# 02_tikhonov_block_framework_tmp

August 19, 2025

[1]:

```python
import warnings
warnings.simplefilter('error', RuntimeWarning)
```

[2]:

```python
# -*- coding: utf-8 -*-
#  Copyright 2019 - 2024 United Kingdom Research and Innovation
#  Copyright 2019 - 2022 The University of Manchester
#  Copyright 2019 - 2024 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)
#                   Margaret Duff (UKRI-STFC)
#                   Hannah Robarts (UKRI-STFC)
```

### 0.1 Tikhonov regularisation using CGLS and block framework

This exercise introduces Tikhonov regularisation and explains how this is implemented in the CIL framework using the so-called block framework.

In a previous exercise, it was seen how CGLS could be used to determine a reconstruction based on the least squares reconstruction problem. It was seen that in case of noisy data, the least squares solution obtained by running until convergence is not desirable due to a high amount of noise. The number of iterations was seen to have a regularising effect, with the smooth, low-frequency components of the image recovered in the first iterations, while high-frequency components of the image such as edges were recovered later. Unfortunately, noise also kicks in, and one needs to pick the number of iterations that best balances the sharpness and amount of noise. As such, the regularising effect is implicitly obtained by choosing the number of iterations to run and never actually running until converged to the least squares solution.

Tikhonov regularisation is more explicit in that a regularisation term is added to the least squares fitting term, specifically a squared 2-norm. This problem should now be solved to convergence instead of using the number of iterations as implicit regularising effect. Instead, a parameter, the regularisation parameter, balances the emphasis on fitting the data and enforcing the regularity and must be chosen to provide the best trade-off.

Tikhonov regularisation tends to offer reduction of noise in the reconstruction, at the price of some blurring. This will be seen in what follows.

To set up Tikhonov problems we need to represent block matrices and concatenate data. In CIL we can do this using BlockOperator and BlockDataContainer as demonstrated in the exercise.

**Learning objectives:** 1. Construct and manipulate BlockOperators and BlockDataContainer, including direct and adjoint operations and algebra. 2. Use Block Framework to solve Tikhonov regularisation with CGLS algorithm. 3. Apply Tikhonov regularisation to tomographic reconstruction and explain the effect of regularisation parameter and operator in regulariser.

First, all imports required are carried out. This includes tools from the cil.framework and cil.optimisation modules, as well as test image generation tools in the tomophantom library and standard imports such as numpy.

[3]:

```
# CIL core components needed
from cil.framework import ImageGeometry, AcquisitionGeometry, BlockDataContainer

# CIL optimisation algorithms and linear operators
from cil.optimisation.algorithms import CGLS
from cil.optimisation.operators import BlockOperator, GradientOperator,␣
 ↪IdentityOperator, FiniteDifferenceOperator

# CIL example synthetic test image
from cil.utilities.dataexample import SHAPES

# CIL display tools
from cil.utilities.display import show2D, show1D, show_geometry

# Forward/backprojector from CIL ASTRA plugin
from cil.plugins.astra import ProjectionOperator

# For shepp-logan test image in CIL tomophantom plugin
from cil.plugins import TomoPhantom as cilTomoPhantom

# Third-party imports
import numpy as np
import matplotlib.pyplot as plt
```

### 0.1.1 Setting up a simulated 2D dataset

A 2D parallel beam case will be simulated. We start by creating a test image and will use the classic Shepp-Logan Phantom with 1024x1024 pixels on the square domain $x : [-1, 1]$, $y : [-1, 1]$. We set up the `ImageGeometry` to specify the dimensions and pixel size of the image:

[4]:

```
# Set up image geometry
n = 256
ig = ImageGeometry(voxel_num_x=n,
                   voxel_num_y=n,
                   voxel_size_x=2/n,
                   voxel_size_y=2/n)
print(ig)
```

```
Number of channels: 1
channel_spacing: 1.0
voxel_num : x256,y256
voxel_size : x0.0078125,y0.0078125
center : x0,y0
```

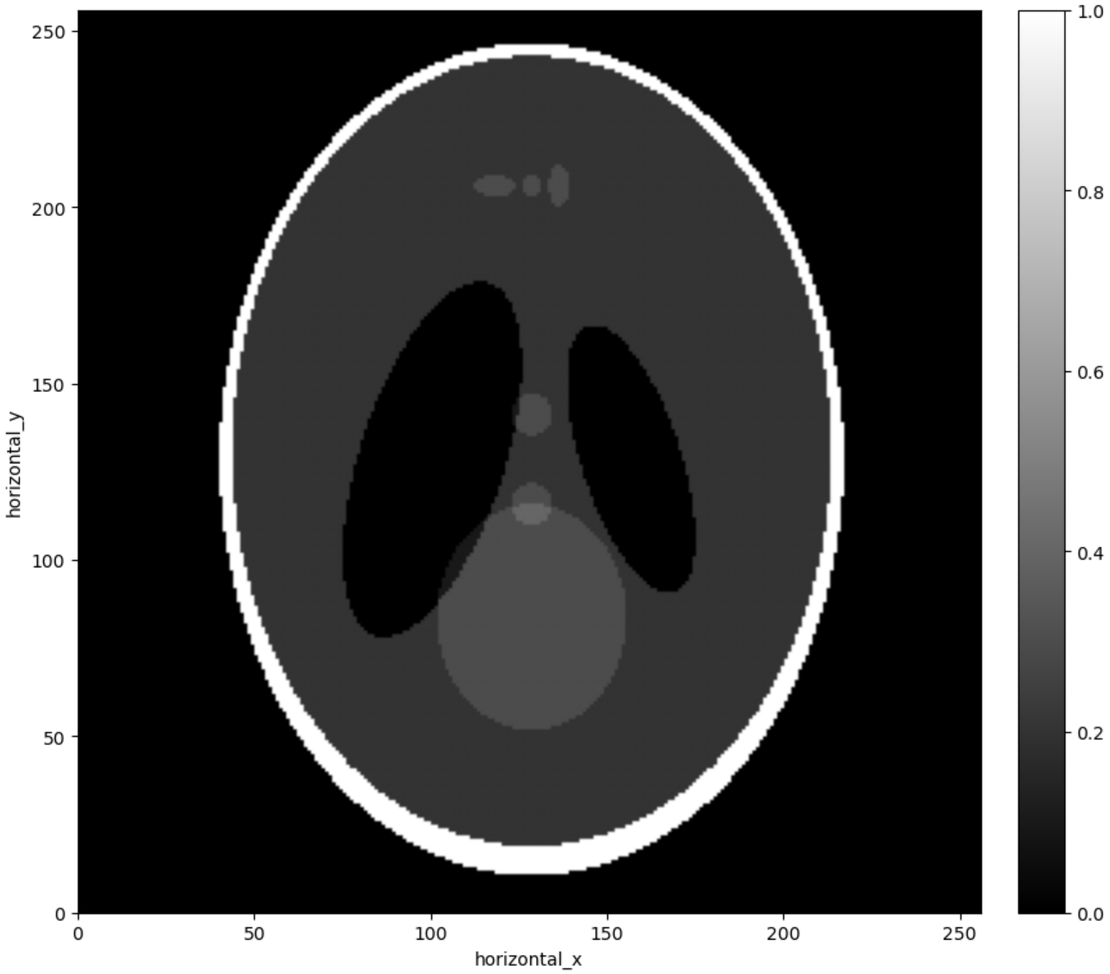
Using the CIL tomophantom plugin we can create a CIL `ImageData` holding the Shepp-Logan image of the desired size:

[5]:

```
phantom2D = cilTomoPhantom.get_ImageData(num_model=1, geometry=ig)
```

[6]:

```
show2D(phantom2D)
```



[6]:

```
<cil.utilities.display.show2D at 0x7fabd7248ce0>
```

Next, we specify the acquisition parameters and store them in an `AcquisitionGeometry` object. We use a parallel-beam geometry with 180 projections, and a detector with the same of number and size of pixels as the image:

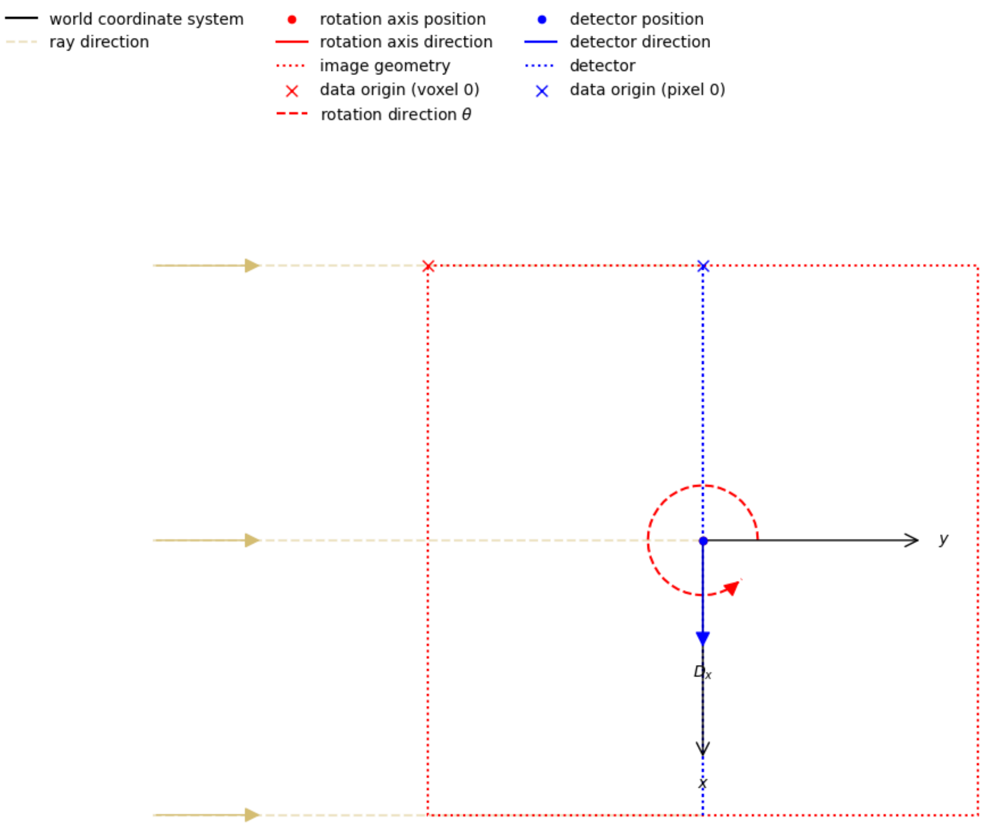
[7]:

```
num_angles = 180
ag = AcquisitionGeometry.create_Parallel2D()  \
                    .set_angles(np.linspace(0, 180, num_angles, endpoint=False))␣
  ↪ \
                    .set_panel(n, 2/n)
print(ag)
```

```
2D Parallel-beam tomography
System configuration:
        Ray direction: [0., 1.]
        Rotation axis position: [0., 0.]
        Detector position: [0., 0.]
        Detector direction x: [1., 0.]
Panel configuration:
        Number of pixels: [256   1]
        Pixel size: [0.0078125 0.0078125]
        Pixel origin: bottom-left
Channel configuration:
        Number of channels: 1
Acquisition description:
        Number of positions: 180
        Angles 0-9 in degrees: [0., 1., 2., 3., 4., 5., 6., 7., 8., 9.]
        Angles 170-179 in degrees: [170., 171., 172., 173., 174., 175., 176.,
177., 178., 179.]
        Full angular array can be accessed with acquisition_data.geometry.angles
Distances in units: units distance
```

We illustrate the geometry:

[8]:
```
show_geometry(ag)
```



```
[8]: <cil.utilities.display.show_geometry at 0x7faba70d1b50>
```

To simulate a sinogram we set up a ProjectionOperator using GPU-acceleration using the ASTRA plugin:

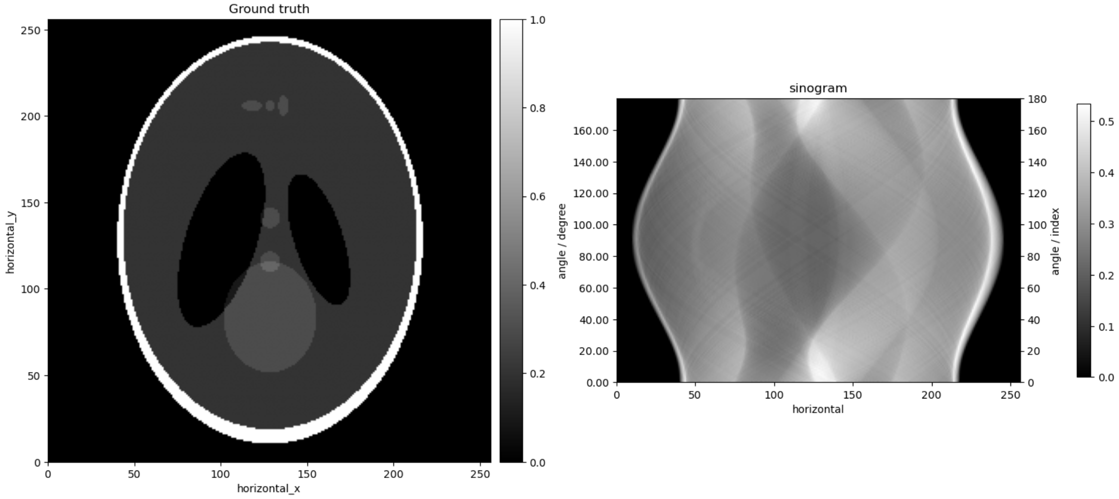
```
[9]: device = "gpu"
     A = ProjectionOperator(ig, ag, device)
```

The ideal noisefree sinogram is created by forward-projecting the phantom:

```
[10]: sinogram = A.direct(phantom2D)
```

The generated test image and sinogram are displayed as images:

[11]:
```python
plots = [phantom2D, sinogram]
titles = ["Ground truth", "sinogram"]
show2D(plots, titles)
```



[11]:
```
<cil.utilities.display.show2D at 0x7fab9d74f5c0>
```

**Add noise to the simulation** Next, Poisson noise will be applied to this noise-free sinogram. The severity of the noise can be adjusted by changing the background_counts variable.

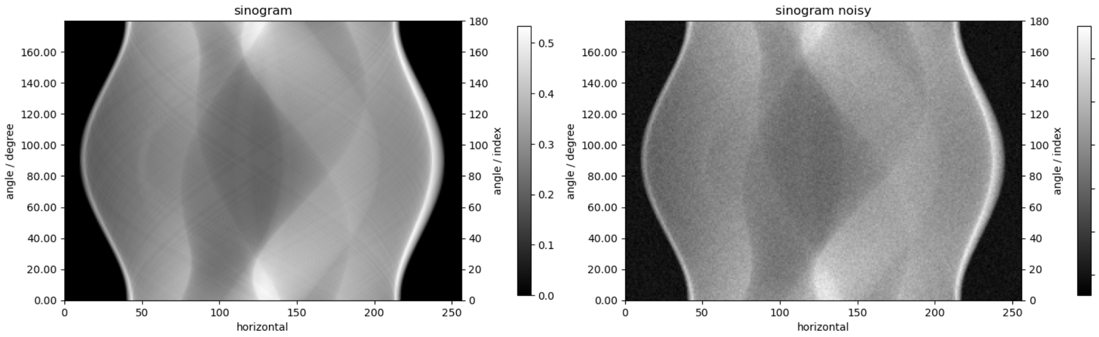
[12]:
```python
# Incident intensity: lower counts will increase the noise
background_counts = 5000

# Convert the simulated absorption sinogram to transmission values using Lambert-Beer.
# Use as mean for Poisson data generation.
# Convert back to absorption sinogram.
counts = background_counts * np.exp(-sinogram.as_array())
noisy_counts = np.random.poisson(counts)
sino_out = -np.log(noisy_counts/background_counts)

# Create new AcquisitionData object with same geometry and fill with noisy data.
sinogram_noisy = ag.allocate()
sinogram_noisy.fill(sino_out)
```

The simulated clean and noisy sinograms are displayed side by side as images:

[13]:
```python
plots = [sinogram, sinogram_noisy]
titles = ["sinogram", "sinogram noisy"]
show2D(plots, titles)
```



[13]:

```
<cil.utilities.display.show2D at 0x7fab9d6acaa0>
```

### 0.1.2 Reconstruct using CGLS

Before describing Tikhonov regularisation, we recall the problem solved by CGLS:

$$\underset{u}{\operatorname{argmin}} \left\|Au - b\right\|_2^2$$

where,

- $A$ is the projection operator
- $b$ is the acquired data
- $u$ is the unknown image to be determined

In the solution provided by CGLS the low frequency components tend to converge faster than the high frequency components. This means we need to control the number of iterations carefully to select the optimal solution.

Set up the CGLS algorithm, including specifying its initial point to start from:

[14]:

```
initial = ig.allocate(0)
cgls_simple = CGLS(initial=initial, operator=A, data=sinogram_noisy)
```

Once set up, we can run the algorithm for a specified number of iterations:

[15]:

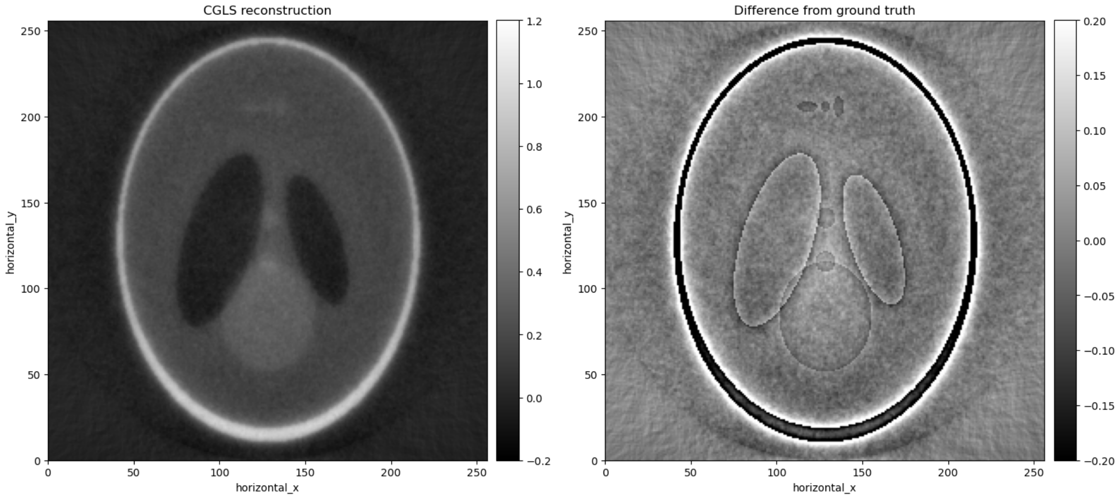
```
cgls_simple.run(5, verbose=True)
```

```
  0%|          | 0/5 [00:00<?, ?it/s]
```

Display the resulting image from CGLS, along with its difference image with the original ground truth image:

[16]:

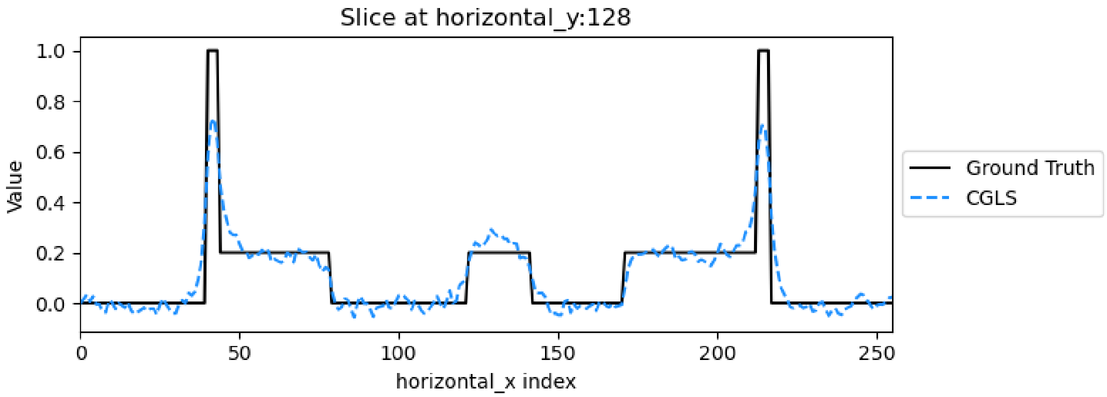
```
plots = [cgls_simple.solution, cgls_simple.solution - phantom2D]
titles = ["CGLS reconstruction","Difference from ground truth" ]
show2D(plots, titles, fix_range=[(-0.2,1.2),(-0.2,0.2)])
```



```
[16]: <cil.utilities.display.show2D at 0x7fab9c18eba0>
```

Plot central vertical line profile of CGLS and ground truth:

```
[17]: show1D([phantom2D, cgls_simple.solution],
              slice_list=[("horizontal_y",int(n/2))],
              dataset_labels=["Ground Truth","CGLS"],
              line_colours=['black','dodgerblue'],
              line_styles=['solid','dashed'])
```



```
[17]: <cil.utilities.display.show1D at 0x7fab9d81b140>
```

**Exercise 1:** Try running fewer and more iterations to see how the image and line profile changes. Try also with noisier data, by specifying a smaller value of background_counts. Remember you can change the number of iterations to run between outputs. Also note that the algorithm will continue from the point it stopped and run more iterations from that point if `run` is called again.

If you want to run from the beginning, the algorithm needs to be re-initialised. Try to stop the algorithm before the solution starts to diverge. go to section start

### 0.1.3 Tikhonov regularisation using CGLS

**Regularisation** Noisy datasets are problematic with an ill-posed problem such as tomographic reconstruction. If we try to solve these using CGLS we end up with an unstable solution. Regularisation adds information in order for us to solve the problem.

**Tikhonov regularisation** We can add a regularisation term to problem solved by CGLS; this gives us the minimisation problem in the following form, which is known as Tikhonov regularisation:

$$\underset{u}{\operatorname{argmin}} \left\|Au - b\right\|_2^2 + \alpha^2 \|Lu\|_2^2$$

where,

- $A$ is the projection operator
- $b$ is the acquired data
- $u$ is the unknown image to be solved for
- $\alpha$ is the regularisation parameter
- $L$ is a regularisation operator

The first term measures the fidelity of the solution to the data. The second term meausures the fidelity to the prior knowledge we have imposed on the system, operator $L$. $\alpha$ controls the trade-off between these terms. $L$ is often chosen to be a smoothing operator like the identity matrix, or a gradient operator **constrained to the squared L2-norm**.

This can be re-written equivalently in the block matrix form:

$$\underset{u}{\operatorname{argmin}} \left\|\binom{A}{\alpha L} u - \binom{b}{0}\right\|_2^2$$

With the definitions:

- $\tilde{A} = \binom{A}{\alpha L}$
- $\tilde{b} = \binom{b}{0}$

this can now be recognised as a least squares problem:

$$\underset{u}{\operatorname{argmin}} \left\|\tilde{A}u - \tilde{b}\right\|_2^2$$

and being a least squares problem, it can be solved using CGLS with $\tilde{A}$ as operator and $\tilde{b}$ as data.

**Introducing the block framework** We can construct $\tilde{A}$ and $\tilde{b}$ using the BlockFramework in the CIL.

$\tilde{A}$ is a (column) BlockOperator of size 2x1 and can be set up by

`BlockOperator(op0,op1)`

The right hand side $\tilde{b}$ is a BlockDataContainer and can be set up by

`BlockDataContainer(DataContainer0, DataContainer1)`

#### Reconstruct using CGLS and the identity operator

The simplest form of Tikhonov uses the identity matrix as the regularisation operator. We use an identity matrix as our regularisation operator we are penalising on the magnitude of the solution $u$, which will tend to reduce the pixel values of $u$.

[18]:
```
L = IdentityOperator(ig)
alpha = 0.1

operator_block = BlockOperator(A, alpha*L)
```

In the formulation of Tikhonov as a least squares problem, we need to set up the right hand side vector $\tilde{b}$ holding both the $b$ and a zero-filled `ImageData` of the right size, matching the range of the regularising operator. The operator allows us to query the geometry of its range and allocate a zero-filled `ImageData` of that geometry. We combine both into a `BlockDataContainer`:

[19]:
```
zero_data = L.range.allocate(0)

data_block = BlockDataContainer(sinogram_noisy, zero_data)
```

Run CGLS as before, but passing the BlockOperator and BlockDataContainer

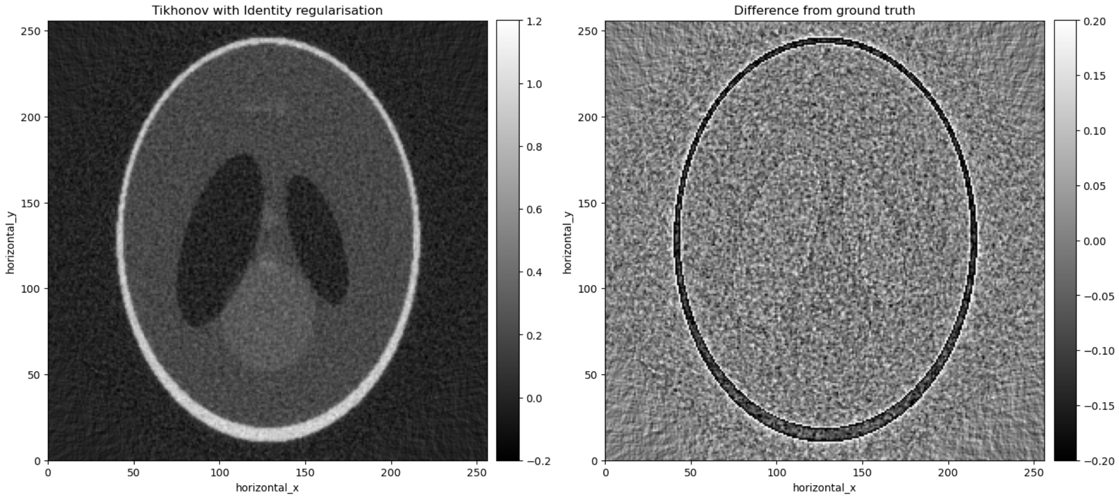
[20]:
```
#setup CGLS with the Block Operator and Block DataContainer
initial = ig.allocate(0)
cgls_tikh = CGLS(initial=initial, operator=operator_block, data=data_block,␣
 ↪update_objective_interval = 10)
```

[21]:
```
#run the algorithm
cgls_tikh.run(100)
```

```
  0%|          | 0/100 [00:00<?, ?it/s]
```

Display results as images and plot central vertical line profile of the Tikhonov with Identity:

[22]:
```
plots = [cgls_tikh.solution, cgls_tikh.solution - phantom2D]
titles = ["Tikhonov with Identity regularisation","Difference from ground␣
 ↪truth" ]
show2D(plots, titles, fix_range=[(-0.2,1.2),(-0.2,0.2)])
```



Tikhonov with Identity regularisation

Difference from ground truth

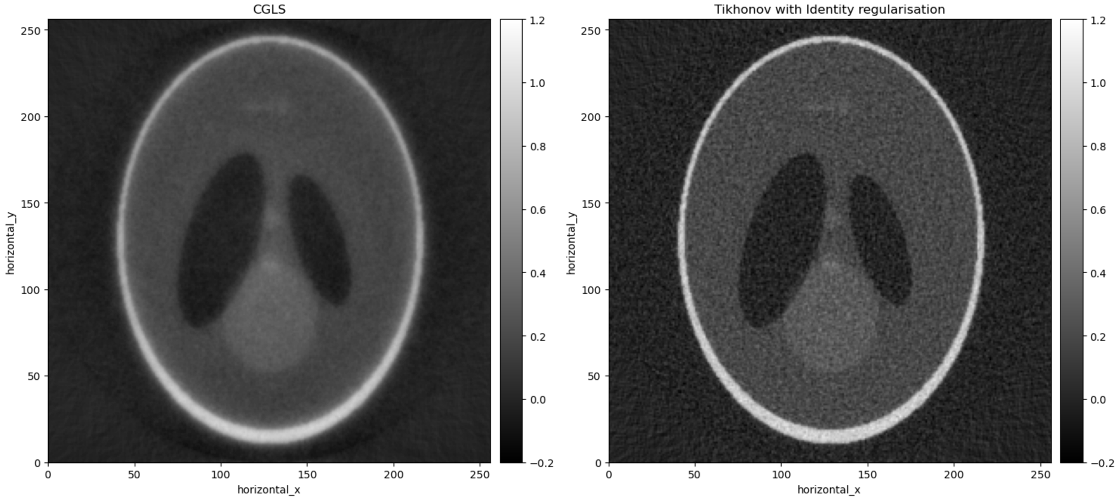
[22]:

```
<cil.utilities.display.show2D at 0x7fab9c3c24b0>
```

Let's compare the reconstructions from CGLS and Tikhonov with identity regularisation.

[23]:

```
plots = [cgls_simple.solution, cgls_tikh.solution]
titles = ["CGLS", "Tikhonov with Identity regularisation" ]
show2D(plots, titles, fix_range=(-0.2,1.2))
```



CGLS

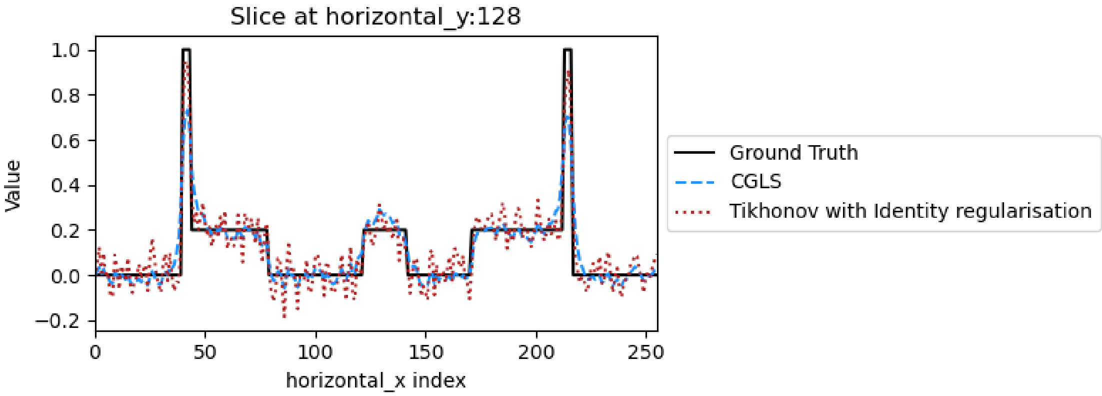
Tikhonov with Identity regularisation

[23]:

```
<cil.utilities.display.show2D at 0x7fabb61da090>
```

[24]:

```
show1D([phantom2D, cgls_simple.solution, cgls_tikh.solution],
       slice_list=[("horizontal_y", int(n/2))],
        dataset_labels=["Ground Truth","CGLS","Tikhonov with Identity␣
↪regularisation"],
        line_colours=['black','dodgerblue','firebrick'],
        line_styles=['solid','dashed','dotted'])
```



```
[24]: <cil.utilities.display.show1D at 0x7fabb60d73e0>
```

**Exercise 2:** Try running Tikhonov with a range of $\alpha$ values from very small to very large, display reconstruction and line profile and describe the effect of $\alpha$. Find the value of $\alpha$ that gives you the best solution. Then change how much noise you add to the data by going back here: set noise and run through the notebook again. Try with `background_counts` set to 5000, 10000 and 1000 remember to find an appropriate value of alpha for each run.

With Tikhonov regularisation the problem should now be solved to convergence instead of using the number of iterations as implicit regularising effect. By increasing the regularisation parameter $\alpha$ we balance the emphasis on fitting the data and enforcing the regularity. A low value of $\alpha$ will give you the CGLS solution, a higher value will reduce the noise in the reconstruction but at the cost of some blurring.

### 0.1.4 Using the BlockFramework to build a gradient operator

The basic Tikhonov with the identity operator provided perhaps a bit of improvement compared to just CGLS, but there was still a lot of noise in the reconstruction and the pixel values had been reduced. Using the identity as regularising operator means that we penalise pixel values that are non-zero, which may not be what we want. Instead, we want to encourage similar values of neighboring pixels to smooth out the noise. This can be achieved by using the gradient as the smoothing operator.

To do that we will again need to use the BlockFramework, which is now demonstrated in a bit more detail.

A discrete gradient operator (using finite differences) can be constructed using BlockOperators.

The direct gradient operator $\nabla$ acts on an image $u$ and returns a BlockDataContainer $\mathbf{w}$, holding finite differences in the $x$ and $y$ directions:

$$\nabla(u) = \begin{bmatrix} \nabla_x \\ \nabla_y \end{bmatrix} * u = \begin{bmatrix} \nabla_x u \\ \nabla_y u \end{bmatrix} = \begin{bmatrix} w_x \\ w_y \end{bmatrix} = \mathbf{w}$$

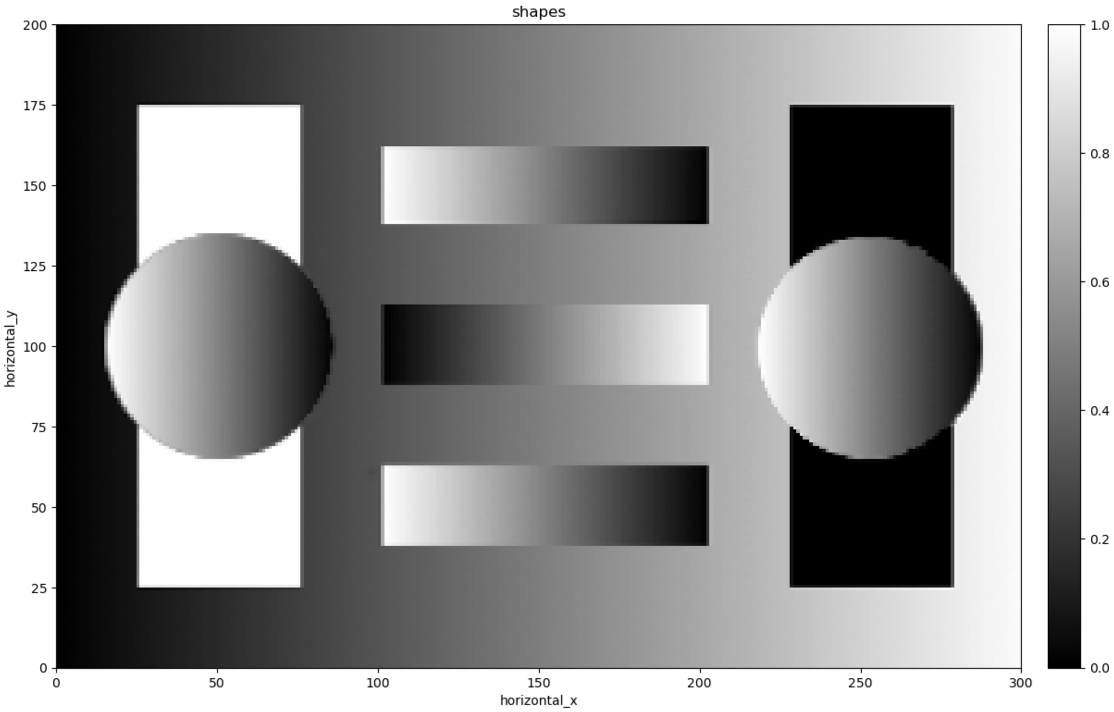
The adjoint gradient operator $\nabla^*$ acts on the BlockDataContainer $\mathbf{y}$ and returns an image $\rho$

$$\nabla^*(\mathbf{w}) = \begin{bmatrix} \nabla_x^* & \nabla_y^* \end{bmatrix} * \begin{bmatrix} w_x \\ w_y \end{bmatrix} = \begin{bmatrix} \nabla_x^* w_x + \nabla_y^* w_y \end{bmatrix} = \rho$$

We load a test image to demonstrate how the gradient operator works:

[25]:

```
shapes = SHAPES.get()
show2D(shapes, "shapes")
```



[25]:

```
<cil.utilities.display.show2D at 0x7fabb5fd5280>
```

The finite difference operator can be called from the framework. This returns the difference between each pair of pixels along one direction.

We need to initialise it with the image geometry, the direction of the calculation and the boundary conditions to use.

`FiniteDifferenceOperator(gm_domain, direction, bnd_cond='Neumann' or 'Periodic')`

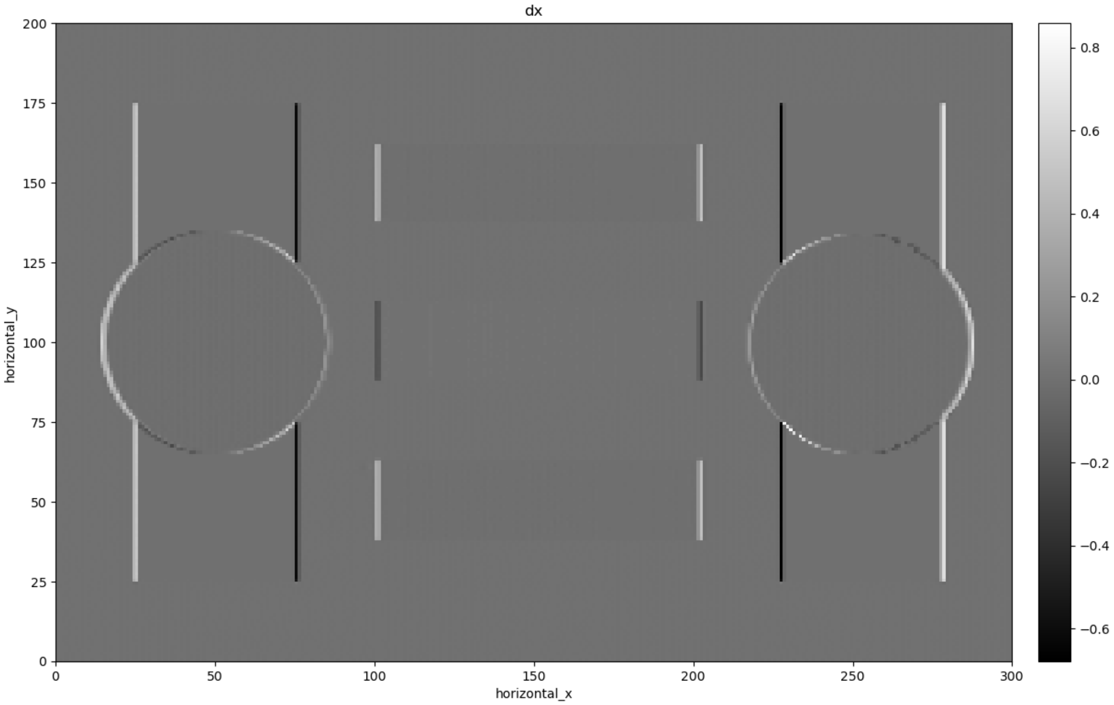
[26]:

```
#define the operator FiniteDiff - needs to image geometry, the direction and␣
 ↪the boundary conditions
```

```
fdx = FiniteDifferenceOperator(shapes.geometry, direction='horizontal_x',␣
↪bnd_cond='Neumann')

#run it over the input image
image_2D_dx = fdx.direct(shapes)

#plot ths results
show2D(image_2D_dx, "dx")
```



```
[26]: <cil.utilities.display.show2D at 0x7fabb60ef740>
```

Note how all vertical edges have been picked up (and their sign) applying this operator doing finite differences in the horizontal direction.

To set up a gradient in both $x$ and $y$ directions, we can create a BlockOperator to contain a finite difference operator for each of the $x$ and $y$ directions. We can apply it (using its `direct` method) to the test image and visualise the result.

```
[27]: # Define the x and y operators
fdx = FiniteDifferenceOperator(shapes.geometry, direction='horizontal_x',␣
↪bnd_cond='Neumann')
fdy = FiniteDifferenceOperator(shapes.geometry, direction='horizontal_y',␣
↪bnd_cond='Neumann')
```

[28]:

```
# Construct the BlockOperator combining the two operators
FD = BlockOperator(fdx, fdy)
```

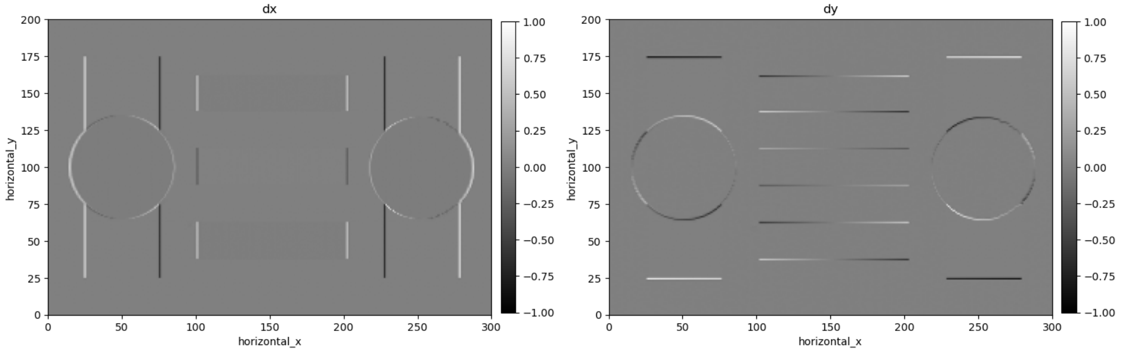
[29]:

```
#run it on the test image
fd_out = FD.direct(shapes)
```

Display output:

[30]:

```
plots = [fd_out.get_item(0), fd_out.get_item(1)]
titles = ["dx", "dy"]
show2D(plots, titles, fix_range=(-1,1))
```



[30]:

```
<cil.utilities.display.show2D at 0x7fabb5d34c80>
```

To see what is going on, we take a closer look at data types.

First, the input is an `ImageData` and its shape is a 2-element vector with the number of pixels in each direction:

[31]:

```
print(type(shapes))
print(shapes)
```

```
<class 'cil.framework.image_data.ImageData'>
Number of dimensions: 2
Shape: (200, 300)
Axis labels: (<ImageDimension.HORIZONTAL_Y: 'horizontal_y'>,
<ImageDimension.HORIZONTAL_X: 'horizontal_x'>)
```

The output however is a `BlockDataContainer`, essentially a list (with additional functionality) holding two `ImageData` elements, one for each direction we have taken finite differences. We can pick out each element of the `BlockDataContainer` and see that they indeed are `ImageData` and print their shapes (number of pixels in each direction):

[32]:

```
#output is BloackDataContainer
print(type(fd_out))
print(fd_out.shape)
```

```
<class 'cil.framework.block.BlockDataContainer'>
(2, 1)
```

[33]:
```
print("\tDataContainer 0")
print(type(fd_out.get_item(0)))
print(fd_out.get_item(0))
```

```
        DataContainer 0
<class 'cil.framework.image_data.ImageData'>
Number of dimensions: 2
Shape: (200, 300)
Axis labels: (<ImageDimension.HORIZONTAL_Y: 'horizontal_y'>,
<ImageDimension.HORIZONTAL_X: 'horizontal_x'>)
```

[34]:
```
print("\tDataContainer 1")
print(type(fd_out.get_item(1)))
print(fd_out.get_item(1))
```

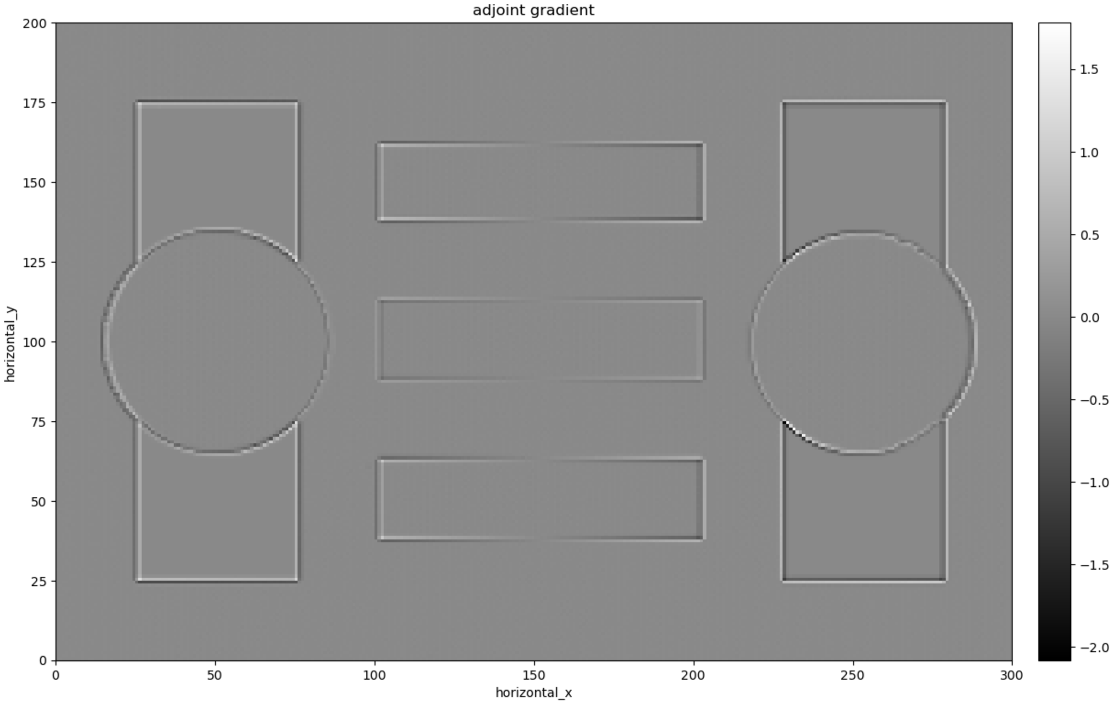
```
        DataContainer 1
<class 'cil.framework.image_data.ImageData'>
Number of dimensions: 2
Shape: (200, 300)
Axis labels: (<ImageDimension.HORIZONTAL_Y: 'horizontal_y'>,
<ImageDimension.HORIZONTAL_X: 'horizontal_x'>)
```

The BlockFramework provides basic algebra between BlockDataContainers, numpy arrays, lists of numbers, DataContainers, subclasses and scalars providing the shape of the containers are compatible - add - subtract - multiply - divide - power - squared_norm

The `BlockOperator` is a special kind of `Operator`, and being an `Operator` it should have an adjoint method. This is automatically provided from the adjoints of the operators. In the present case our `BlockOperator` will take a `BlockDataContainer` as input to its adjoint and return an `ImageData`, as visualised below:

[35]:
```
# Run the adjoint method
adjoint_output = FD.adjoint(fd_out)

show2D(adjoint_output, "adjoint gradient")
```



```
[35]: <cil.utilities.display.show2D at 0x7fabb5a6e960>
```

### 0.1.5 A deeper look at the BlockFramework

**BlockDataContainer** BlockDataContainer holds datacontainers as a column vector

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

**BlockOperator:** BlockOperator is a matrix of operators.

$$K = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \\ A_5 & A_6 \end{bmatrix}_{(3,2)} * \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{(2,1)}}_{\mathbf{x}} = \begin{bmatrix} A_1 x_1 + A_2 x_2 \\ A_3 x_1 + A_4 x_2 \\ A_5 x_1 + A_6 x_2 \end{bmatrix}_{(3,1)} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}_{(3,1)} = \mathbf{y}$$

Column: Share the same domains $X_1, X_2$ Rows: Share the same ranges $Y_1, Y_2, Y_3$

$$K : (X_1 \times X_2) \rightarrow (Y_1 \times Y_2 \times Y_3)$$

$$A_1, A_3, A_5 : \text{share the same domain } X_1$$
$$A_2, A_4, A_6 : \text{share the same domain } X_2$$

$$\begin{aligned} &A_1 : X_1 \to Y_1, \quad &A_3 : X_1 \to Y_2, \quad &A_5 : X_1 \to Y_3 \\ &A_2 : X_2 \to Y_1, \quad &A_4 : X_2 \to Y_2, \quad &A_6 : X_2 \to Y_3 \end{aligned}$$

### 0.1.6 Reconstruct using Tikhonov by CGLS with the gradient operator

**Tikhonov regularisation** Now we go back to our Tikhonov reconstruction, this time use the gradient operator in the regulariser.

$$\operatorname{argmin} \left\| \left(\begin{smallmatrix} A \\ \alpha\nabla \end{smallmatrix}\right) u - \left(\begin{smallmatrix} b \\ 0 \end{smallmatrix}\right) \right\|_2^2$$

With the definitions:

- $\tilde{A} = \left(\begin{smallmatrix} A \\ \alpha\nabla \end{smallmatrix}\right)$
- $\tilde{b} = \left(\begin{smallmatrix} b \\ 0 \end{smallmatrix}\right)$

And solve using CGLS:

$$\underset{u}{\operatorname{argmin}} \left\| \tilde{A} u - \tilde{b} \right\|_2^2$$

We'll use the framework's `Gradient()` operator - this is an optimised form of FD over the space dimensions (or even or space+channels in case of multiple channels).

**Exercise 3:** Set up the BlockOperator $\tilde{A}$ and the BlockDataContainer $\tilde{b}$ as before but with the Gradient operator. Outline code to be completed is given in the next two code cells. Once set up, run the following cells to execute CGLS with these as input. Run Tikhonov reconstruction using gradient regularisation. Try a range of $\alpha$ values ranging from very small to very large, visualise the resulting image and central line profiles, and describe the effect of the regularisation parameter choice. Find the $\alpha$ that (visually) gives you the best solution.

[ ]:

```

```

**Uncomment the following line and run the cell to see the solution, to run the lines you'll need to run the cell a second time**

[36]:

```
operator_block = BlockOperator( A, alpha * L, shape=(2,1))
data_block = BlockDataContainer(sinogram_noisy, L.range_geometry().allocate(0))
```

[37]:

```
#setup CGLS with the block operator and block data
initial = ig.allocate(0)
cgls_tikh_g = CGLS(initial=initial, operator=operator_block, data=data_block, update_objective_interval = 10)
```

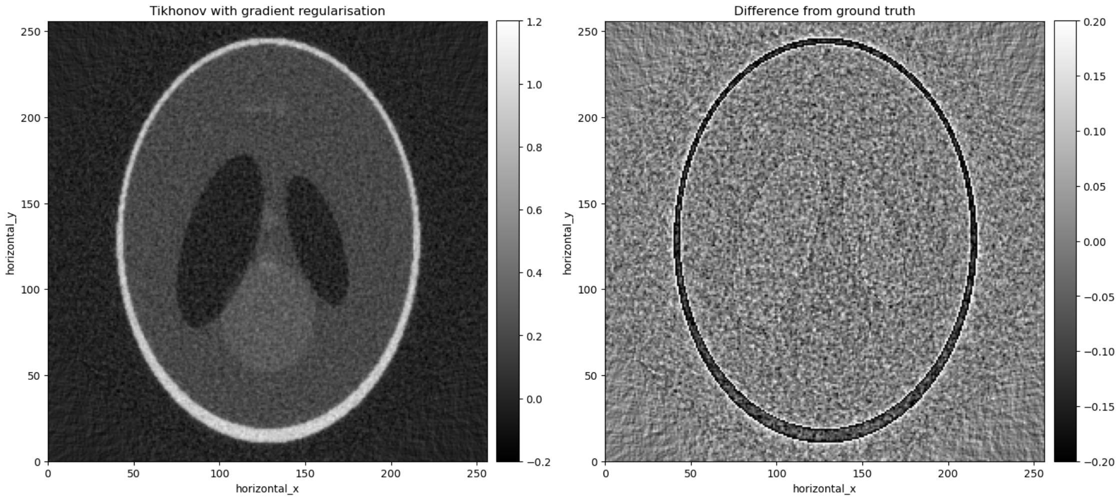
[38]:

```
#run the algorithm
cgls_tikh_g.run(200, verbose=True)
```

```
0%|          | 0/200 [00:00<?, ?it/s]
```

[39]:

```
#plot the results
plots = [cgls_tikh_g.solution, cgls_tikh_g.solution - phantom2D]
titles = ["Tikhonov with gradient regularisation", "Difference from ground␣
 ↪truth" ]
show2D(plots, titles, fix_range=[(-0.2,1.2),(-0.2,0.2)])
```



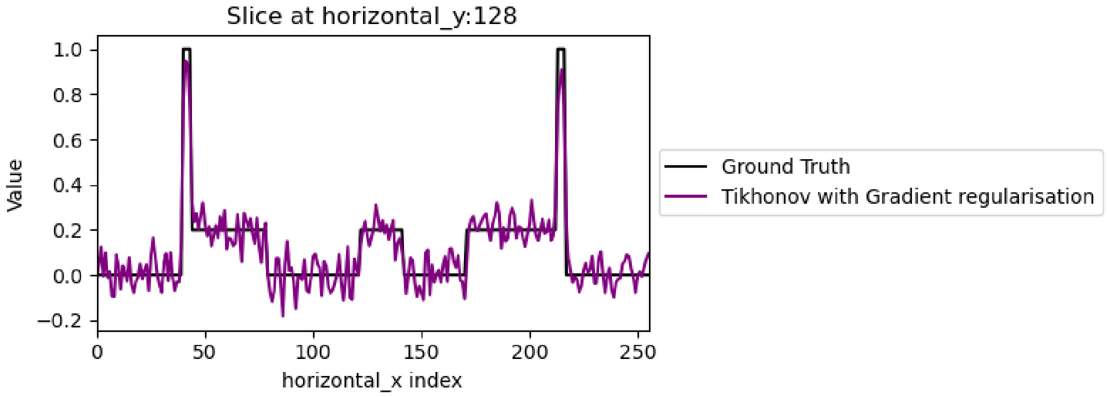
[39]:

```
<cil.utilities.display.show2D at 0x7fabb5fced80>
```

Central vertical line profiles of ground truth and Tikhonov with Gradient operator:

[40]:

```
show1D([phantom2D, cgls_tikh_g.solution],
       slice_list=[("horizontal_y",int(n/2))],
       dataset_labels=["Ground Truth","Tikhonov with Gradient regularisation"],
       line_colours=['black','purple'],
       line_styles=['solid','solid'])
```

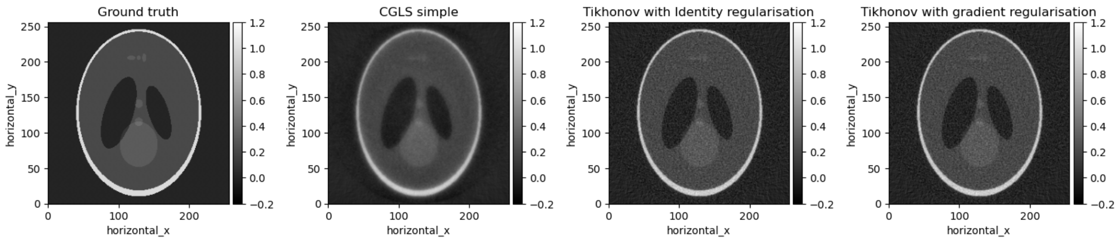


```
[40]: <cil.utilities.display.show1D at 0x7fabb5d0a600>
```

### 0.1.7 Summary

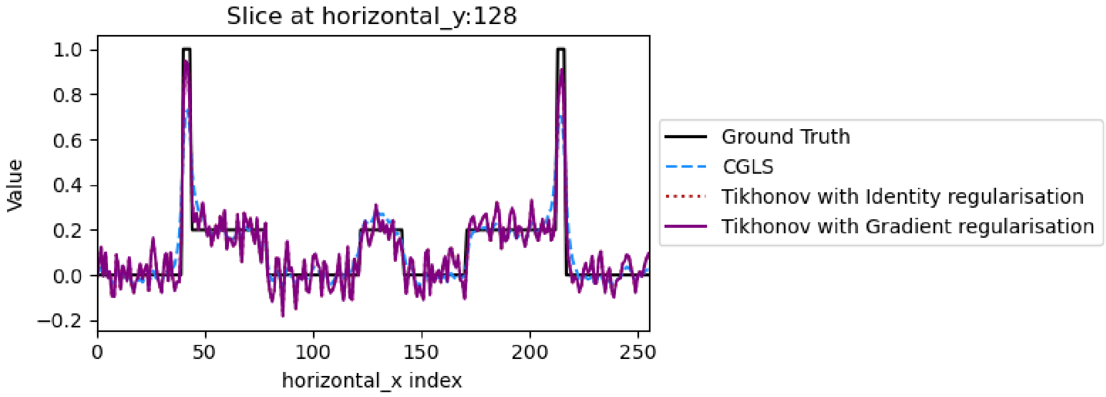
**Comparison of the outputs of each reconstruction** To wrap up we compare the reconstructions produced by all reconstruction methods considered in this notebook: Simple CGLS, Tikhonov with Identity regularisation and Tikhonov with Gradient regularisation, along with the ground truth image. We display images and central vertical line profiles:

```
[41]: plots = [phantom2D, cgls_simple.solution, cgls_tikh.solution, cgls_tikh_g.
        ↪solution]
      titles = ["Ground truth", "CGLS simple", "Tikhonov with Identity␣
        ↪regularisation", "Tikhonov with gradient regularisation" ]
      show2D(plots, titles, fix_range=(-0.2,1.2), num_cols=4)
```



```
[41]: <cil.utilities.display.show2D at 0x7fab9c1e5b80>
```

```
[42]: show1D([phantom2D, cgls_simple.solution, cgls_tikh.solution, cgls_tikh_g.
        ↪solution],
             slice_list=[("horizontal_y",int(n/2))],
             dataset_labels=["Ground Truth","CGLS","Tikhonov with Identity␣
        ↪regularisation", "Tikhonov with Gradient regularisation"],
             line_colours=['black','dodgerblue','firebrick','purple'],
             line_styles=['solid','dashed','dotted','solid'])
```



```
[42]: <cil.utilities.display.show1D at 0x7fabb5030440>
```

As can be seen from images and line profiles, Tikhonov with Gradient regularisation allows us to reduce the noise in the reconstruction substantially. However, we may pay a price in terms of blurring the edges.

**Learning objectives:**

After having worked through this notebook, we have now seen how to:

1. Construct and manipulate BlockOperators and BlockDataContainer, including direct and adjoint operations and algebra.
2. Use Block Framework to solve Tikhonov regularisation with CGLS algorithm.
3. Apply Tikhonov regularisation to tomographic reconstruction and explain the effect of regularisation parameter and operator in regulariser.

This completes the present exercise on Tikhonov regularisation with the CGLS algorithm. In other exercises we will see how to use the CIL framework for real data reconstruction and how to do regularisation based on non-smooth optimisation, to help preserve edges better, while reducing the noise.