02_Dynamic_CT_tmp

August 19, 2025

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

[2]:
```
#  Copyright 2019 - 2022 United Kingdom Research and Innovation
#  Copyright 2019 - 2022 The University of Manchester
#
#  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:    Evangelos Papoutsellis (UKRI-STFC)
#                   Gemma Fardell (UKRI-STFC)
#                   Laura Murgatroyd (UKRI-STFC)
#                   Casper da Costa-Luis (UKRI-STFC)
```

1 Dynamic Sparse CT

In this demo, we focus on different reconstruction methods for sparse dynamic CT using an open-access dynamic dataset available from Heikkilä_et_al_Zenodo. The aim is to demonstrate how to increase the temporal resolution, or to reduce the radiation dose of a CT scan, without sacrificing the quality of the reconstructions.

The gel phantom simulates diffusion of liquids inside plant stems, namely the flow of iodine-based contrast agents used in high resolution tomographic X-ray imaging of plants. In order to test different reconstruction methods, this radiation resistant phantom with similar diffusion properties was constructed. For more information, please see Heikkilä_et_al.

1.1 Learning objectives

- Create a **2D + channels** acquisition geometry for the dynamic tomographic data. Channels correspond to different time steps (frames).
- Create **sparse data** using the `Slicer` processor.
- Run FBP reconstruction for every time-channel.
- Setup PDHG for 2 different regularisers:
  - Spatiotemporal Total Variation: regularisation applied equally along the spatial and temporal dimensions (channels).
  - **Directional Total Variation (dTV)**: regularisation applied for each channel using a reference image reference.

1.2 Data

This example requires `GelPhantomData_b4.mat` and `GelPhantom_extra_frames.mat` data from https://zenodo.org/record/3696817#.YKTRP5MzZp9

The direct download urls are: - https://zenodo.org/record/3696817/files/GelPhantomData_b4.mat - https://zenodo.org/record/3696817/files/GelPhantom_extra_frames.mat

Once downloaded update `path_common` to run the script.

[3]:
```
#Set the path to the directory containing the data
path_common = '/mnt/share/materials/SIRF/Fully3D/CIL/GelPhantom'
```

[4]:
```
# Import libraries
from cil.framework import AcquisitionGeometry
from cil.io import NEXUSDataWriter, NEXUSDataReader
from cil.optimisation.algorithms import PDHG
from cil.optimisation.operators import GradientOperator, BlockOperator
from cil.optimisation.functions import IndicatorBox, BlockFunction,␣
 ↪L2NormSquared, MixedL21Norm
from cil.plugins.astra import ProjectionOperator, FBP
from cil.plugins.ccpi_regularisation.functions import FGP_dTV
from cil.processors import Slicer
from cil.utilities.display import show2D, show_geometry
from cil.utilities.jupyter import islicer

import numpy as np
from tqdm.auto import trange
from utilities_dynamic_ct import read_frames, read_extra_frames
from IPython.display import clear_output
```

1.3 Data information: Gel phantom

The sample is an agarose-gel phantom, perfused with a liquid contrast agent in a 50ml Falcon test tube (ø 29 × 115mm). The aim of this experiment was to simulate diffusion of liquids inside plant

stems, which cannot withstand high radiation doses from a denser set of measurement angles. After the agarose solidified, five intact plastic straws were made into the gel and filled with 20% sucrose solution to guarantee the diffusion by directing osmosis to the gel body.

Every measurement consists of 360 projections with 282 detector bins obtained from a flat-panel circular-scan cone-beam microCT-scanner. Only the central slice is provided, resulting in a 2D **fanbeam geometry**. The primary measurements consisted of 17 consecutive time frames, with initial stage of no contrast agent followed by steady increase and diffusion into the gel body over time. Data is given in two different resolutions corresponding to reconstructions of size:

- 256 x 256: **GelPhantomData_b4.mat**
- 512 x 512: **GelPhantomData_b2.mat**

For this notebook, a 256x256 resolution is selected. In addition to the primary measurements, a more densely sampled measurements from the first time step and an additional 18th time step are provided in **GelPhantom_extra_frames.mat**

- Pre-scan: **720 projections**
- Post-scan: **1600 projections**

1.3.1 Load and read dynamic data (mat files)

[5]:
```
data_mat = "GelPhantomData_b4"
file_info = read_frames(path_common, data_mat)
```

From the `file_info` variable, we have all the information in order to define our acquisition geometry and create our CIL acquisition data.

Note that the pixel size of the detector is wrong. The correct pixel size should be doubled.

[6]:
```
# Get sinograms + metadata
sinograms = file_info['sinograms']
frames = sinograms.shape[0]
angles = file_info['angles']
distanceOriginDetector = file_info['distanceOriginDetector']
distanceSourceOrigin = file_info['distanceSourceOrigin']
# Correct the pixel size
pixelSize = 2*file_info['pixelSize']
numDetectors = file_info['numDetectors']
```

1.3.2 Exercise 1: Create acquisition and image geometries

For this dataset, we have a 2D cone geometry with 17 time channels. Using the metadata above, we can define the acquisition geometry `ag` with

```
ag = AcquisitionGeometry.create_Cone2D(source_position = [0, distanceSourceOrigin],
                                       detector_position = [0, -distanceOriginDetector])\
    .set_panel(numDetectors, pixelSize)\
    .set_channels(frames)\
```

```
    .set_angles(angles, angle_unit="radian")\
    .set_labels(['channel','angle', 'horizontal'])
```

For the image geometry `ig` we use the following code and crop our image domain to `[256,256]`:

```
ig = ag.get_ImageGeometry()
```

[ ]:

1.3.3 Exercise 1: Solution

[7]:
```
# Create acquisition + image geometries
ag = AcquisitionGeometry.create_Cone2D(source_position = [0,␣
 ↪distanceSourceOrigin],
                                       detector_position = [0,␣
 ↪-distanceOriginDetector])\
    .set_panel(numDetectors, pixelSize)\
    .set_channels(frames)\
    .set_angles(angles, angle_unit="radian")\
    .set_labels(['channel','angle', 'horizontal'])
ig = ag.get_ImageGeometry()
ig.voxel_num_x = 256
ig.voxel_num_y = 256
```

Then, we create an `AcquisitionData` by allocating space from the acquisition geometry `ag`. This is filled with every **sinogram per time channel**.

[8]:
```
data = ag.allocate()
for i in range(frames):
   data.fill(sinograms[i], channel=i)
```

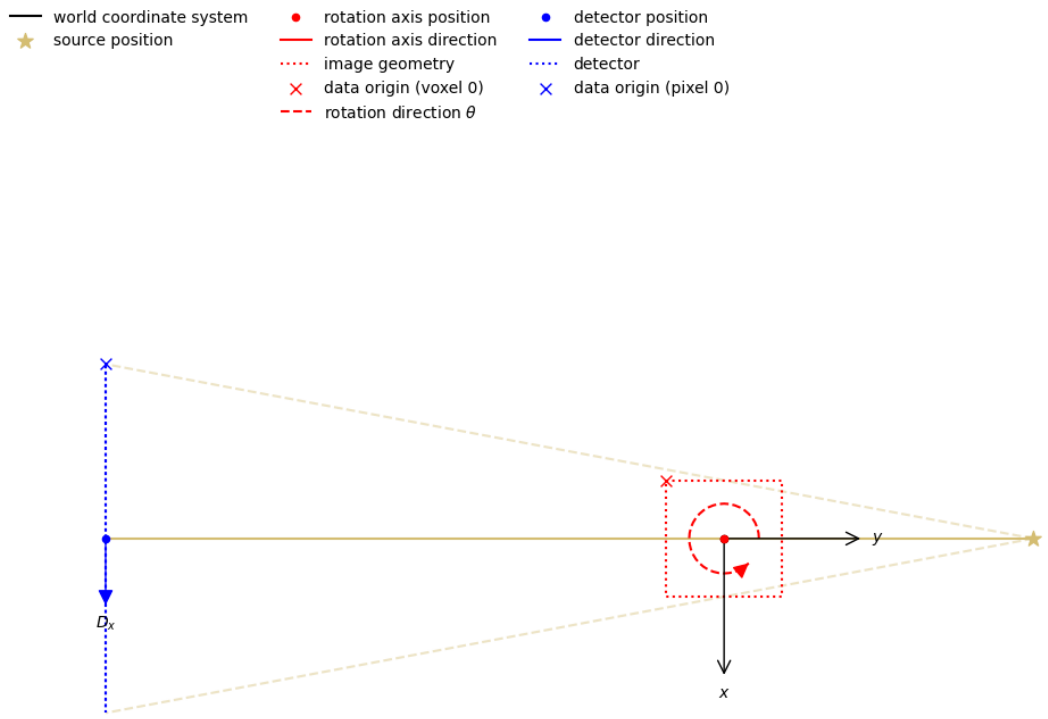
1.3.4 Show acquisition data

[9]:
```
islicer(data, direction=0, cmap="inferno", title="Time frame")
```

[9]:
```
HBox(children=(Output(), Box(children=(Play(value=8, interval=500, max=16),
VBox(children=(Label(value='Slice …
```

1.3.5 Show acquisition geometry

[10]:
```
show_geometry(ag);
```



1.3.6 Create Sparse CT data

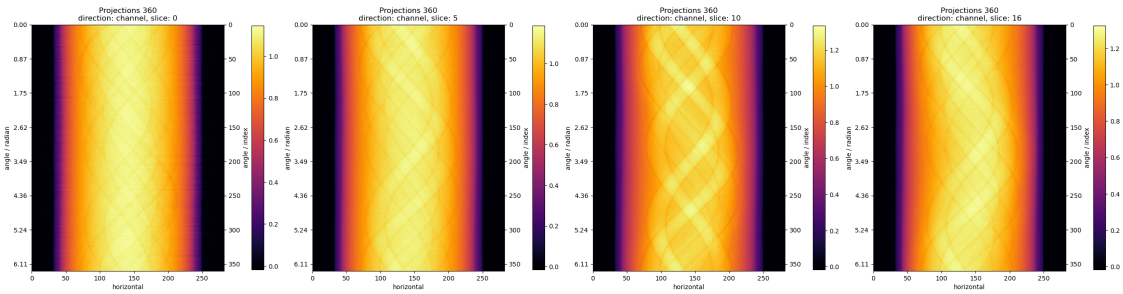
In order to simulate a sparse acquisition, from the total of 360 projections we select a number of projections depending on the size of the `step`:

- step = 1 –> 360/1 projections
- step = 5 –> 360/5 = 72 projections
- step = 10 –> 360/10 = 36 projections
- step = 20 –> 360/20 = 18 projections

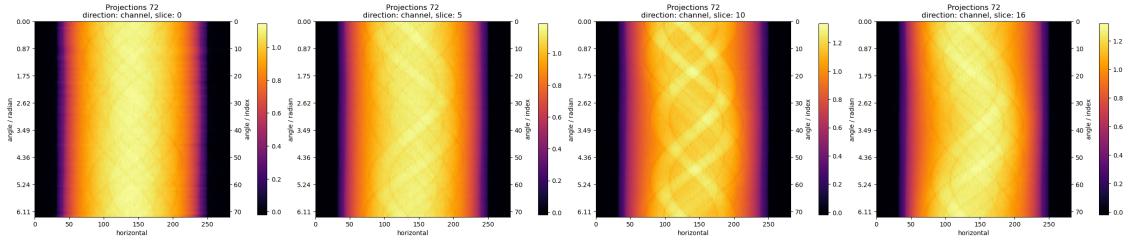
We create the sparse data using the `Slicer` processor. For every case, we show the dynamic data for 4 different time frames and save them using the `NEXUSDataWriter` processor in the `SparseData` directory.

[11]:
```
# Create and save Sparse Data with different angular sampling: 18, 36, 72, 360␣
 ↪projections
for step in (1, 5, 10, 20):
    name_proj = f"data_{360/step:.0f}"
    new_data = Slicer(roi={'angle':(0,360,step)})(data)
    ag = new_data.geometry

    show2D(new_data, slice_list=[0, 5, 10, 16], num_cols=4, origin="upper",
           cmap="inferno", title=f"Projections {360/step:.0f}", size=(25, 20))

    writer = NEXUSDataWriter(file_name=f"SparseData/{name_proj}.nxs",␣
 ↪data=new_data)
    writer.write()
```

```
New geometry:  2D Cone-beam tomography
System configuration:
        Source position: [ 0.     , 99.9995]
        Rotation axis position: [0., 0.]
        Detector position: [   0.      , -199.99975]
        Detector direction x: [1., 0.]
Panel configuration:
        Number of pixels: [282   1]
        Pixel size: [0.4 0.4]
        Pixel origin: bottom-left
Channel configuration:
        Number of channels: 17
Acquisition description:
        Number of positions: 360
        Angles 0-9 in radians: [0.         , 0.01745329, 0.03490658, 0.05235988,
0.06981317, 0.08726646,
 0.10471976, 0.12217305, 0.13962634, 0.15707964]
        Angles 350-359 in radians: [6.1086526, 6.126106 , 6.143559 , 6.161012 ,
6.1784654, 6.195919 ,
 6.213372 , 6.2308254, 6.2482786, 6.265732 ]
        Full angular array can be accessed with acquisition_data.geometry.angles
Distances in units: units distance
Shape out:  (17, 360, 282)
New geometry shape:  (17, 360, 282)
```

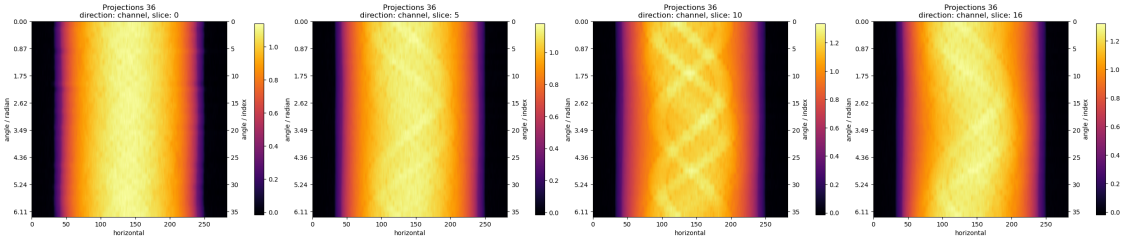


```
New geometry:  2D Cone-beam tomography
System configuration:
        Source position: [ 0.    , 99.9995]
        Rotation axis position: [0., 0.]
        Detector position: [   0.     , -199.99975]
        Detector direction x: [1., 0.]
Panel configuration:
        Number of pixels: [282   1]
        Pixel size: [0.4 0.4]
        Pixel origin: bottom-left
Channel configuration:
        Number of channels: 17
Acquisition description:
        Number of positions: 72
        Angles 0-9 in radians: [0.        , 0.08726646, 0.17453292, 0.2617994 ,
0.34906584, 0.43633232,
 0.5235988 , 0.61086524, 0.6981317 , 0.7853982 ]
        Angles 62-71 in radians: [5.4105206, 5.497787 , 5.5850534, 5.67232  ,
5.7595863, 5.846853 ,
 5.934119 , 6.021386 , 6.1086526, 6.195919 ]
        Full angular array can be accessed with acquisition_data.geometry.angles
Distances in units: units distance
Shape out:  (17, 72, 282)
New geometry shape:  (17, 72, 282)
```

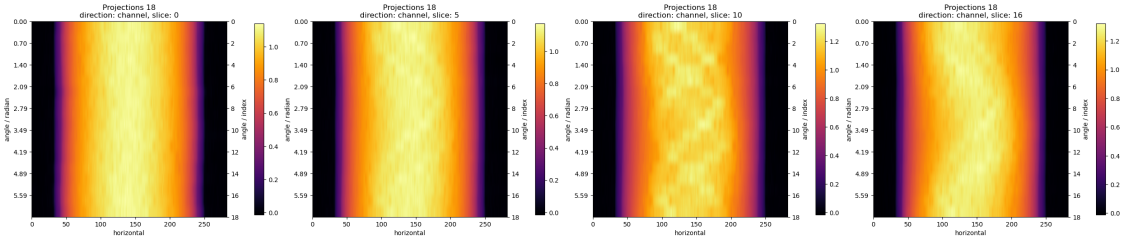



```
New geometry:  2D Cone-beam tomography
System configuration:
        Source position: [ 0.    , 99.9995]
        Rotation axis position: [0., 0.]
        Detector position: [   0.     , -199.99975]
        Detector direction x: [1., 0.]
Panel configuration:
        Number of pixels: [282   1]
        Pixel size: [0.4 0.4]
        Pixel origin: bottom-left
```

```
Channel configuration:
        Number of channels: 17
Acquisition description:
        Number of positions: 36
        Angles 0-9 in radians: [0.        , 0.17453292, 0.34906584, 0.5235988 ,
0.6981317 , 0.87266463,
 1.0471976 , 1.2217305 , 1.3962634 , 1.5707964 ]
        Angles 26-35 in radians: [4.537856 , 4.712389 , 4.886922 , 5.061455 ,
5.2359877, 5.4105206,
 5.5850534, 5.7595863, 5.934119 , 6.1086526]
        Full angular array can be accessed with acquisition_data.geometry.angles
Distances in units: units distance
Shape out:  (17, 36, 282)
New geometry shape:  (17, 36, 282)
```




```
New geometry:  2D Cone-beam tomography
System configuration:
        Source position: [ 0.    , 99.9995]
        Rotation axis position: [0., 0.]
        Detector position: [   0.     , -199.99975]
        Detector direction x: [1., 0.]
Panel configuration:
        Number of pixels: [282   1]
        Pixel size: [0.4 0.4]
        Pixel origin: bottom-left
Channel configuration:
        Number of channels: 17
Acquisition description:
        Number of positions: 18
        Angles 0-17 in radians: [0.        , 0.34906584, 0.6981317 , 1.0471976 ,
1.3962634 , 1.7453293 ,
 2.0943952 , 2.443461  , 2.7925267 , 3.1415927 , 3.4906585 , 3.8397243 ,
 4.1887903 , 4.537856  , 4.886922  , 5.2359877 , 5.5850534 , 5.934119  ]
Distances in units: units distance
Shape out:  (17, 18, 282)
New geometry shape:  (17, 18, 282)
```



1.3.7 Load sparse CT data

For the rest of the notebook, we use the sparse acquisition data `data_36`, i.e., only **36 projections** and perform the following:

- FBP reconstruction per time frame.
- Spatiotemporal TV reconstruction.
- Directional Total variation.

For the other cases, you can change the value of the `num_proj` below. Available data are: `data_18`, `data_36`, `data_72` and `data_360`.

[12]:
```
num_proj = 36
reader = NEXUSDataReader(file_name=f"SparseData/data_{num_proj}.nxs")
data = reader.load_data()
ag = data.geometry
```

1.4 Channelwise FBP

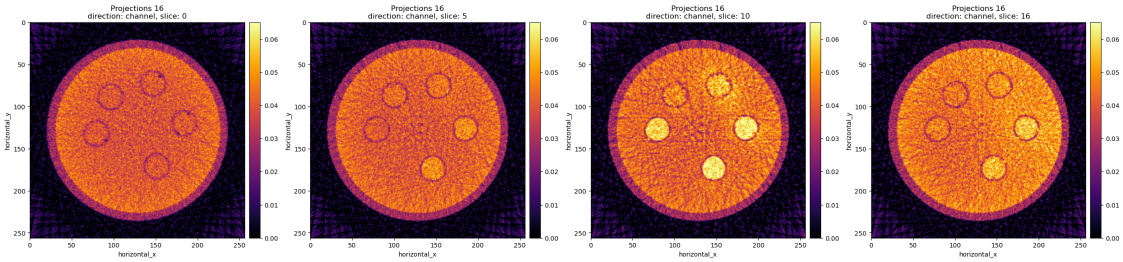
For the **channelwise** FBP reconstruction, we perform the following steps

- Allocate a space using the full image geometry (2D+channels) `ig` geometry.
- Extract the 2D acquisition and image geometries, using `ag.get_slice(channel=0)` and `ig.get_slice(channel=0)`.
- Run FBP reconstruction using the 2D sinogram data for every time frame.
- Fill the 2D FBP reconstruction with respect to the `channel=i` using the `fill` method.

[13]:
```
fbp_recon = ig.allocate()

ag2D = ag.get_slice(channel=0)
ig2D = ig.get_slice(channel=0)
fbp = FBP(ig2D,ag2D)

for i in trange(ig.channels, unit="timeframe"):
    data_single_channel = data.get_slice(channel=i)
    fbp.set_input(data_single_channel)
    fbp_recon.fill(fbp.get_output(), channel=i)
```

```
show2D(fbp_recon, slice_list=[0, 5, 10, 16], num_cols=4, origin="upper",␣
 ↪fix_range=(0,0.065),
       cmap="inferno", title=f"Projections {i}", size=(25, 20))

writer = NEXUSDataWriter(file_name=f"FBP_reconstructions/
 ↪FBP_projections_{num_proj}.nxs", data=fbp_recon)
writer.write()
```

```
  0%|          | 0/17 [00:00<?, ?timeframe/s]
```



1.4.1 Exercise 2: Total Variation reconstruction

For the TV reconstruction, we use the **Explicit formulation** of the PDHG algorithm. See the PDHG notebook for more information.

- Define the `ProjectionOperator` and the `GradientOperator` using `correlation=SpaceChannels`.

```
A = ProjectionOperator(ig, ag, 'gpu')
Grad = GradientOperator(ig, correlation = "SpaceChannels")
```

- Use the `BlockOperator` to define the operator K.

```
K = BlockOperator(A, Grad)
```

- Use the `BlockFunction` to define the function $\mathcal{F}$ that contains the fidelity term `L2NormSquared(b=data)` and the regularisation term `alpha_tv * MixedL21Norm()`, with `alpha_tv = 0.00063`. Finally, use the `IndicatorBox(lower=0.0)` to enforce a non-negativity constraint for the function $\mathcal{G}$.

```
F = BlockFunction(0.5*L2NormSquared(b=data), alpha_tv * MixedL21Norm())
G = IndicatorBox(lower=0)
```

```
[ ]:
```

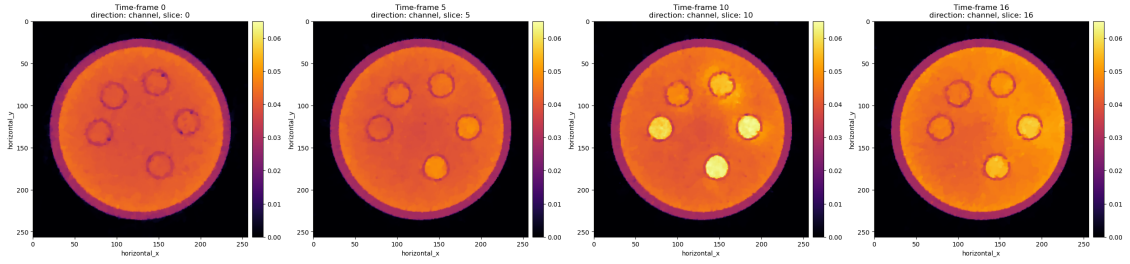
1.4.2 Exercise 2: Solution

[14]:
```
A = ProjectionOperator(ig, ag, 'gpu')
Grad = GradientOperator(ig, correlation="SpaceChannels")

K = BlockOperator(A, Grad)

alpha_tv = 0.00054
F = BlockFunction(0.5*L2NormSquared(b=data), alpha_tv * MixedL21Norm())
G = IndicatorBox(lower=0)

normK = K.norm()
sigma = 1./normK
tau = 1./normK

pdhg_tv = PDHG(f=F, g=G, operator=K, update_objective_interval=100)
pdhg_tv.run(500, verbose=1)

tv_recon = pdhg_tv.solution
```

```
  0%|          | 0/500 [00:00<?, ?it/s]
```

1.4.3 Show TV reconstruction for 4 different time frames

[15]:
```
titles_sinos = [f"Time-frame {i}" for i in (0, 5, 10, 16)]

show2D(tv_recon, slice_list = [0,5,10,16], num_cols=4,␣
 ↪origin="upper",fix_range=(0,0.065),
       cmap="inferno", title=titles_sinos, size=(25, 20))

writer = NEXUSDataWriter(file_name=f"TV_reconstructions/
 ↪TV_reconstruction_projections_{num_proj}.nxs", data=tv_recon)
writer.write()
```



1.5 Directional Total Variation

For our final reconstruction, we use a **structure-based prior**, namely the directional Total Variation (dTV) introduced in Ehrhardt MJ, Arridge SR.

In comparison with the Total variation regulariser,

$$\mathrm{TV}(u) = \|\nabla u\|_{2,1} = \sum |\nabla u\|_2,$$

in the Direction Total variation, a weight in front of the gradient is used based on a **reference image**. This acts as prior information from which edge structures are propagated into the reconstruction process. For example, an image from another modality, e.g., MRI, , can be used in the PET reconstruction, see Ehrhardt2016, Ehrhardt2016MRI. Another popular setup, is to use either both modalities or even channels in a joint reconstruction problem simultaneously, improving significantly the quality of the image, see for instance Knoll et al, Kazantsev_2018.

Definition: The dTV regulariser of the image u given the reference image v is defined as

$$d\mathrm{TV}(u, v) := \|D_v \nabla u\|_{2,1} = \sum_{i,j=1}^{M,N} \left(|D_v \nabla u|_2\right)_{i,j},$$

where the weight D_v depends on the normalised gradient ξ_v of the reference image v,

$$D_v = \mathbb{I}_{2\times 2} - \xi_v \xi_v^T, \quad \xi_v = \frac{\nabla v}{\sqrt{\eta^2 + |\nabla v|_2^2}}, \quad \eta > 0.$$

In this dynamic sparse CT framework, we apply the dTV regulariser for each time frame t which results to the following minimisation problem:

$$u_t^* = \underset{u}{\operatorname{argmin}} \frac{1}{2}\|A_{\mathrm{sc}} u_t - b_t\|^2 + \alpha\, d\mathrm{TV}(u_t, v_t),$$

where A_{sc}, b_t, u_t^*, denote the single channel `ProjectionOperator`, the sinogram data and the reconstructed image for the time frame t respectively.

1.5.1 Reference images

In terms of the reference images $(v_t)_{t=0}^{16}$, we are going to use the FBP reconstructions of the additional tomographic data. There are two datasets in `GelPhantom_extra_frames.mat` with dense sampled measurements from the first and last (18th) time steps:

- Pre-scan data with 720 projections
- Post-scan data with 1600 projections

We first read the matlab files and create the following acquisition data:

- data_pre_scan
- data_post_scan

1.5.2 Read matlab files for the extra frames

[16]:
```
data_mat_extra = "GelPhantom_extra_frames"

pre_scan_info = read_extra_frames(path_common, data_mat_extra,␣
 ↪"GelPhantomFrame1_b4")
```

```
post_scan_info = read_extra_frames(path_common, data_mat_extra,␣
 ↪"GelPhantomFrame18_b4")
```

1.5.3 Acquisition geometry for the 1st frame: 720 projections

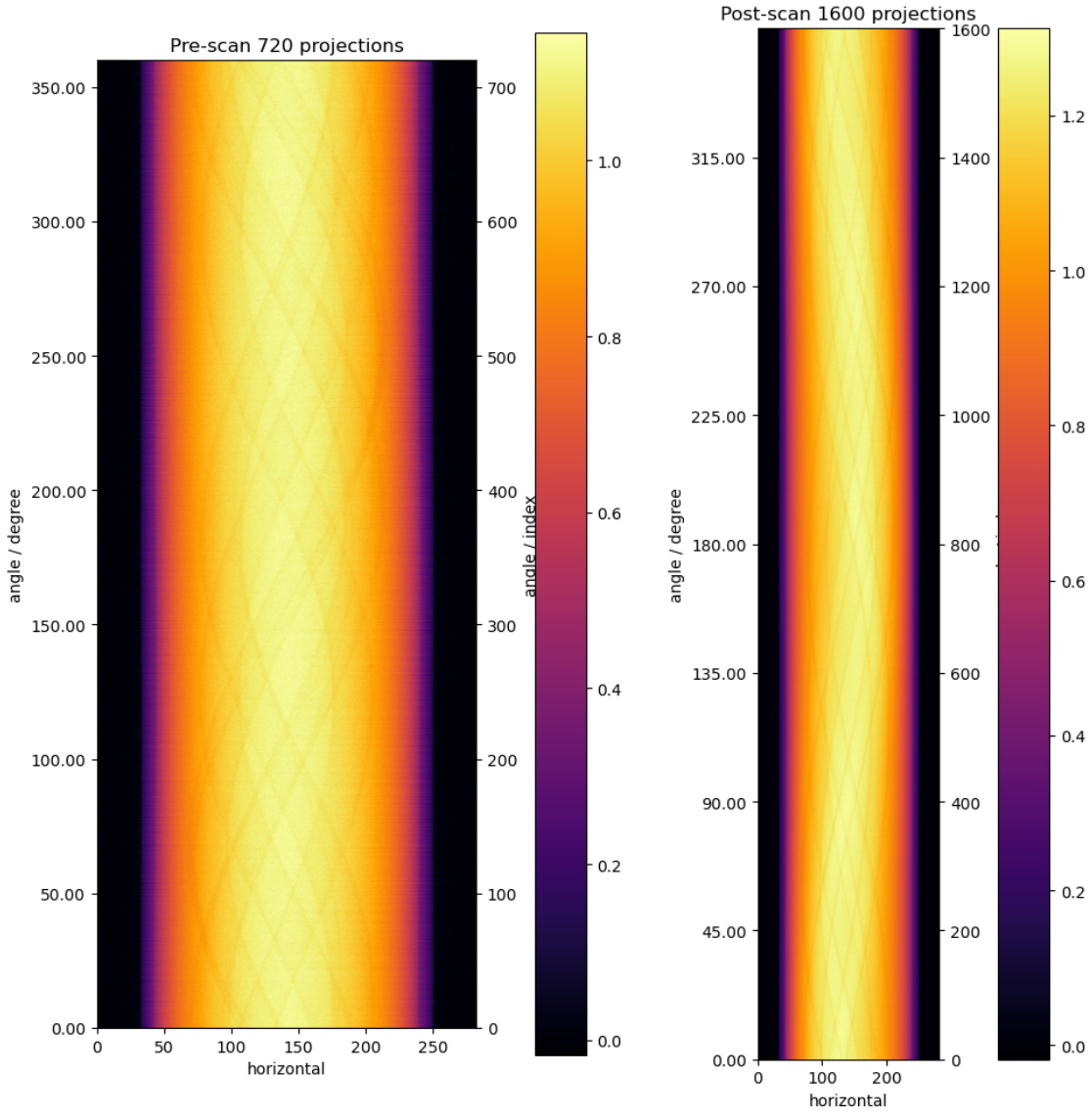
[17]:
```
ag2D_pre_scan = AcquisitionGeometry.create_Cone2D(source_position=[0,␣
 ↪pre_scan_info['distanceSourceOrigin']],
                                                  detector_position=[0,␣
 ↪-pre_scan_info['distanceOriginDetector']])\
    .set_panel(num_pixels = pre_scan_info['numDetectors'], pixel_size =␣
 ↪2*pre_scan_info['pixelSize'])\
    .set_angles(pre_scan_info['angles'])\
    .set_labels(['angle', 'horizontal'])
```

1.5.4 Acquisition geometry for the 18th frame: 1600 projections

[18]:
```
ag2D_post_scan = AcquisitionGeometry.create_Cone2D(source_position=[0,␣
 ↪post_scan_info['distanceSourceOrigin']],
                                                   detector_position=[0,␣
 ↪-post_scan_info['distanceOriginDetector']])\
    .set_panel(num_pixels = post_scan_info['numDetectors'], pixel_size =␣
 ↪2*post_scan_info['pixelSize'])\
    .set_angles(post_scan_info['angles'])\
    .set_labels(['angle', 'horizontal'])
```

1.5.5 Create Acquisition data: Pre-scan, Post-scan

[19]:
```
data_pre_scan = ag2D_pre_scan.allocate()
data_pre_scan.fill(pre_scan_info['sinograms'])

data_post_scan = ag2D_post_scan.allocate()
data_post_scan.fill(post_scan_info['sinograms'])

show2D([data_pre_scan,data_post_scan], title=["Pre-scan 720 projections",␣
 ↪"Post-scan 1600 projections"], cmap="inferno", size=(10,10));
```



1.5.6 FBP reconstruction (Reference images)

For the FBP reconstruction of the pre/post scan we use the 2D image geometry, `ig2D` and the corresponding acquisition geometries `ag2D_pre_scan` and `ag2D_post_scan`.

1.5.7 Exercise 3: Perform FBP to obtain the reference images

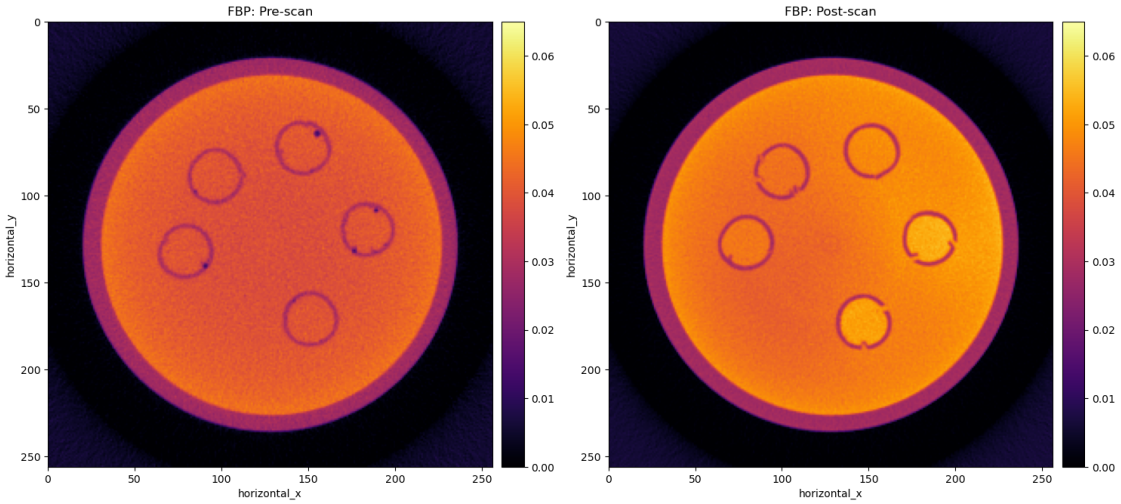
[ ]:

1.5.8 Exercise 3: Solution

[20]:
```
fbp_recon_pre_scan = FBP(ig2D, ag2D_pre_scan)(data_pre_scan)
fbp_recon_post_scan = FBP(ig2D, ag2D_post_scan)(data_post_scan)
```

1.5.9 Show and save the reference images

[21]:
```
show2D([fbp_recon_pre_scan,fbp_recon_post_scan],
       title=["FBP: Pre-scan", "FBP: Post-scan"], cmap="inferno",␣
 ↪origin="upper", fix_range=(0,0.065))

writer = NEXUSDataWriter(file_name=f"FBP_reconstructions/FBP_pre_scan.nxs",␣
 ↪data=fbp_recon_pre_scan)
writer.write()

writer = NEXUSDataWriter(file_name=f"FBP_reconstructions/FBP_post_scan.nxs",␣
 ↪data=fbp_recon_post_scan)
writer.write()
```



1.5.10 Edge information from the normalised gradient ξ_v

In the following we compute the normalised gradient ξ_v for the two reference images using different η values:

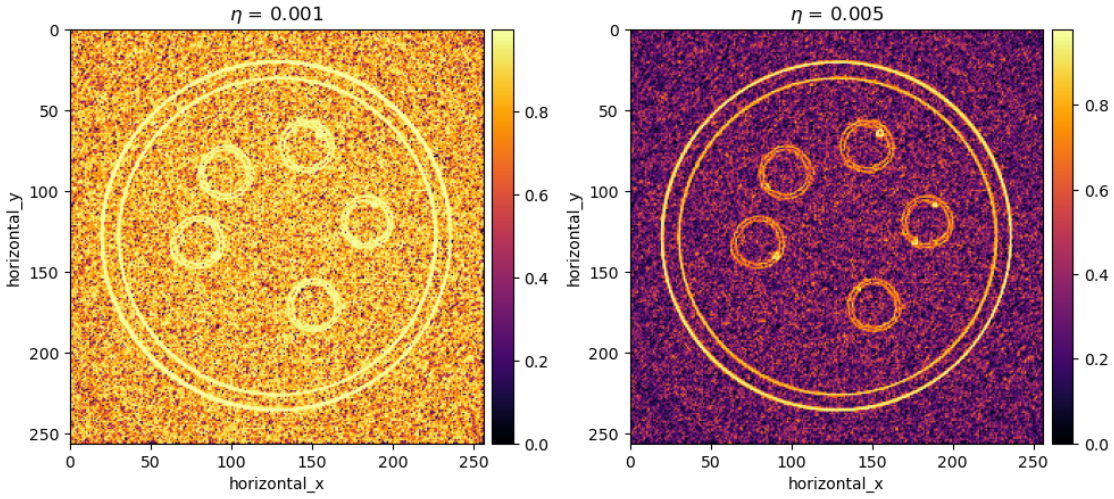
$$\xi_v = \frac{\nabla v}{\sqrt{\eta^2 + |\nabla v|_2^2}}, \quad \eta > 0$$

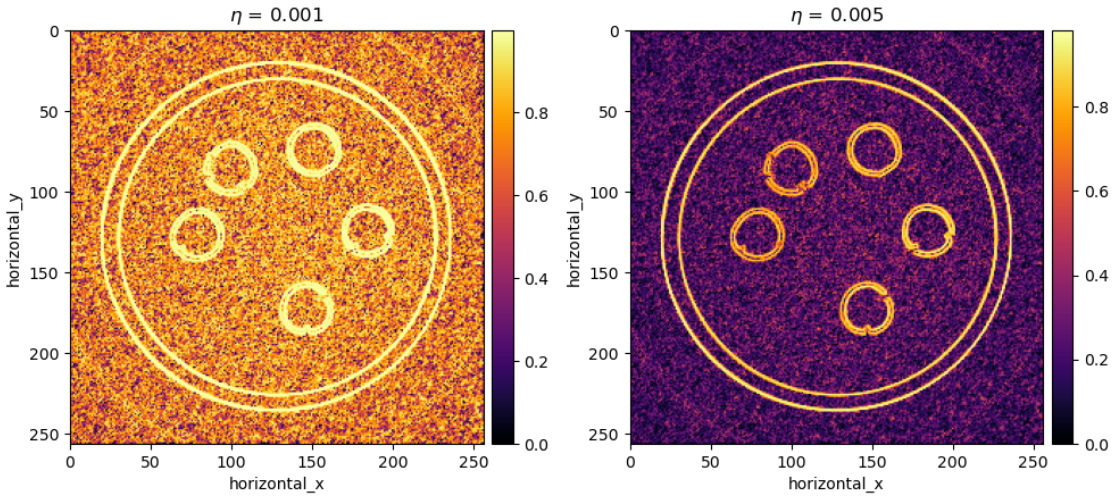
[22]:
```
def xi_vector_field(image, eta):
    ig = image.geometry
```

```
    ig.voxel_size_x = 1.
    ig.voxel_size_y = 1.
    G = GradientOperator(ig)
    numerator = G.direct(image)
    denominator = np.sqrt(eta**2 + numerator.get_item(0)**2 + numerator.
 ↪get_item(1)**2)
    xi = numerator/denominator

    return (xi.get_item(0)**2 + xi.get_item(1)**2).sqrt()

etas = [0.001, 0.005]

xi_post_scan = []
xi_pre_scan = []

for i in etas:
    xi_post_scan.append(xi_vector_field(fbp_recon_post_scan, i))
    xi_pre_scan.append(xi_vector_field(fbp_recon_pre_scan, i))
```

1.5.11 Edge information from the pre-scan and post-scan reference images

[23]:
```
title_etas = [f"$\\eta$ = {eta}" for eta in etas]
show2D(xi_pre_scan, cmap="inferno", title=title_etas, origin="upper",␣
 ↪num_cols=2, size=(10,10))
show2D(xi_post_scan, cmap="inferno", title=title_etas, origin="upper",␣
 ↪num_cols=2,size=(10,10));
```





1.6 Directional Total variation reconstruction

In total we have 17 time frames, and we need 17 reference images. Due to a slight movement of the sample at the beginning of the experiment, we apply the pre-scan reference image for the first time frame and use the post-scan reference image for the remaining time frames.

One could apply other configurations for the reference image in the intermediate time frames. For example, in order to reconstruct the $(t+1)$th time frame, one could use the tth time frame reconstruction as reference. A more sophisticated reference selection approach is applied in hyperspectral computed tomography in Kazantsev_2018.

1.6.1 Setup and run the PDHG algorithm

In order to solve the Dynamic dTV problem, we use the implicit formulation of the PDHG algorithm, where the **Fast Gradient Projection** algorithm, under the **dTV** regulariser, is used for the inner proximal problem.

- We first define the single slice `ProjectionOperator` using the 2D image and acquisition geometries and compute the `sigma` and `tau` stepsizes.

```
K = ProjectionOperator(ig2D, ag2D, 'gpu')

normK = K.norm()
sigma = 1./normK
tau = 1./normK
```

- We allocate space for the dTV reconstruction, i.e., `dtv_recon` using the full image geometry `ig`.
- Use the following values: `max_iteration=100`, `alpha_dtv = 0.0072`, `eta=0.005`.

- Loop over all the time frames (`tf`) and update :
  - the fidelity term `0.5 * L2NormSquared(b=data.subset(channel=tf)` for the function $\mathcal{F}$,
  - the regularisation term ($\mathcal{G}$) using the `FGP_dTV` function class from the CIL plugin of the CCPi-Regularisation Toolkit. For `tf=0` the pre-scan reference is used and for `tf>0` the post-scan reference is used.

1.6.2 Define single slice projection operator

[24]:
```
K = ProjectionOperator(ig2D, ag2D, 'gpu')
```

1.6.3 Parameters for the dTV regulariser and the PDHG algorithm

[25]:
```
normK = K.norm()
sigma = 1./normK
tau = 1./normK

dtv_recon = ig.allocate()

max_iterations = 100
alpha_dtv = 0.0072
eta = 0.005
```

1.6.4 Loop over all channels and update:

- the acquisition data in the `L2NormSquared` fidelity term,
- the reference image in the `FGP_dTV` regulariser.

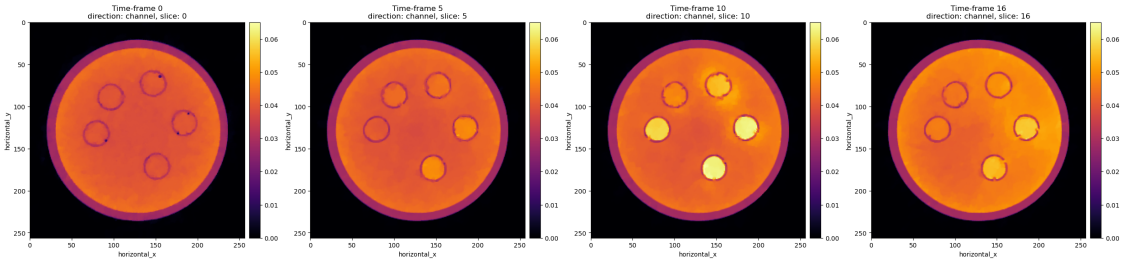
[26]:
```
for tf in trange(ig.channels, unit="timeframe"):
    F = 0.5 * L2NormSquared(b=data.get_slice(channel=tf))
    G = alpha_dtv * FGP_dTV(reference=(fbp_recon_pre_scan if tf==0 else␣
 ↪fbp_recon_post_scan), eta=eta, device='gpu')

    pdhg_dtv = PDHG(f=F, g=G, operator=K, tau=tau, sigma=sigma,␣
 ↪update_objective_interval=1000)
    pdhg_dtv.run(max_iterations, verbose=0)
    clear_output(wait=True)

    dtv_recon.fill(pdhg_dtv.solution, channel=tf)
```

```
  0%|          | 0/17 [00:00<?, ?timeframe/s]
```

```
/home/bgb37495/miniconda3/envs/cil_test_demos/lib/python3.12/site-
packages/cil/plugins/ccpi_regularisation/functions/regularisers.py:412:
UserWarning: FGP_dTV: the __call__ method is not implemented. Returning NaN.
  warnings.warn("{}: the __call__ method is not implemented. Returning
NaN.".format(self.__class__.__name__))
```

```
/home/bgb37495/miniconda3/envs/cil_test_demos/lib/python3.12/site-
packages/cil/plugins/ccpi_regularisation/functions/regularisers.py:433:
UserWarning: FGP_dTV: the convex_conjugate method is not implemented. Returning
NaN.
  warnings.warn("{}: the convex_conjugate method is not implemented. Returning
NaN.".format(self.__class__.__name__))
```

[27]:
```
show2D(dtv_recon, slice_list = [0,5,10,16], num_cols=4,␣
↪origin="upper",fix_range=(0,0.065),
       cmap="inferno", title=titles_sinos, size=(25, 20));
```



1.7 FBP vs TV vs dTV reconstructions vs FBP (360 projections).

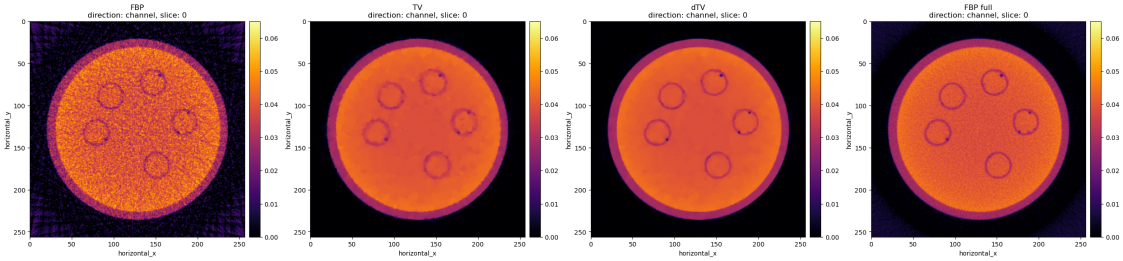
For our final comparison, we reconstruct the full dataset, i.e., 360 projections and compare it with the FBP, TV and dTV reconstruction with 36 projection.

[28]:
```
# Load data with 360 projections
reader = NEXUSDataReader(file_name="SparseData/data_360.nxs")
data_360 = reader.load_data()
ag2D = data_360.geometry.get_slice(channel=0)

# Perform channelwise FBP reconstruction
fbp_recon_360 = ig.allocate()
fbp = FBP(ig2D, ag2D)
for i in range(ig.channels):
    data_single_channel = data_360.get_slice(channel=i)
    fbp.set_input(data_single_channel)
    fbp_recon_360.fill(fbp.get_output(), channel=i)
```

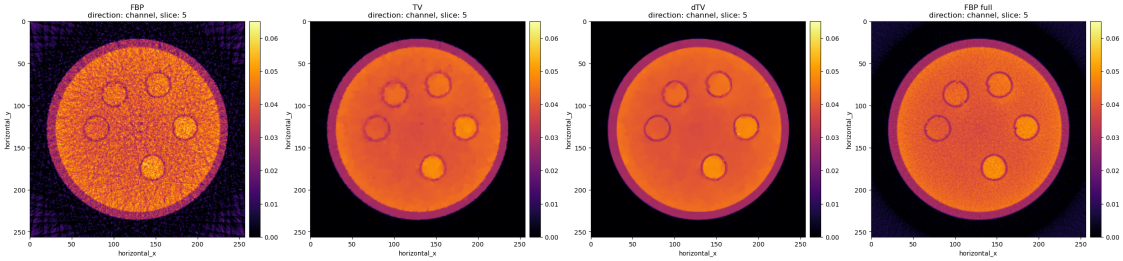
1.7.1 Show FBP, TV, dTV with 36 projections and FBP with 360 projections.

[29]:
```
for time_frame in (0, 5, 10, 16):
    print(f"Time-frame {time_frame}")
    show2D([fbp_recon, tv_recon, dtv_recon, fbp_recon_360],␣
↪slice_list=time_frame, num_cols=4, origin="upper", fix_range=(0,0.065),
           cmap="inferno", title=['FBP', 'TV', 'dTV', 'FBP full'], size=(25,␣
↪20))
```

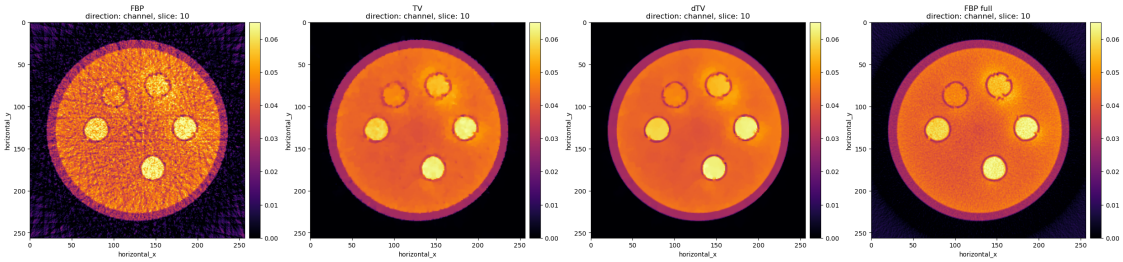
Time-frame 0



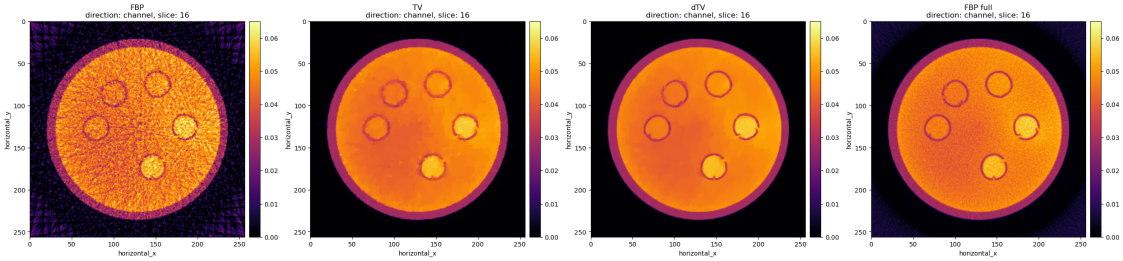
Time-frame 5



Time-frame 10



Time-frame 16



1.7.2 Exercise 4

As an additional **Exercise**, you can try other configurations, namely the datasets `data_18` and `data_72` by changing the number of projections. The optimal regularisation parameters for spatiotemporal TV and dTV are reported in Table 1 in Papoutsellis et al.

1.8 Conclusions

In this notebook, we presented three different reconstruction methods for undersampled dynamic tomographic data. The **channelwise FBP** and the **Spatiotemporal TV** and **Directional TV** regularisation. We focused on the reconstruction of the dataset with 36 projections out of 360 projections.