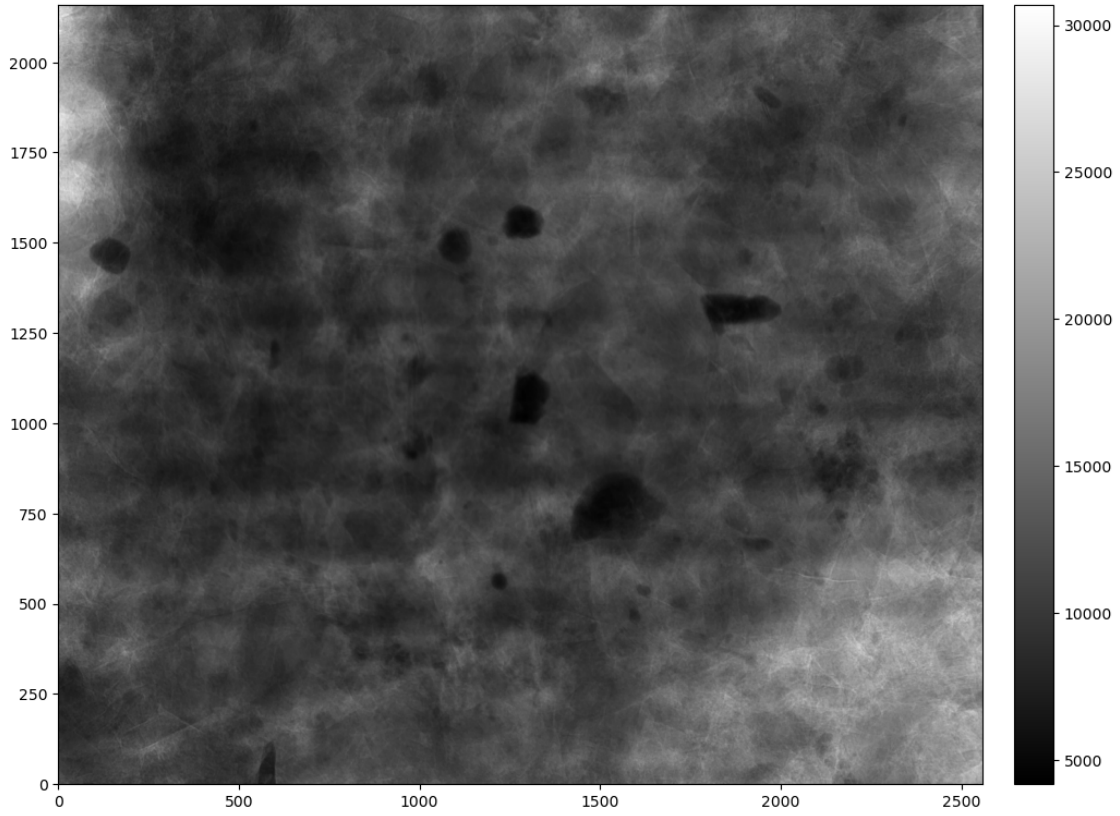
The data contains selected 2D projections, flat and dark fields, as well as complete 2D sinograms for 4 horizontal slices. We first load and display a couple of projections

[6]:
```
proj1 = plt.imread(os.path.join(datapath,"proj","BBii_0131.tif"))
show2D(proj1)
```



[6]: <cil.utilities.display.show2D at 0x7ff9249777d0>

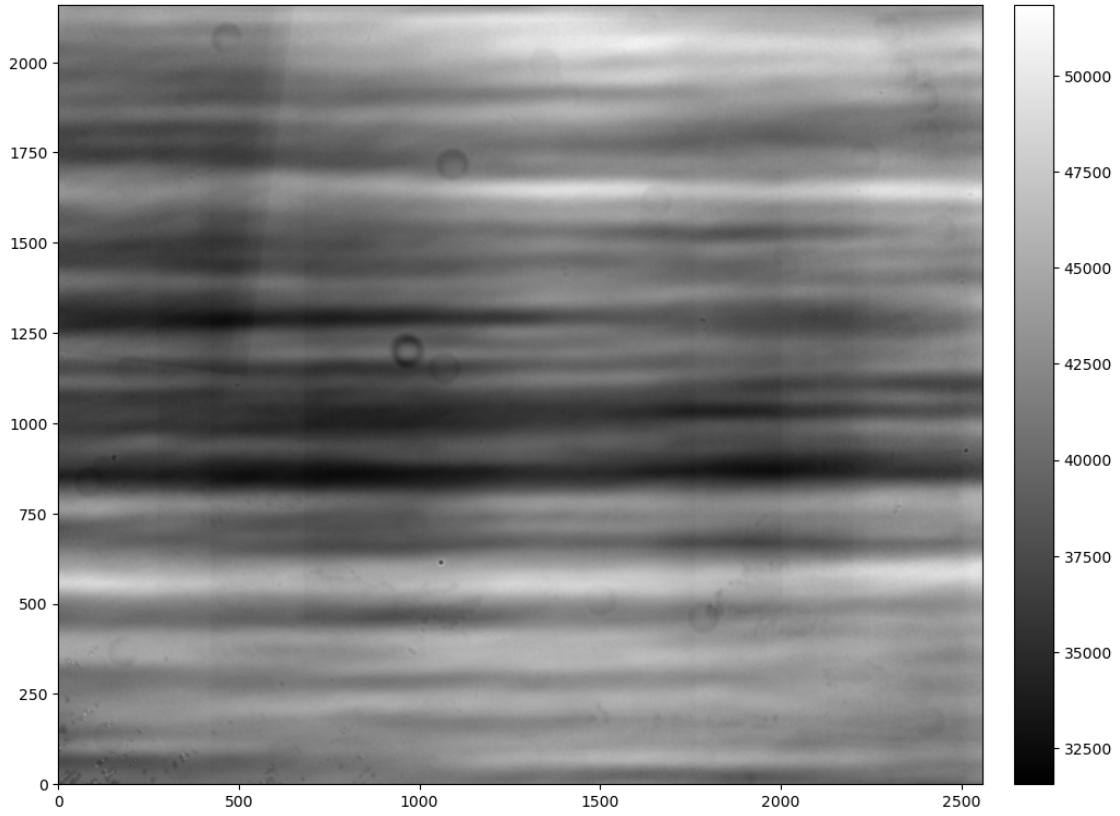
[7]:
```
proj2 = plt.imread(os.path.join(datapath,"proj","BBii_0931.tif"))
show2D(proj2)
```



[7]: <cil.utilities.display.show2D at 0x7ff9395177d0>

We also load and display a flat field (image taken before projections with source on, and sample out):

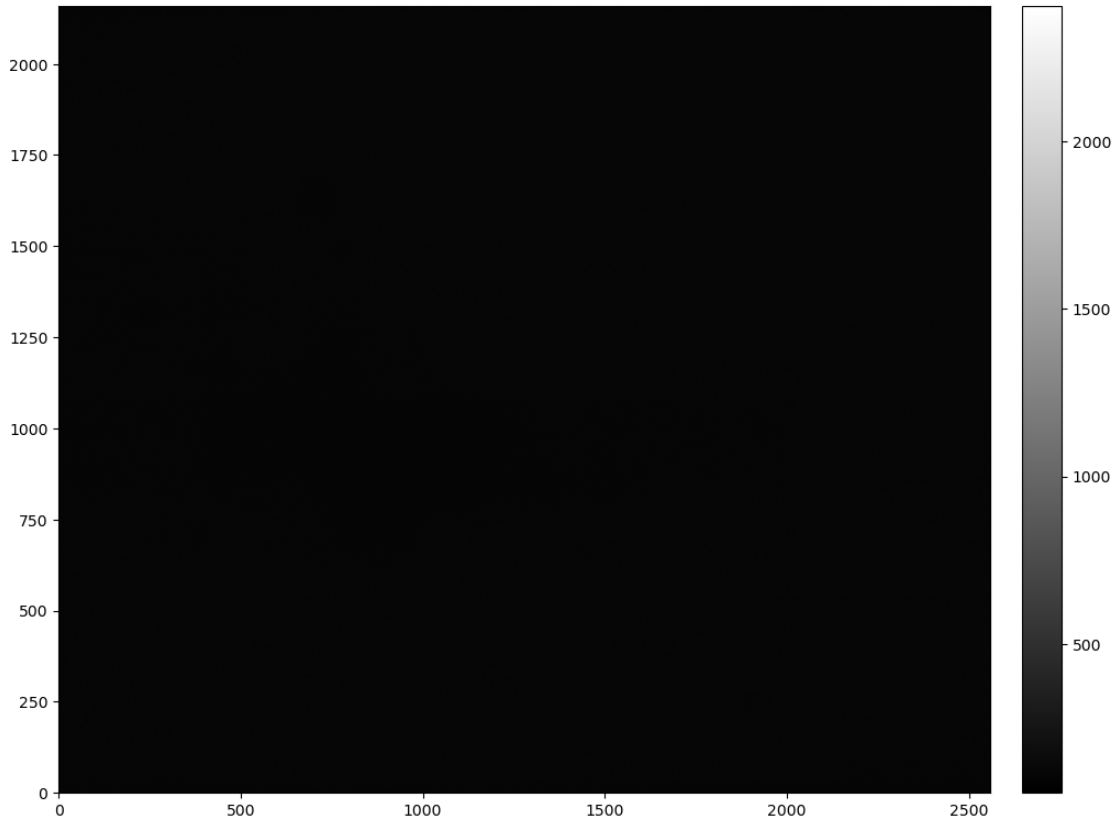
[8]:
```
flat1 = plt.imread(os.path.join(datapath,"proj","BBii_0031.tif"))
show2D(flat1)
```



[8]:
```
<cil.utilities.display.show2D at 0x7ff924c4b680>
```

We also load and display a dark field (image taken before projections and flat with source off, and sample out, to capture any background counts):

[9]:
```
dark1 = plt.imread(os.path.join(datapath,"proj","BBii_0002.tif"))
show2D(dark1)
```



[9]: <cil.utilities.display.show2D at 0x7ff924df6150>

Projections have 2160 rows and 2560 columns as seen by:

[10]:
```
proj1.shape
```

[10]: (2160, 2560)

For convenience sinograms for four selected slices have been extracted from the full 1500 projections and are provided as mat-files. We choose one and load it:

Load demo data set and display the first raw projection

[11]:
```
filename = "slice_0270_data.mat"
all_data = scipy.io.loadmat(os.path.join(datapath,filename))
```

The data contains projections, flats and darks for the selected slice. There are 1500 projections of size 2560 pixels:

[12]:
```
projs = all_data['X_proj'].astype(np.float32)
projs.shape
```

[12]:
```
(2560, 1500)
```

There are 200 flats (100 taken before, and 100 taken after the projections):

[13]:
```
flats = all_data['X_flat'].astype(np.float32)
flats.shape
```

[13]:
```
(2560, 200)
```

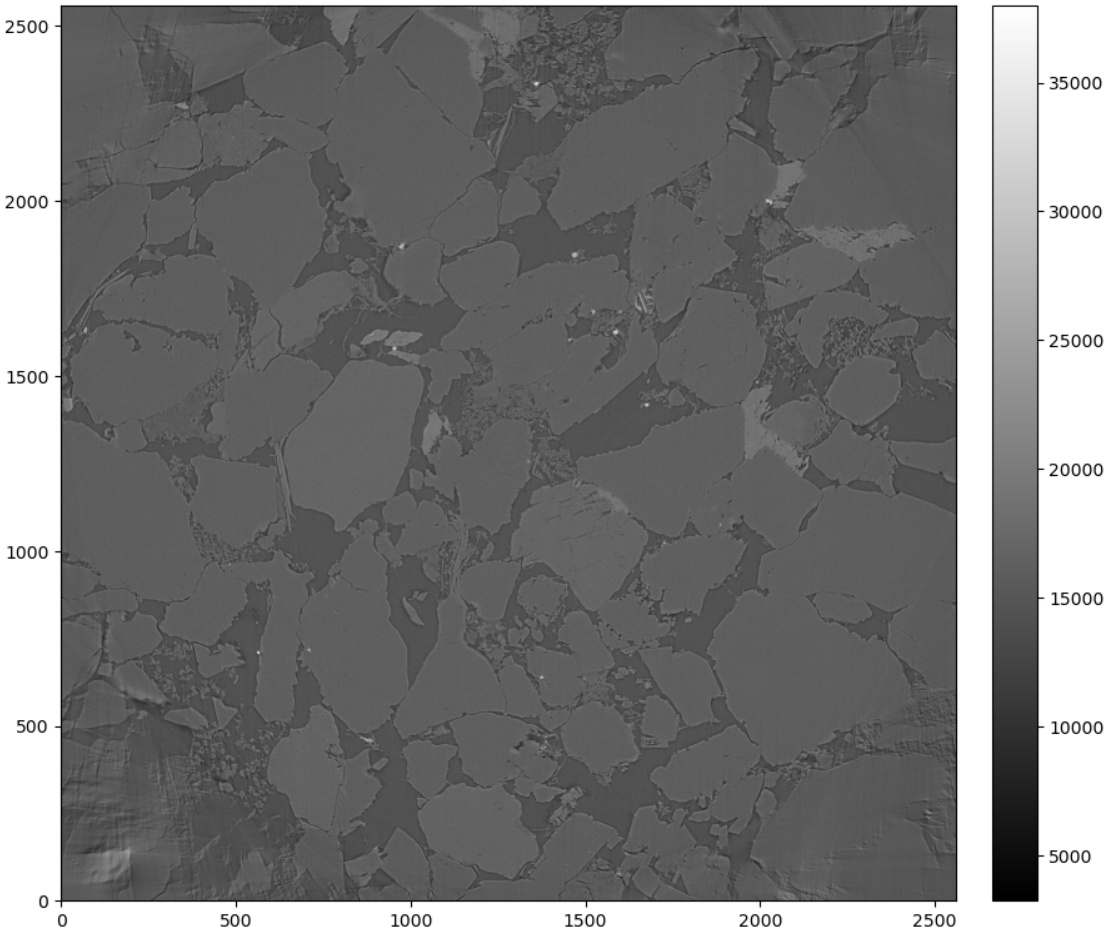
There are 30 darks taken at the beginning of the experiment:

[14]:
```
darks = all_data['X_dark'].astype(np.float32)
darks.shape
```

[14]:
```
(2560, 30)
```

The data provided also contains the reconstruction produced at the synchrotron where the data was acquired. We load and display it to see what kind of image we aim to reconstruct:

[15]:
```
vendor_recon = plt.imread(os.path.join(datapath,"recon","BBii_0270.rec.16bit.
↪tif"))
show2D(np.rot90(vendor_recon))
```



[15]:
```
<cil.utilities.display.show2D at 0x7ff924c7f230>
```

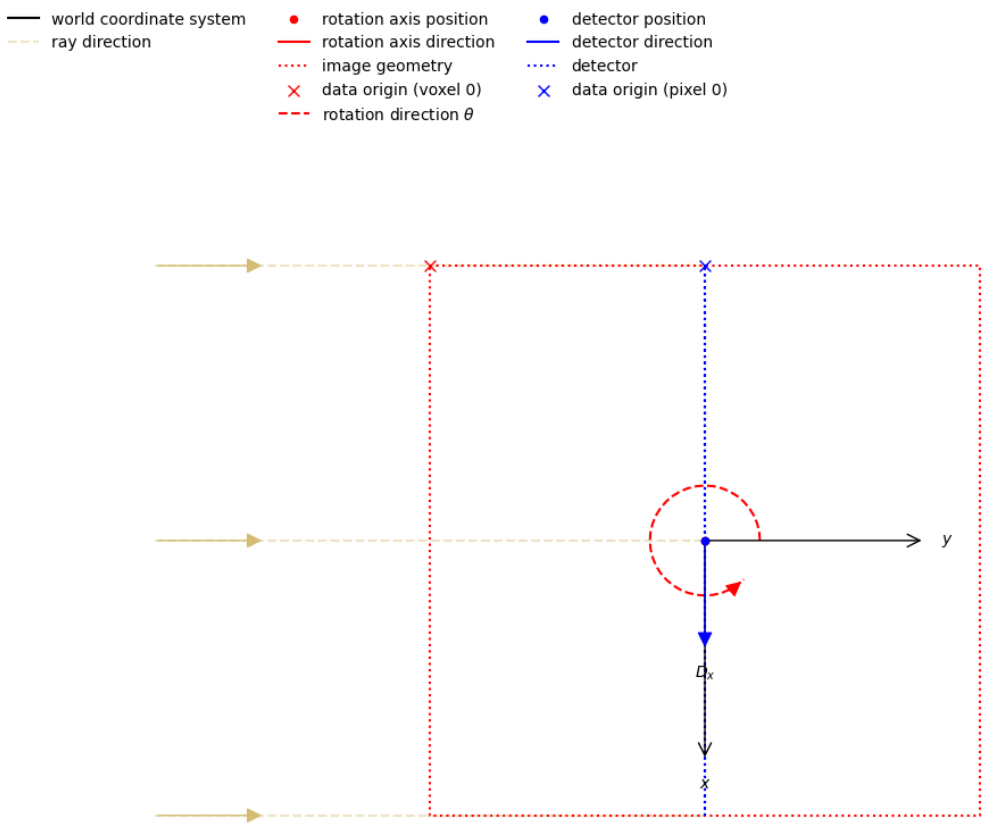
OK, we have now taken a look at the data and are ready to start producing our own reconstruction. We need to go through a number of steps to get there.

As a first step toward reconstructing the data, we specify the CIL `AcquisitionGeometry` for a 2D parallel-beam geometry with 1500 projections over 0 to 180 degrees each consisting of 2560 detector pixels.

[16]:
```
ag = AcquisitionGeometry.create_Parallel2D()  \
         .set_panel(num_pixels=(2560))        \
         .set_angles(angles=np.linspace(0,180,1500,endpoint=False))
```

We can illustrate the geometry specified:

[17]:
```
show_geometry(ag)
```



[17]:
```
<cil.utilities.display.show_geometry at 0x7ff91a4baf60>
```

To create the CIL data structure `AcquisitionData` holding the data we check again the size of the projections

[18]:
```
projs.shape
```

[18]:
```
(2560, 1500)
```

so along the first dimension are the horizontal detector pixels and along the second the projections/angles. We then tell CIL which axes are which:

[19]:
```
ag.set_labels(['horizontal','angle'])
print(ag.dimension_labels)
```

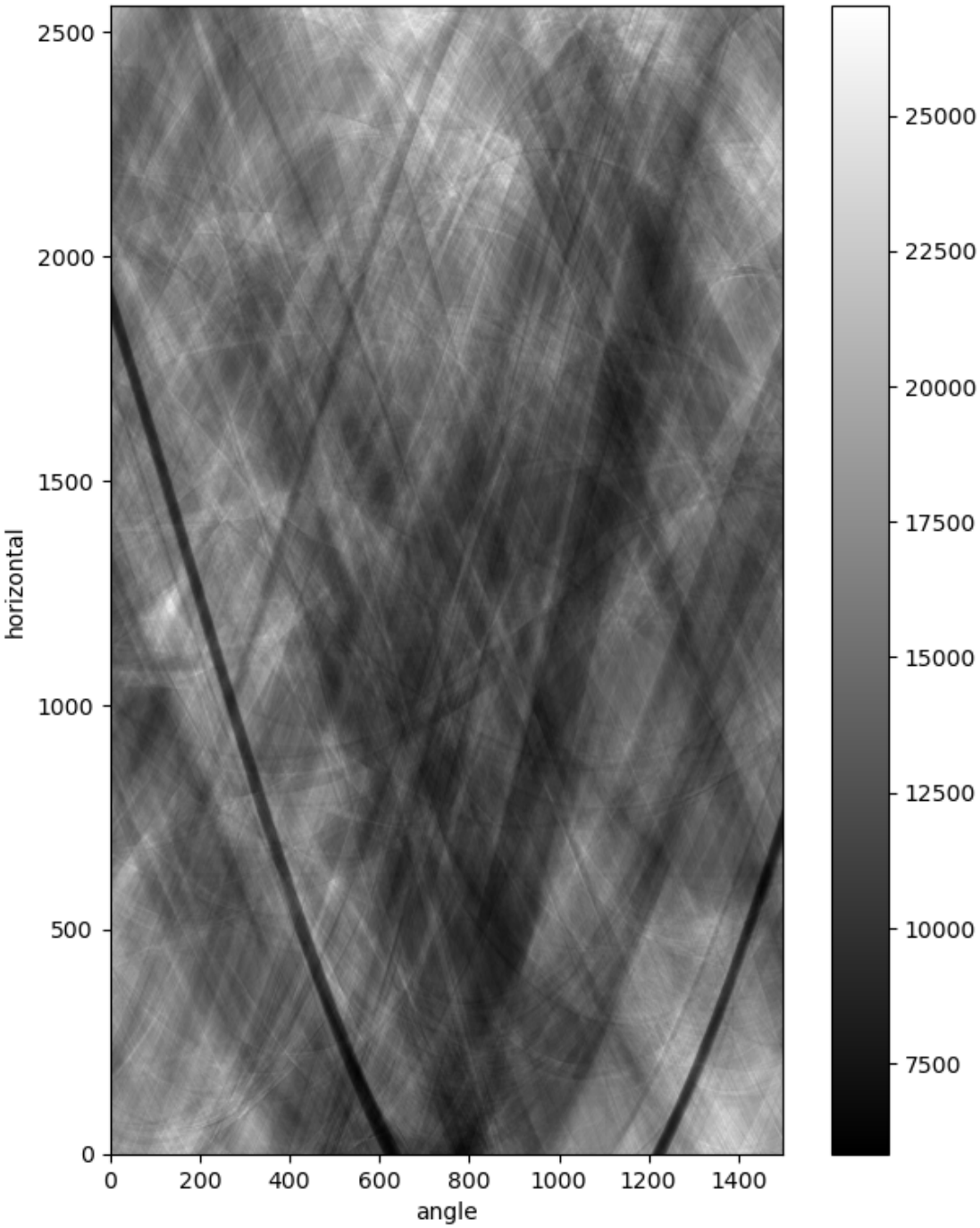
```
(<AcquisitionDimension.HORIZONTAL: 'horizontal'>, <AcquisitionDimension.ANGLE:
'angle'>)
```

Now we create an `AcquisitionData` holding the projections and the geometry. We link the `projs` numpy array to the `AcquisitionData` without creating an additional copy.

[20]:
```
data = AcquisitionData(projs, geometry=ag, deep_copy=False)
```

We can take a look with the CIL `show2D` display function:

[21]:
```
show2D(data)
```



[21]: `<cil.utilities.display.show2D at 0x7ff91a480dd0>`

Let us try reconstructing straight from the raw projections. First we need to make sure the data matches the order expected by the ASTRA-Toolbox plugin. We use `reorder('astra')` to check and reorder the data if required.

[22]:
```
data.reorder('astra')
```

Now we must specify the `ImageGeometry` we want for the reconstruction grid, here we choose the default which can be generated from the `AcquisitionGeometry`:

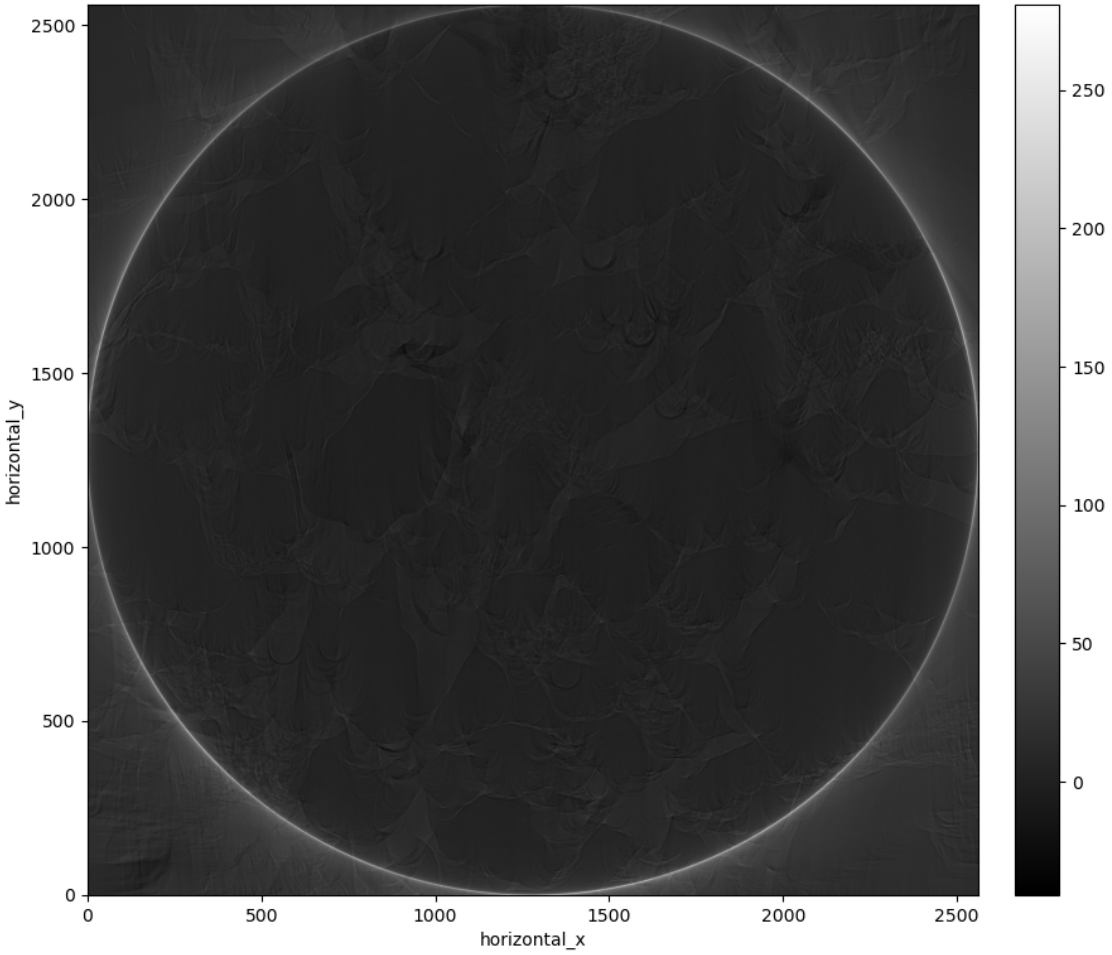
[23]:
```
ig = ag.get_ImageGeometry()
```

We set up and run the FBP reconstructor with our dataset and specify the backend to use:

[24]:
```
rec1 = FBP(data, ig, backend='astra').run()
```

```
FBP recon

Input Data:
        angle: 1500
        horizontal: 2560

Reconstruction Volume:
        horizontal_y: 2560
        horizontal_x: 2560

Reconstruction Options:
        Backend: astra
        Filter: ram-lak
        Filter cut-off frequency: 1.0
        FFT order: 13
        Filter_inplace: False
        Split processing: 0

Reconstructing in 1 chunk(s):
```

[25]:
```
show2D(rec1)
```

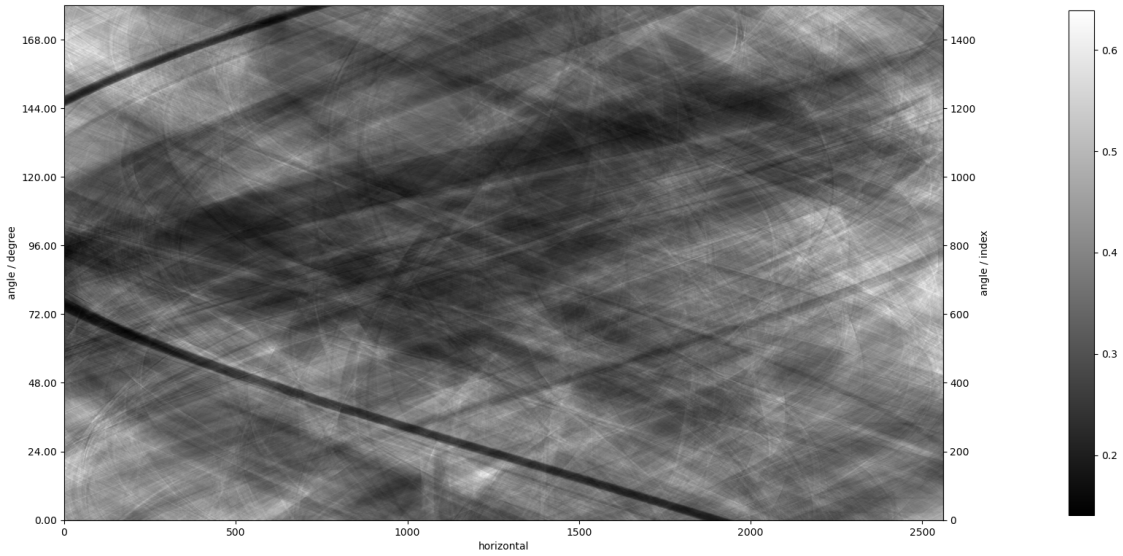


[25]: <cil.utilities.display.show2D at 0x7ff91a2e8140>

On close inspection we see some of the right features but distorted and wrong colours. The first thing we are missing is to normalise the data, i.e., apply flat and dark field correction. This is achieved by the CIL `Normaliser` processor and we simply use the mean over the flat and dark images respectively:

[26]:
```
data2 = Normaliser(flat_field=flats.mean(axis=1),
                   dark_field=darks.mean(axis=1)
                  )(data)
```

[27]:
```
show2D(data2)
```



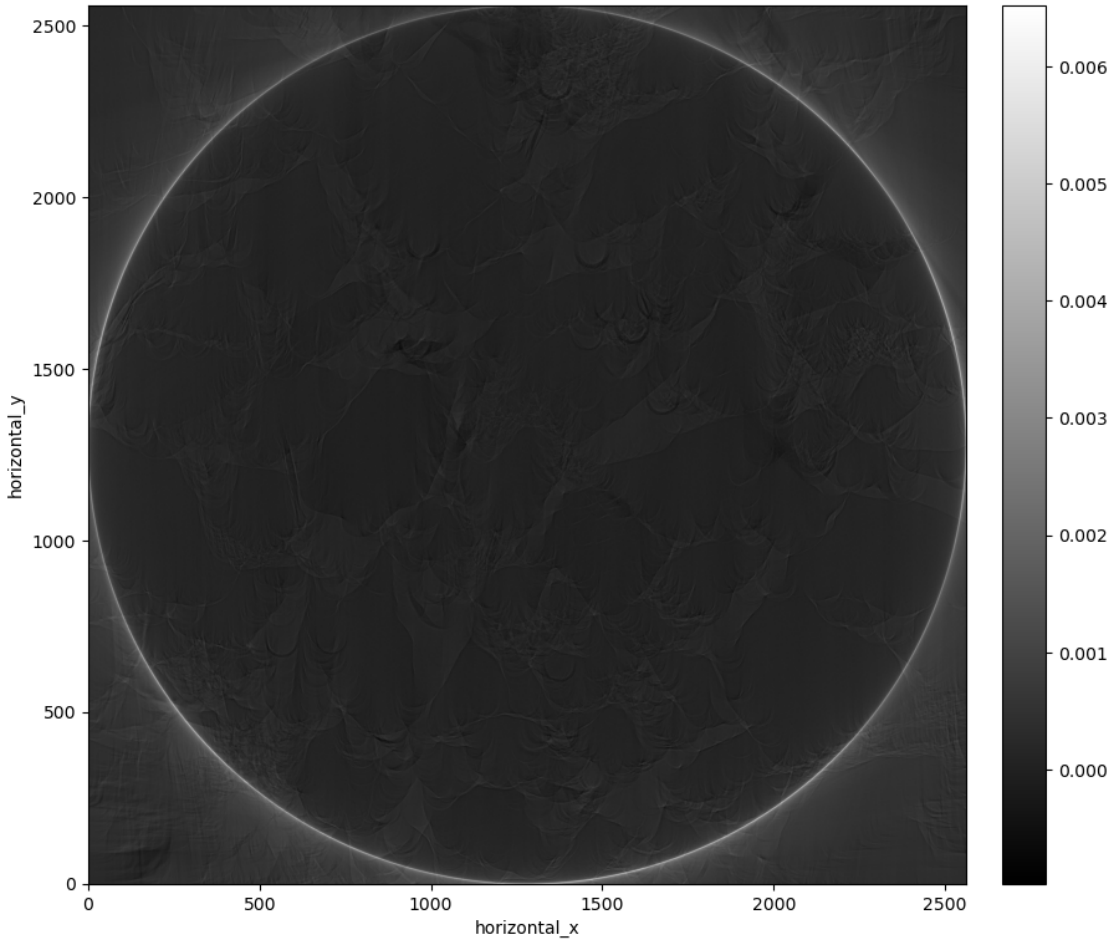
[27]: <cil.utilities.display.show2D at 0x7ff91a33bb30>

Compared to the previous sinogram of the raw projections, we see on the colourbar that the range is now within 0 to 1 as is what we need. We try reconstructing again:

[28]:
```
rec2 = FBP(data2, ig, backend='astra').run()
```

```
FBP recon

Input Data:
        angle: 1500
        horizontal: 2560

Reconstruction Volume:
        horizontal_y: 2560
        horizontal_x: 2560

Reconstruction Options:
        Backend: astra
        Filter: ram-lak
        Filter cut-off frequency: 1.0
        FFT order: 13
        Filter_inplace: False
        Split processing: 0

Reconstructing in 1 chunk(s):
```

[29]:
```
show2D(rec2)
```



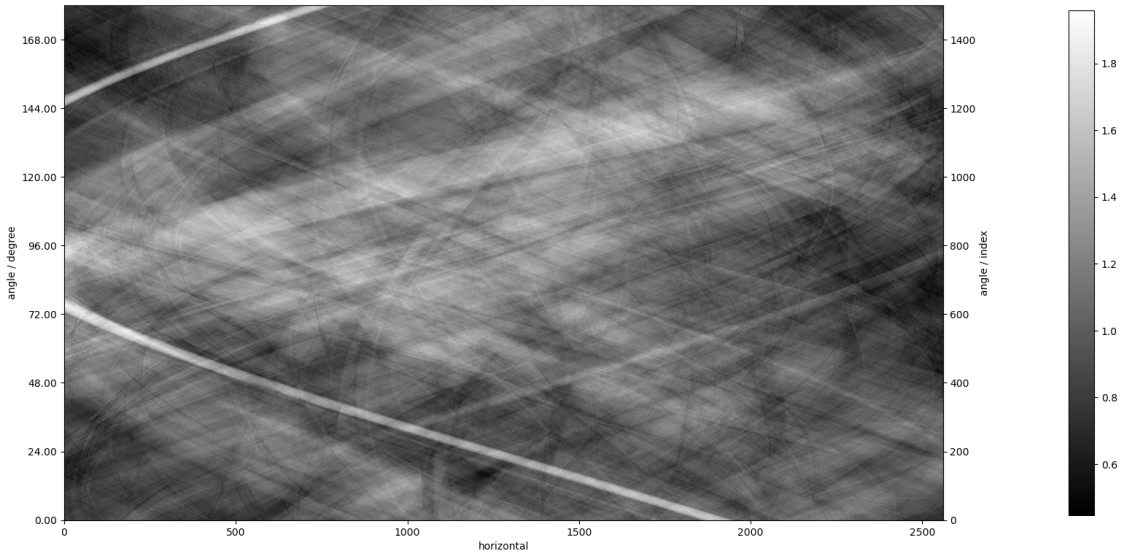
[29]: <cil.utilities.display.show2D at 0x7ff923073260>

It looks that same, only has the range change as seen on the colourbar. We realise that we need to apply the negative logarithm according to the Lambert-Beer law, which can be done manually or using a CIL `Processor`, which will prevent outliers and zeros from causing trouble:

[30]:
```
data3 = TransmissionAbsorptionConverter()(data2)
```

```
INFO:cil.processors.TransmissionAbsorptionConverter:
 Current min_intensity = 0.0: output may contain NaN or inf. Ensure your data
only contains positive values or set min_intensity to a small positive value.
```

[31]:
```
show2D(data3)
```



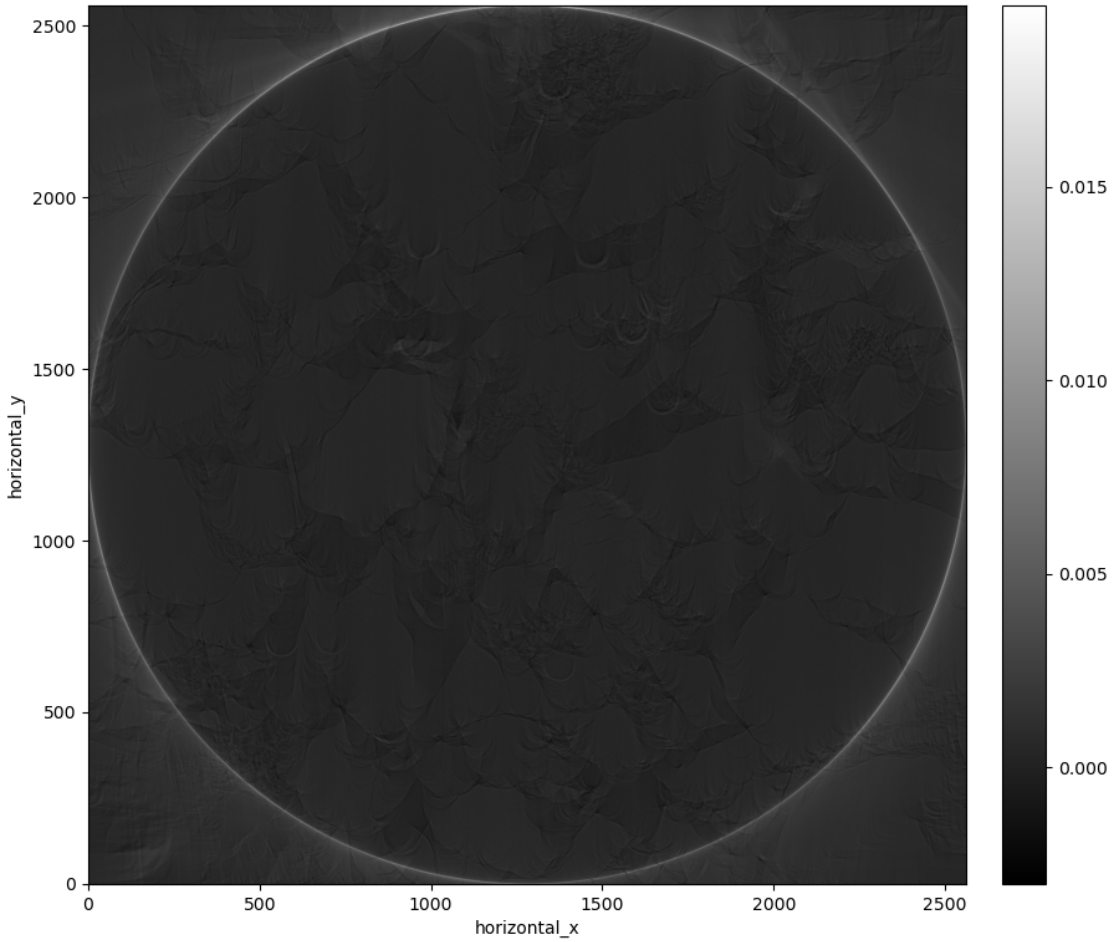
[31]: <cil.utilities.display.show2D at 0x7ff91a3de150>

We see the data has been transformed so the big sine band shows up white instead of dark as before. We attempt reconstructing:

[32]:
```
rec3 = FBP(data3, ig, backend='astra').run()
```

```
FBP recon

Input Data:
        angle: 1500
        horizontal: 2560

Reconstruction Volume:
        horizontal_y: 2560
        horizontal_x: 2560

Reconstruction Options:
        Backend: astra
        Filter: ram-lak
        Filter cut-off frequency: 1.0
        FFT order: 13
        Filter_inplace: False
        Split processing: 0

Reconstructing in 1 chunk(s):
```

[33]:
```
show2D(rec3)
```

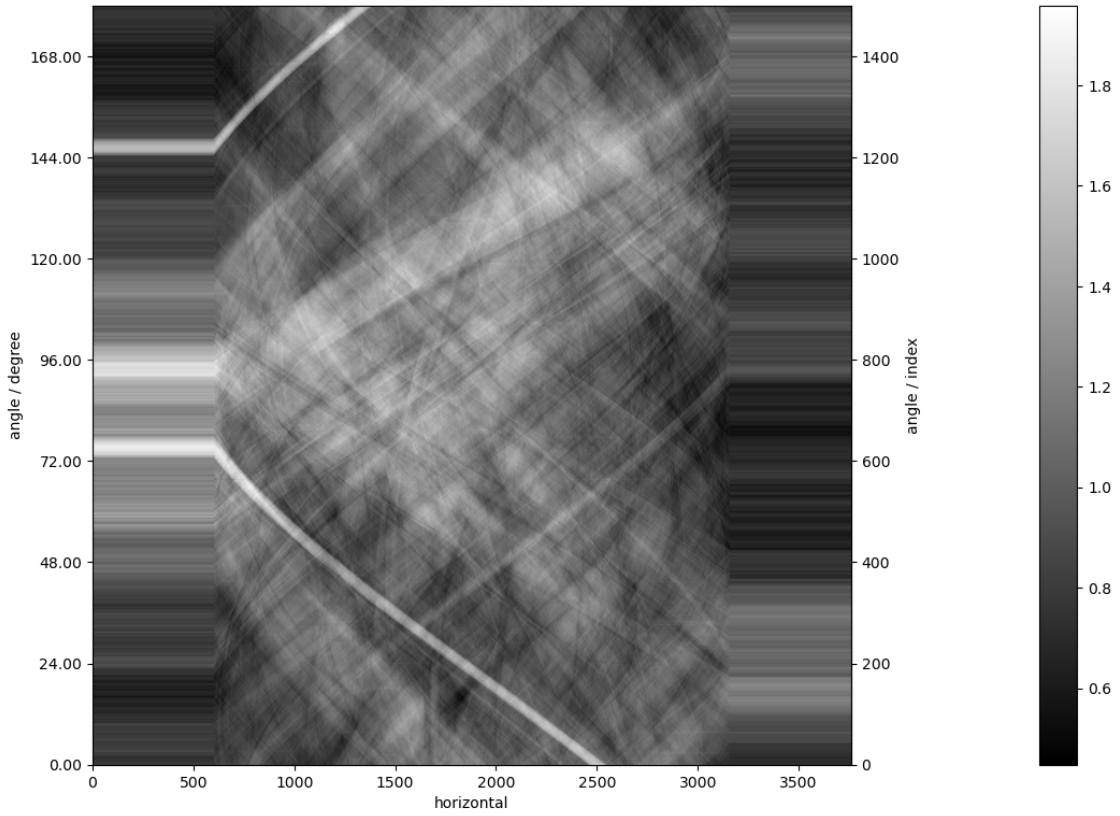


[33]: <cil.utilities.display.show2D at 0x7ff8e04349b0>

The colours have flipped now, so it is a step forward, but there is a big white ring. This is because the data is region-of-interest, i.e., the sample was larger than the field of view, so projections are truncated (i.e. do not have air on both sides as also seen in the sinograms). A simple way to compensate for this is to extend or pad the data on both sides of the projections. Using the CIL `Padder` processor we can for example pad by the left and rightmost pixel values, and we can play with the amount of padding required to push the ring out of the reconstruction. A `padsize` of about 600 is required:

[34]:
```
padsize = 600
data4 = Padder.edge(pad_width={'horizontal': padsize})(data3)
```

[35]:
```
show2D(data4)
```



[35]:
```
<cil.utilities.display.show2D at 0x7ff8e0533b30>
```

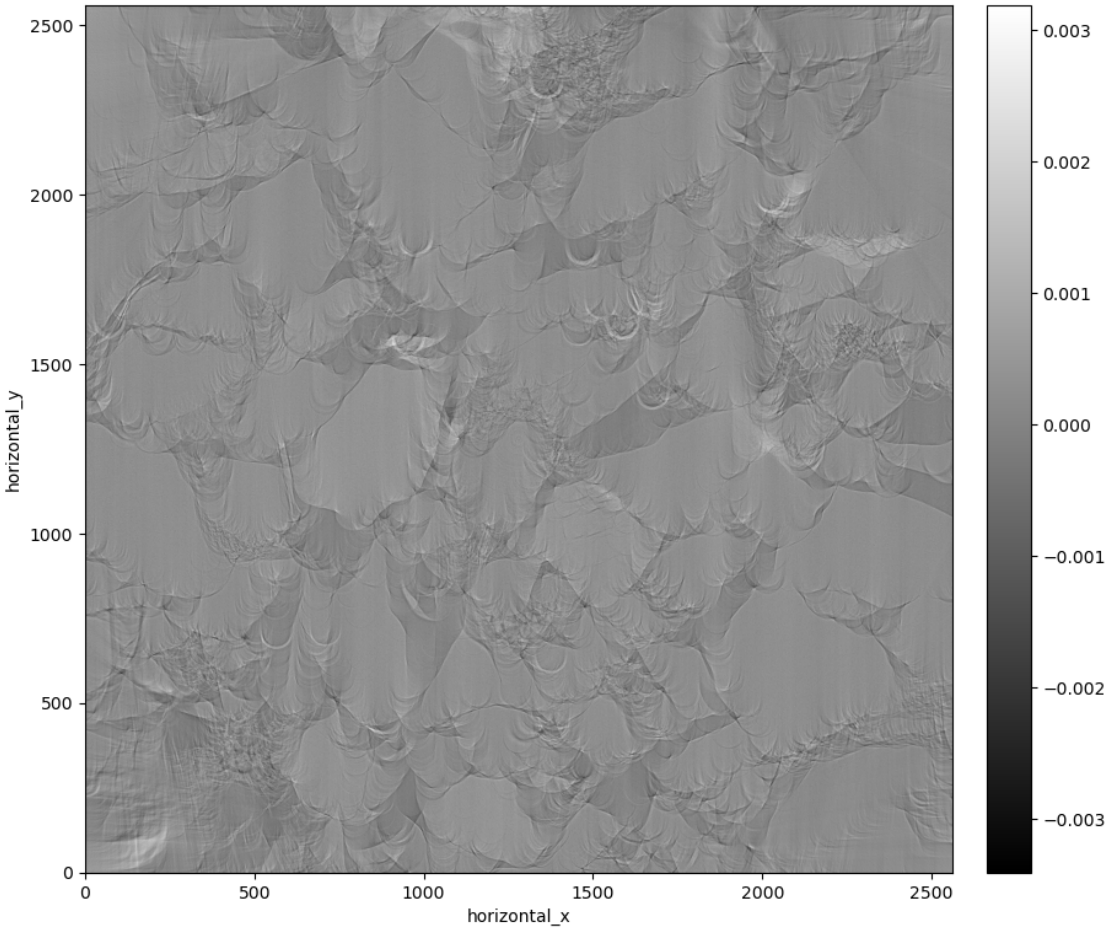
We see the data has been extended left and right. In effect the sinogram is now larger, so we need to create a new FBP reconstructor configured for the new padded data. We keep the reconstruction volume (defined by our image geometry) the same as before as we are not interested in the extended region.

[36]:
```
rec4 = FBP(data4, ig, backend='astra').run()
```

```
FBP recon

Input Data:
        angle: 1500
        horizontal: 3760

Reconstruction Volume:
        horizontal_y: 2560
        horizontal_x: 2560

Reconstruction Options:
        Backend: astra
        Filter: ram-lak
```

```
        Filter cut-off frequency: 1.0
        FFT order: 13
        Filter_inplace: False
        Split processing: 0

    Reconstructing in 1 chunk(s):
```

[37]:
```
show2D(rec4)
```



[37]:
```
<cil.utilities.display.show2D at 0x7ff8e0448050>
```

With `padsize=600` we see the region-of-interest ring has been successfully moved out and the sample features are more clearly seen. However there are still artifacts in the form of U-shaped stripes. These are centre-of-rotation artifacts caused by the sample not being perfectly centered during scanning. The log file of the dataset provides the centre value that was determined at the synchrotron. Here we use a CIL `Processor` to determine the offset and update the geometry. It works by doing FBP reconstructions for a range of offset parameters, evaluates a quality metric based on image sharpness and searches for the best offset. This technique is designed for use when

you have 360 degrees of data, but can be applied to 180 degrees but will be very sample dependent.

[38]:
```
data5 = CentreOfRotationCorrector.image_sharpness(backend='astra',␣
↪search_range=100, tolerance=0.1)(data4)
```

```
INFO:cil.processors.CofR_image_sharpness:evaluated 11 points

INFO:cil.processors.CofR_image_sharpness:evaluated 11 points

INFO:cil.processors.CofR_image_sharpness:Centre of rotation correction found
using image_sharpness

INFO:cil.processors.CofR_image_sharpness:backend FBP/FDK astra

INFO:cil.processors.CofR_image_sharpness:Calculated from slice: centre

INFO:cil.processors.CofR_image_sharpness:Centre of rotation shift = 44.489077
pixels

INFO:cil.processors.CofR_image_sharpness:Centre of rotation shift = 44.489077
units at the object

INFO:cil.processors.CofR_image_sharpness:Return new dataset with centred
geometry
```

We can compute the centre of rotation from a CIL geometry. The argument `distance_units` allows you to define the units of the returned value. Here we specify pixels.

[39]:
```
data5.geometry.get_centre_of_rotation(distance_units='pixels')
```

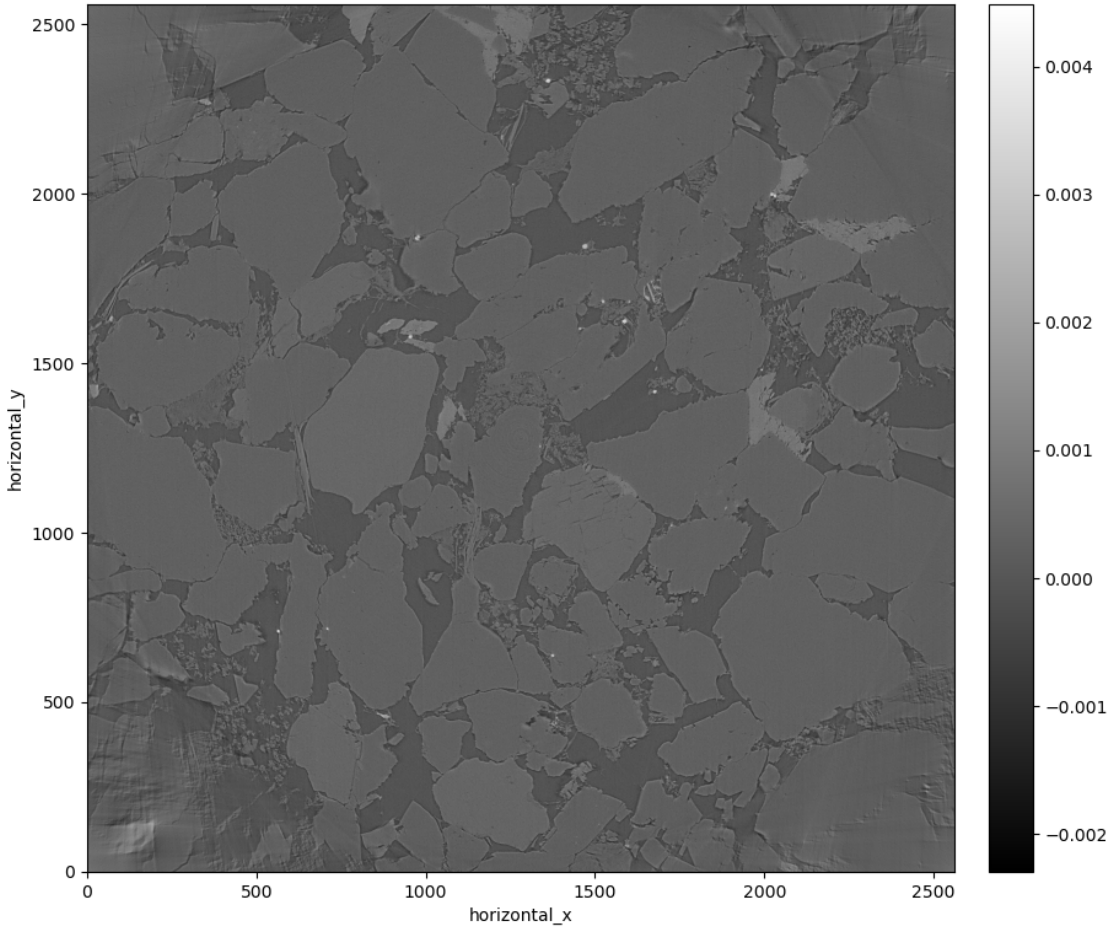
[39]:
```
{'offset': (44.48907676939449, 'pixels'), 'angle': (0.0, 'radian')}
```

[40]:
```
rec5 = FBP(data5, ig, backend='astra').run()
```

```
FBP recon

Input Data:
        angle: 1500
        horizontal: 3760

Reconstruction Volume:
        horizontal_y: 2560
        horizontal_x: 2560

Reconstruction Options:
        Backend: astra
        Filter: ram-lak
        Filter cut-off frequency: 1.0
        FFT order: 13
        Filter_inplace: False
        Split processing: 0
```

```
Reconstructing in 1 chunk(s):
```

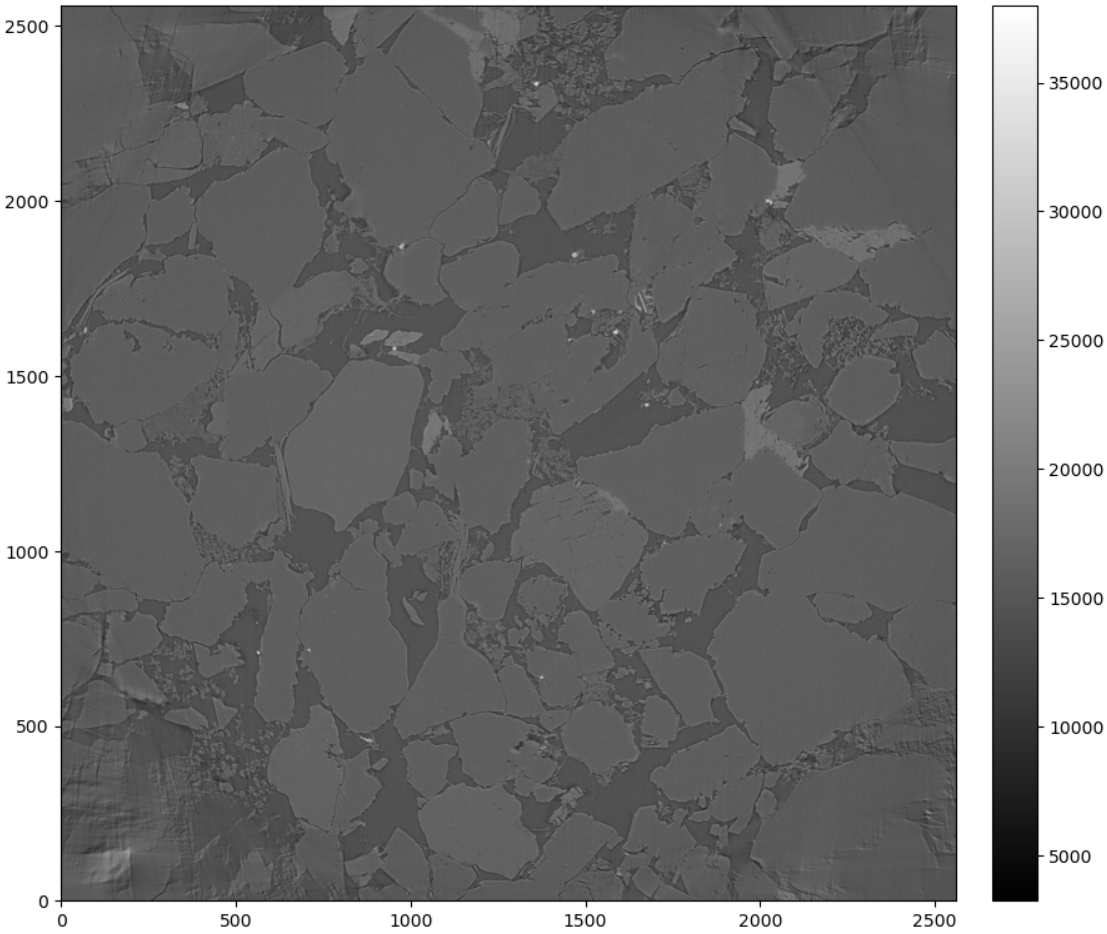
[41]:
```
show2D(rec5)
```



[41]:
```
<cil.utilities.display.show2D at 0x7ff8e0920800>
```

This reproduces the result from the synchrotron quite well, we show it again here for comparison:

[42]:
```
show2D(np.rot90(vendor_recon))
```



[42]:
```
<cil.utilities.display.show2D at 0x7ff8e0942ae0>
```

Should the centre of rotation correction method fail, one can manually specify a rotation axis offset and carry out reconstruction in the following way. In this way, one may experiment with different offsets and manually search for a suitable value by visual inspection of the resulting reconstructions as function of the offset.

Try to find the correct axis offset within the range -100 to 100 pixels.

Make a copy of the dataset first:

[43]:
```
data_cor_manual = data4.copy()
```

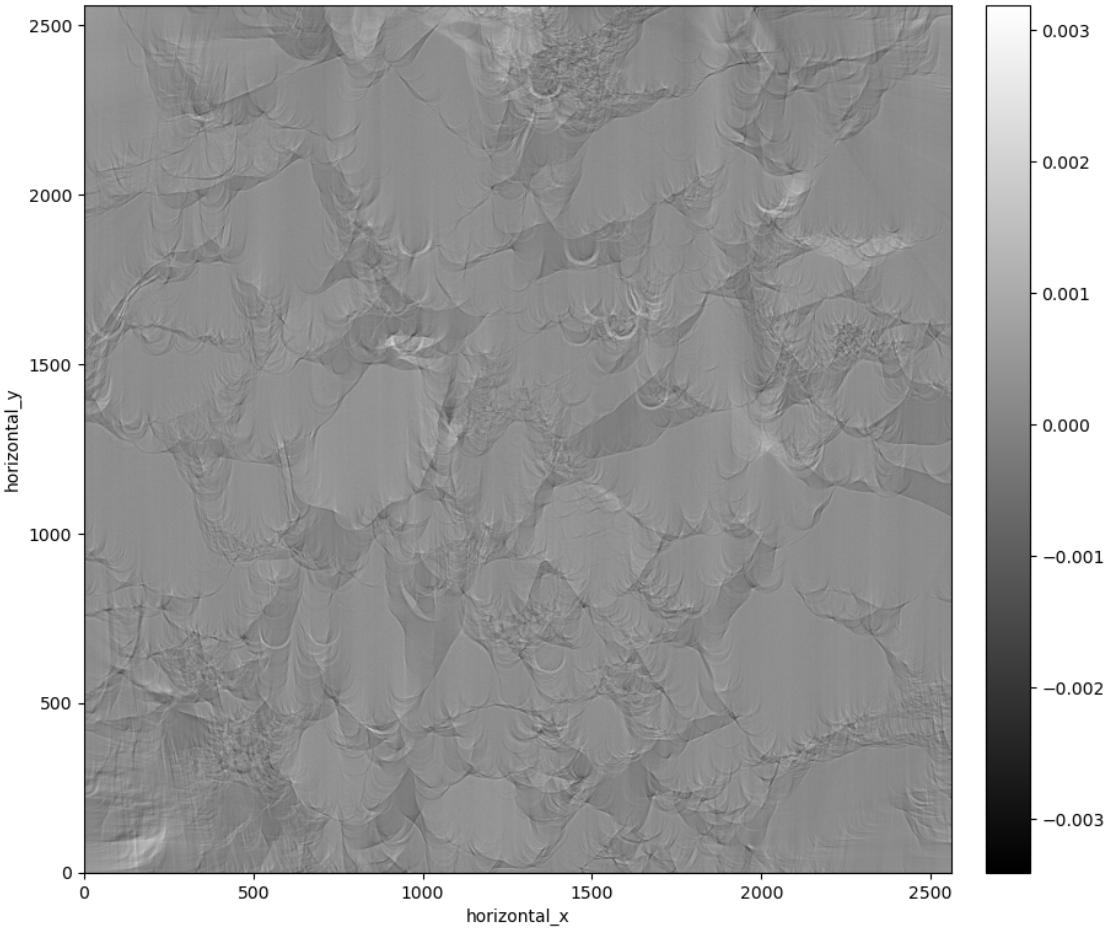
Then update the geometry with different centre of rotation offsets:

[44]:
```
offset = 0
data_cor_manual.geometry.set_centre_of_rotation(offset, distance_units='pixels')
```

Compute FBP reconstruction and display assessing the output by eye:

[45]:
```
rec_cor_manual = FBP(data_cor_manual, ig, backend='astra').run()

show2D(rec_cor_manual)
```

```
FBP recon

Input Data:
        angle: 1500
        horizontal: 3760

Reconstruction Volume:
        horizontal_y: 2560
        horizontal_x: 2560

Reconstruction Options:
        Backend: astra
        Filter: ram-lak
        Filter cut-off frequency: 1.0
        FFT order: 13
        Filter_inplace: False
        Split processing: 0

Reconstructing in 1 chunk(s):
```



[45]: <cil.utilities.display.show2D at 0x7ff8e056f4a0>