7-gVirtualXRay_vs_real_radiograph_PMMA_block

March 2, 2022

[1]:
```
%matplotlib inline

from IPython.display import display
from IPython.display import Image
from utils import * # Code shared across more than one notebook
```

Main contributors: F. P. Vidal and Jenna Allsup

Purpose: In this notebook, we aim to demonstrate that gVirtualXRay is able to generate analytic simulations on GPU comparable to X-ray radiographs taken with a clinically utilised equipment. Such devices often produce images in negative and image filtering, such as image sharpening, may alter their appearance. Another issue is the lack of flat-field correction to account for variations in the pixel-to-pixel sensitivity of the detector.

Material and Methods: We simulate an image with gVirtualXRay and compare it with a ground truth image. For this purpose, we use a real radiograph of a plexiglass (PMMA) block. Its size is 32x27x4cm. The simulated geometry is registered using numerical optimisation so that its location and orientation match those of the real PMMA block. The source-to-object distance (SOD) and the source-to-detector distance (SDD) are also optimised. The initial estimation of the SDD is 130cm. The beam spectrum is polychromatic and the energy response of the detector is considered. The pixel-to-pixel sensitivity of the detector is corrected in the real radiograph. A plausible image sharpening filter is also integrated and optimised to mimic the appearance of a real image. Photon noise is also added to increase realism. The amount of noise is estimated for the real radiograph and optimised in the simulated image.

Results: The mean absolute percentage error (MAPE), also known as mean absolute percentage deviation (MAPD), between the real radiograph and the simulated image without noise and with noise. The **zero-mean normalised cross-correlation and** Structural Similarity Index (SSIM)** are also used.

The **ZNCC** for the image **without noise** is **98.56%**, and **with noise 98.53%**. Both values are extremely close to 100%. The **SSIM** for the image **without noise** is **0.93**, and **with noise 0.90**. Both values are extremely close to 1.0. These results show that the simulated image is similar to the real X-ray radiograph acquired with a medical device. MAPE is, however, **10.69%** for the image **without noise**, and **10.75% with noise**. The difference between the real X-ray radiograph and the simulated image is due to the sharpening filter (we do not actually know how the manufacturer implemented it) and the pixel-to-pixel sensitivity of the detector (even after correction the horizontal profile in the middle of the real X-ray radiograph is not flat).

The calculations were performed on the following platform:

[2]:
```
printSystemInfo()
```

```
OS:
        Linux 5.3.18-150300.59.49-default
        x86_64

CPU:
        AMD Ryzen 7 3800XT 8-Core Processor

RAM:
        63 GB
GPU:
        Name: GeForce RTX 2080 Ti
        Drivers: 455.45.01
        Video memory: 11 GB
```

1 Import packages

- Pretty common stuff: re, sys, os, math, copy, numpy
- cma for optimisation (minimisation of an objective function)
- gvxrPython3 to simulate X-ray images
- imageio to generate a GIF file
- matplotlib to plot graphs and display images
- scikit-image for comparing images and for implementing the image sharpening filter
- scikit-learn for computing the image comparison
- scipy for resampling the beam spectrum
- SimpleITK to load the medical DICOM file
- spekpy to generate a beam spectrum
- tifffile to load and save TIFF images in Float32

[3]:
```
import re, sys, os, math, copy

import numpy as np # Who does not use Numpy?

import cma # Optimisation (minimisation of an objective function)
import gvxrPython3 as gvxr # Simulate X-ray images
import imageio # Generate a GIF file
import matplotlib # Plot graphs and display images
# old_backend =  matplotlib.get_backend()
# matplotlib.use("Agg")  # Prevent showing stuff

from matplotlib.cm import get_cmap
import matplotlib.pyplot as plt # Plotting
from matplotlib.colors import LogNorm # Look up table
from matplotlib.colors import PowerNorm # Look up table
import matplotlib.colors as mcolors
```

```
font = {'family' : 'serif',
        #'weight' : 'bold',
         'size'   : 12.5
       }
matplotlib.rc('font', **font)
# matplotlib.rc('text', usetex=True)

from skimage.util import compare_images # Comparing images
from skimage.filters import gaussian # Implementing the image sharpening filter
from skimage.metrics import structural_similarity as ssim # Quantifying the␣
 ↪similarities between the simulated image and the ground truth
from sklearn.metrics import mean_squared_error # For the objective function
from sklearn.metrics import mean_absolute_percentage_error as mape #␣
 ↪Quantifying the difference between the simulated image and the ground truth
from scipy import signal # Resampling the beam spectrum
import SimpleITK as sitk # Load the medical DICOM file
import spekpy as sp # Generate a beam spectrum
from tifffile import imwrite, imread # Load and save TIFF images in Float32

import datetime # For the runtime

from utils import *
```

2 Preparation of the ground truth image

2.1 Read the real X-ray radiograph from a DICOM file

[4]:
```
reader = sitk.ImageFileReader()
reader.SetImageIO("GDCMImageIO")
reader.SetFileName("PMMA_data/DX000000")
reader.LoadPrivateTagsOn()
reader.ReadImageInformation()
volume = reader.Execute()
raw_real_image = sitk.GetArrayFromImage(volume)[0]
```

[5]:
```
# meta_data_keys = volume.GetMetaDataKeys()

# print("DICOM fields:")
# for key in meta_data_keys:
#     print(key, volume.GetMetaData(key))
```

[6]:
```
# def cleanTags(raw_string):
#     regular_expression = re.compile('<.*?>')
#     clean_text = re.sub(regular_expression, '', raw_string)
```

```
#     return clean_text
```

[7]:
```
# field = volume.GetMetaData("0033|1022")

# for item in field.split("\n"):
#     if "KV" in item:
#         kv = float(cleanTags(item))


# print(kv)
```

2.2 Extract the image size and pixel spacing from the DICOM file

It will be useful to set the X-ray detector parameters for the simulation, and to display the images in millimetres.

[8]:
```
spacing = volume.GetSpacing()[0:2]
size = volume.GetSize()[0:2]
```

2.3 Extract the kVp from the DICOM file

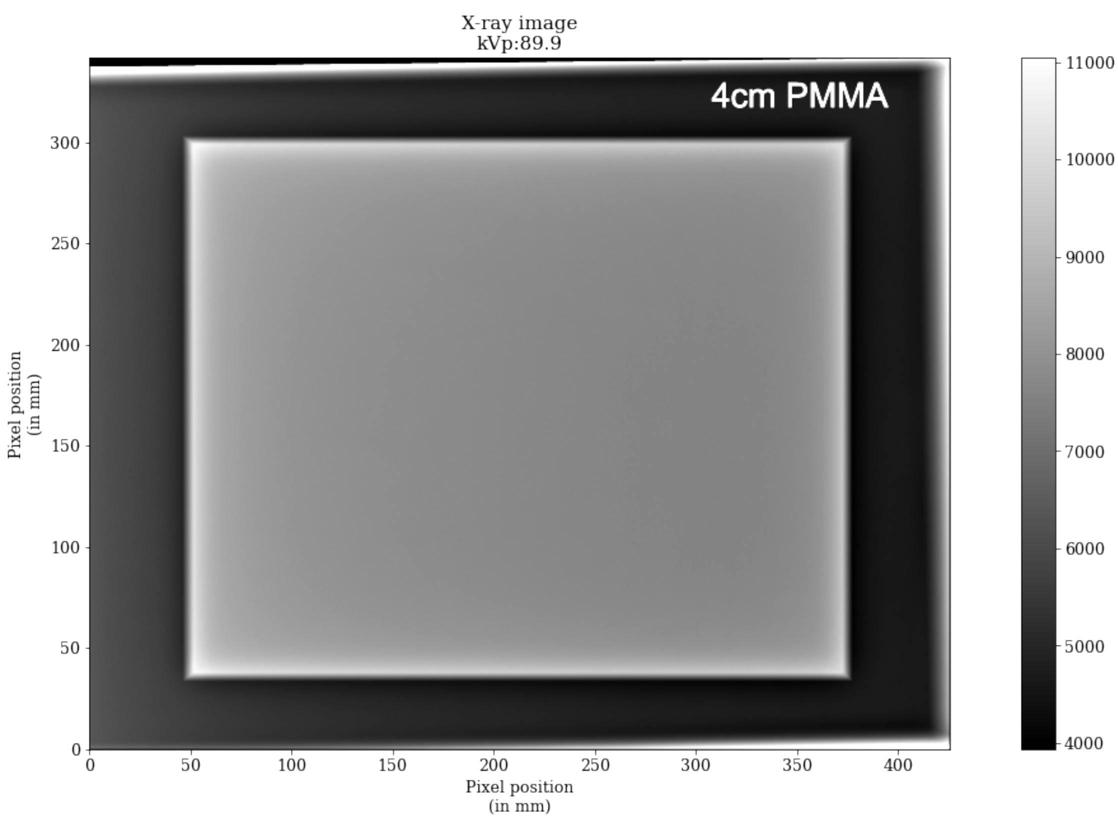
It will be useful to generate a realistic beam spectrum.

[9]:
```
kVp = float(volume.GetMetaData("0018|0060"))
print("Peak kilo voltage output of the x-ray generator used: ", kVp)
```

```
Peak kilo voltage output of the x-ray generator used:  89.9
```

2.4 Display the experimental radiograph from the DICOM file

[10]:
```
plt.figure(figsize= (20,10))
xrange=range(raw_real_image.shape[1])
yrange=range(raw_real_image.shape[0])

plt.xlabel("Pixel position\n(in mm)")
plt.ylabel("Pixel position\n(in mm)")
plt.title("X-ray image\nkVp:" + str(kVp))
plt.imshow(raw_real_image, cmap="gray",
           vmin=3934, vmax = 11045,
           extent=[0,(raw_real_image.shape[1]-1)*spacing[0],0,(raw_real_image.
 ↪shape[0]-1)*spacing[1]])
plt.colorbar(orientation='vertical')

plt.savefig('plots/PMMA_experimental_image.pdf')
plt.savefig('plots/PMMA_experimental_image.png')
```



We can see on the picture that the pixel-to-pixel sensitivity of the detector varries. Pixels on the left-hand side of the image are brighter than on the right-hand side.

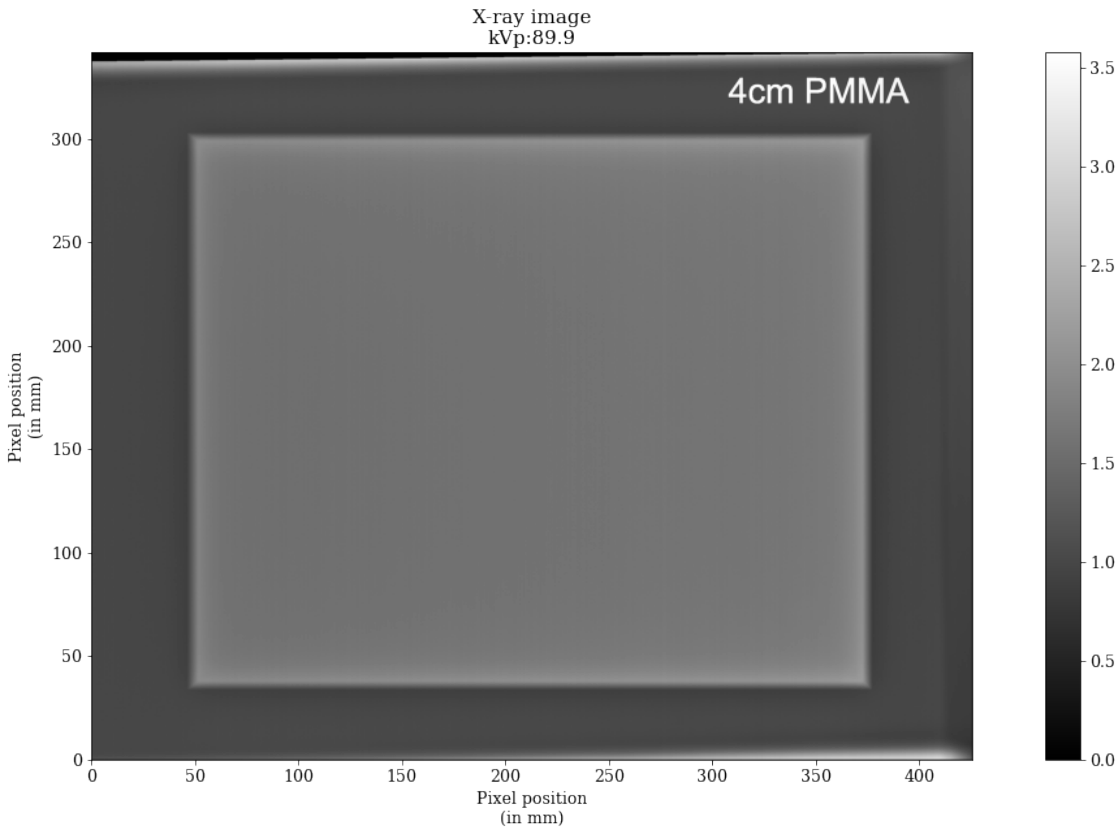
2.5 Flat-field correction

We are going to correct the pixel-to-pixel sensitivity of the detector and implement a mock flat-field correction. We extract two lines from the image where there is no object. The two lines are stored in two 1D arrays. We create a new array as the element-wise average of the two arrays. Every line of the image is then corrected with an element-wise division.

[11]:
```
line1 = raw_real_image[222,:] # A line of the image with no object
line2 = raw_real_image[2283,:] # Another line of the image with no object
line = 0.5 * line1 + 0.5 * line2 # Element-wise average of the two arrays
line[line.shape[0]-1] = 1 # Make sure the last element is not zero (to avoid a␣
 ↪division by zero)

# Create the full-field image
full_field = np.zeros(raw_real_image.shape)

for i in range(raw_real_image.shape[1]):
    full_field[:,i] = line[i]

# Apply the correction
real_image = raw_real_image / full_field

# Save the corrected image
imwrite("gVirtualXRay_output_data/PMMA_block_real_image_flat.tif", real_image.
 ↪astype(np.single))
```

Display the image after correction

[12]:
```
plt.figure(figsize= (20,10))
xrange=range(real_image.shape[1])
yrange=range(real_image.shape[0])

plt.xlabel("Pixel position\n(in mm)")
plt.ylabel("Pixel position\n(in mm)")
plt.title("X-ray image\nkVp:" + str(kVp))
plt.imshow(real_image, cmap="gray",
           # vmin=-1, vmax = 1,
           extent=[0,(real_image.shape[1]-1)*spacing[0],0,(real_image.
 ↪shape[0]-1)*spacing[1]])
plt.colorbar(orientation='vertical')

plt.savefig('plots/PMMA_experimental_image_flat.pdf')
plt.savefig('plots/PMMA_experimental_image_flat.png')
```

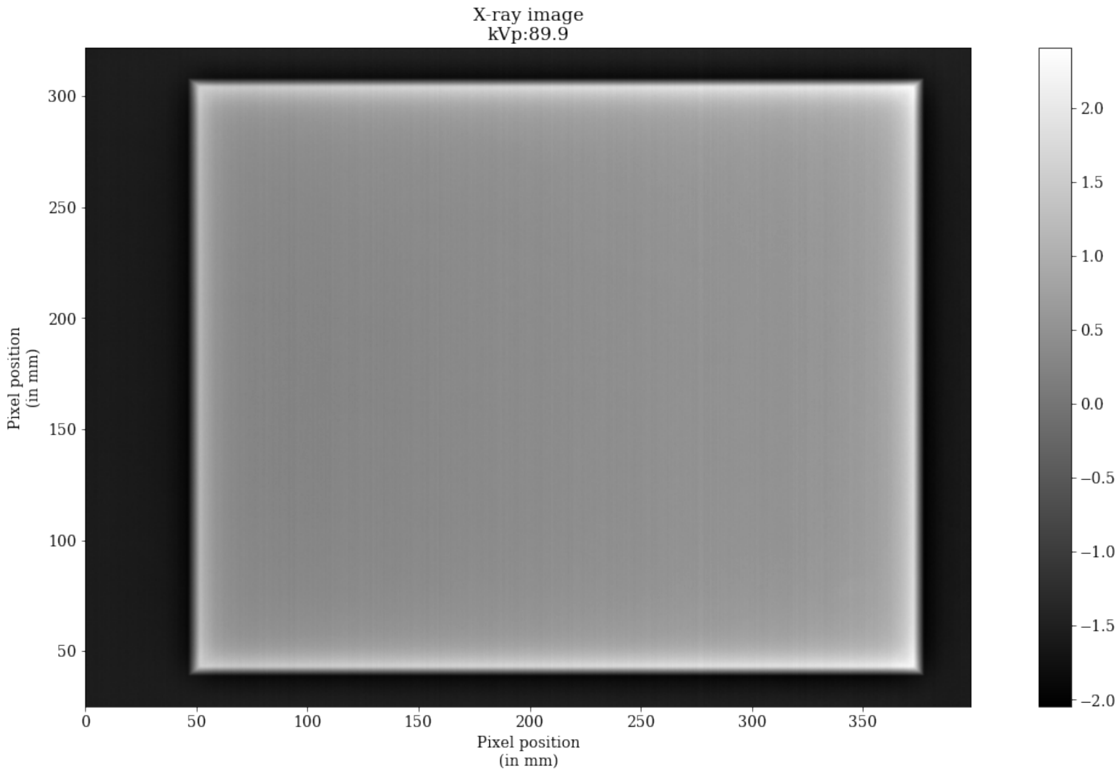


2.6 Crop it to remove the text

There are parts of the image that need to be removed, e.g. the text.

[13]:
```
plt.figure(figsize= (20,10))

roi = [179, 2300, 0, 2848]
roi_real_image = standardisation(real_image[179:2300,0:2848])
roi_raw_real_image = standardisation(real_image[179:2300,0:2848])

xrange=range(roi_real_image.shape[1])
yrange=range(roi_real_image.shape[0])

plt.xlabel("Pixel position\n(in mm)")
plt.ylabel("Pixel position\n(in mm)")
plt.title("X-ray image\nkVp:" + str(kVp))
plt.imshow(roi_real_image, cmap="gray",
           extent=[roi[2] * spacing[0], (roi[3] -1) *spacing[0],
 ↪roi[0]*spacing[0],(roi[1] - 1)*spacing[0]])
plt.colorbar(orientation='vertical')
```

```
plt.savefig('plots/PMMA_experimental_image-cropped.pdf')
plt.savefig('plots/PMMA_experimental_image-cropped.png')
```



3 Initialise gVirtualXRay

3.1 Set the experimental parameters (e.g. source and detector positions, etc.)

We use known parameters as much as possible, for example we know the size and composition of the sample. Some parameters are extracted from the DICOM file, such as detector size, pixel resolution, and voltage of the X-ray tube.

[14]:
```
source_detector_distance_in_cm = 130  # See email Mon 05/07/2021 15:29
block_width_in_cm = 32
block_height_in_cm = 27
block_thickness_in_cm = 4

window_size =  [800, 450]
source_position = [0.0, 0.0, source_detector_distance_in_cm -␣
 ↪block_thickness_in_cm / 2, "cm"]
detector_position = [0.0, 0.0, - block_thickness_in_cm / 2, "cm"]
```

```
detector_up = [0, 1, 0]
```

3.2 Initialise the simulation engine

[15]:
```
# Create an OpenGL context
print("Create an OpenGL context:",
    str(window_size[0]) + "x" + str(window_size[1])
)

gvxr.createWindow(-1, True, "EGL")

gvxr.setWindowSize(
    window_size[0],
    window_size[1]
)
```

```
Create an OpenGL context: 800x450
```

```
0
1.5
4.5.0 NVIDIA 455.45.01


Wed Mar  2 17:15:58 2022 ---- Create window gvxrStatus: Create window
Wed Mar  2 17:15:59 2022 ---- EGL version: Wed Mar  2 17:15:59 2022 ---- OpenGL
version supported by this platform OpenGL renderer:   GeForce RTX 2080
Ti/PCIe/SSE2
OpenGL version:    4.5.0 NVIDIA 455.45.01
OpenGL vender:     NVIDIA Corporation
Wed Mar  2 17:15:59 2022 ---- Use OpenGL 4.5.0 0 500 500
0 0 800 450
```

3.3 Function to create a PMMA block

[16]:
```
def createBlock(x, y, z, r, h):
    # Remove all the geometries from the whole scenegraph
    gvxr.removePolygonMeshesFromSceneGraph()

    # Make a cube
    gvxr.makeCube("PMMA block", 1.0, "mm")

    # Translation vector
    gvxr.translateNode("PMMA block", x, y, z, "cm")

    # Rotation angle
    gvxr.rotateNode("PMMA block", r, 0, 0, 1)
```

```
    # Scaling factors
    gvxr.scaleNode("PMMA block", block_width_in_cm * 10, h * 10,␣
 ↪block_thickness_in_cm * 10);

    # Apply the transformation matrix so that we can save the corresponding STL␣
 ↪file
    gvxr.applyCurrentLocalTransformation("PMMA block");

    # Set the matrix's material properties
    gvxr.setCompound("PMMA block", "C5O2H8")
    gvxr.setDensity("PMMA block", 1.18, "g/cm3")

    # Add the matrix to the X-ray renderer
    gvxr.addPolygonMeshAsInnerSurface("PMMA block")
```

[17]:
```
# Initial guess
x = y = 0
z = 0
r = 0
createBlock(x, y, z, r, block_height_in_cm)
```

3.4 Set the source position

[18]:
```
# Set up the beam
print("Set up the beam")
print("\tSource position:", source_position)
gvxr.setSourcePosition(
    source_position[0],
    source_position[1],
    source_position[2],
    source_position[3]
);

gvxr.usePointSource();

focal_spot_size = volume.GetMetaData("0018|1413")
# gvxr.setFocalSpot(focal_spot_size, focal_spot_size, focal_spot_size, "mm");
print("size of focal spot (in mm):", focal_spot_size)
```

```
Set up the beam
        Source position: [0.0, 0.0, 128.0, 'cm']
size of focal spot (in mm): 11.653
```

3.5 Get the spectrum from the DICOM file

[19]:
```
spectrum = {};
filter_material = "Al"      # See email Mon 05/07/2021 15:29
filter_thickness_in_mm = 3  # See email Mon 05/07/2021 15:29

s = sp.Spek(kvp=kVp)
s.filter(filter_material, filter_thickness_in_mm) # Filter by 3 mm of Al
unit = "keV"
k, f = s.get_spectrum(edges=True) # Get the spectrum

min_energy = sys.float_info.max
max_energy = -sys.float_info.max

for energy, count in zip(k, f):
    count = round(count)

    if count > 0:

        max_energy = max(max_energy, energy)
        min_energy = min(min_energy, energy)

        if energy in spectrum.keys():
            spectrum[energy] += count
        else:
            spectrum[energy] = count
```

Reformat the data

[20]:
```
# get the integral nb of photons
nbphotons=0.
energy1 = -1.
energy2 = -1.

for energy in spectrum.keys():

    if energy1<0:
        energy1 = float(energy)
    elif energy2<0:
        energy2 = float(energy)
    nbphotons += float(spectrum[energy])
sampling = (energy2-energy1)

# get spectrum
data = []
for energy in spectrum.keys():
    source = [float(energy),float(spectrum[energy])/(nbphotons*sampling)]
```

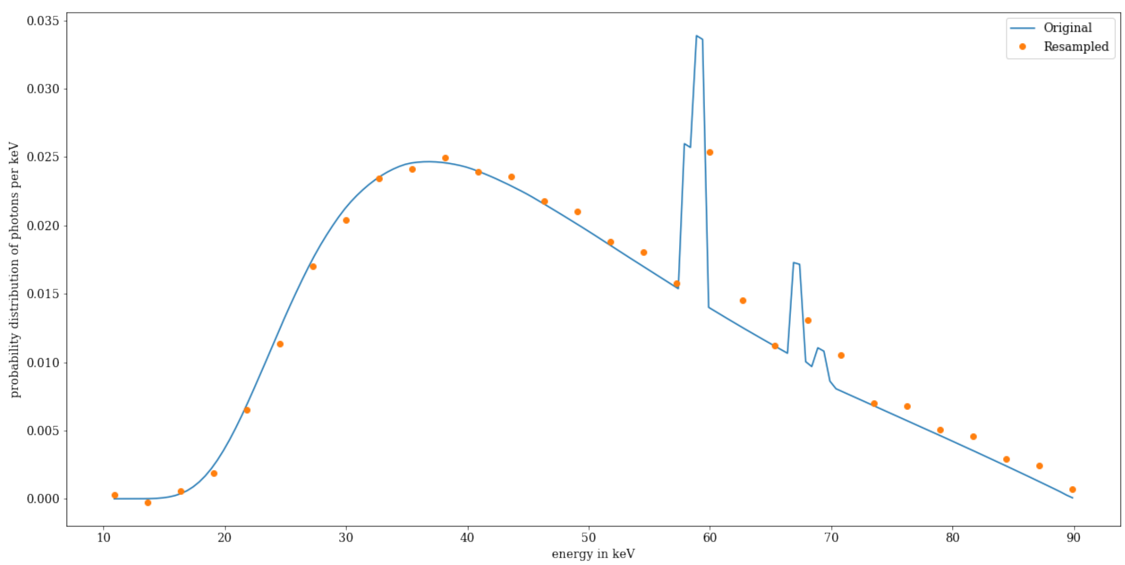
```
    data.append(source)

data_array = np.array(data)

energies, counts = data_array.T
```

Resample the data to reduce the number of bins

[21]:
```
number_of_energy_bins = 30
count_set = signal.resample(counts, number_of_energy_bins)
energy_set = np.linspace(energies.min(), energies.max(), number_of_energy_bins,␣
↪endpoint=True)
```

Plot the beam spectrum from spekpy and the resampled vervion

[22]:
```
plt.figure(figsize= (20,10))
plt.plot(energies, counts, label="Original")
plt.plot(energy_set, count_set, "o", label="Resampled")
plt.xlabel('energy in keV')
plt.ylabel('probability distribution of photons per keV')
plt.legend()
plt.savefig("plots/PMMA_block_spectrum.pdf")
```



3.6 Load the beam spectrum in the simulator

[23]:
```
gvxr.resetBeamSpectrum() # To be on the safe side when debugging
for energy, count in zip(energy_set, count_set):
    gvxr.addEnergyBinToSpectrum(energy, unit, count);
```

3.7 Set the X-ray detector

[24]:
```
# Set up the detector
print("Set up the detector");
print("\tDetector position:", detector_position)
gvxr.setDetectorPosition(
    detector_position[0],
    detector_position[1],
    detector_position[2],
    detector_position[3]
);

print("\tDetector up vector:", detector_up)
gvxr.setDetectorUpVector(
    detector_up[0],
    detector_up[1],
    detector_up[2]
);
```

```
Set up the detector
        Detector position: [0.0, 0.0, -2.0, 'cm']
        Detector up vector: [0, 1, 0]
```

[25]:
```
print("\tDetector number of pixels:", size)
gvxr.setDetectorNumberOfPixels(
    size[0],
    size[1]
);

print("\tPixel spacing:", spacing)
gvxr.setDetectorPixelSize(
    spacing[0],
    spacing[1],
    "mm"
);
```

```
        Detector number of pixels: (3040, 2442)
        Pixel spacing: (0.14, 0.14)
```

Load the detector response in energy

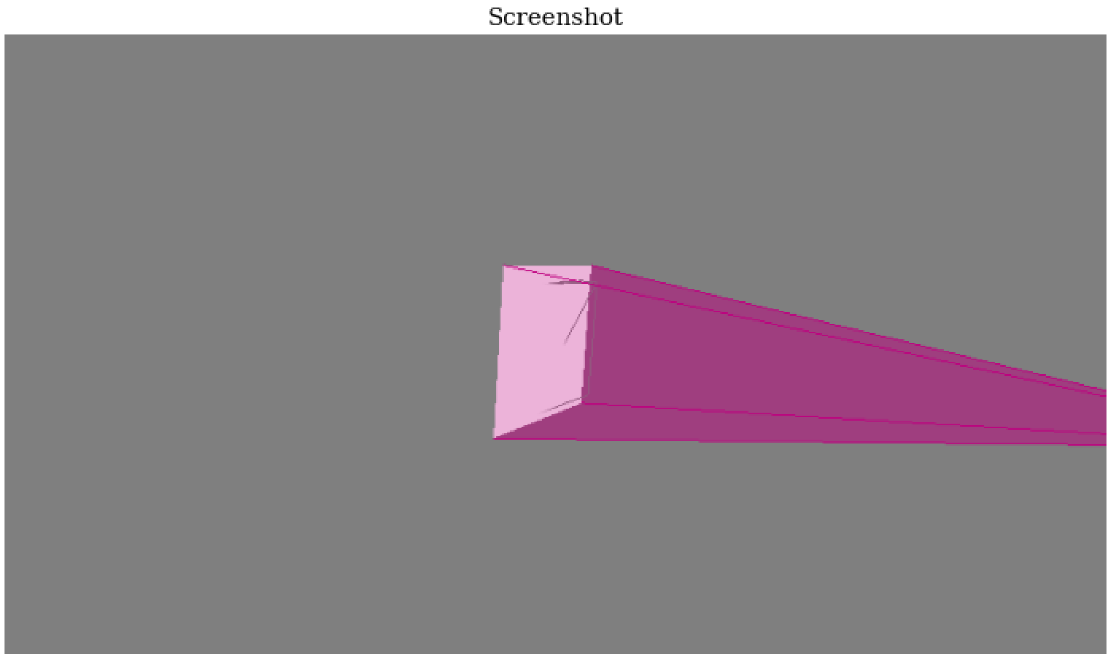
[26]:
```
gvxr.clearDetectorEnergyResponse() # To be on the safe side
gvxr.loadDetectorEnergyResponse("Gate_data/responseDetector.txt",
                                "MeV")
```

3.8 Take a screenshot of the 3D environment

[27]:
```
gvxr.displayScene()

gvxr.useNegative()
gvxr.useLighing()
gvxr.useWireframe()
gvxr.setSceneRotationMatrix([0.43813619017601013, 0.09238918125629425, -0.
 ↪8941444158554077, 0.0,
                             0.06627026945352554, 0.9886708855628967, 0.
 ↪13463231921195984, 0.0,
                             0.8964602947235107, -0.11824299395084381, 0.
 ↪4270564019680023, 0.0,
                             0.0, 0.0, 0.0, 1.0])
gvxr.setZoom(1639.6787109375)

gvxr.displayScene()
```

[28]:
```
screenshot = gvxr.takeScreenshot()
```

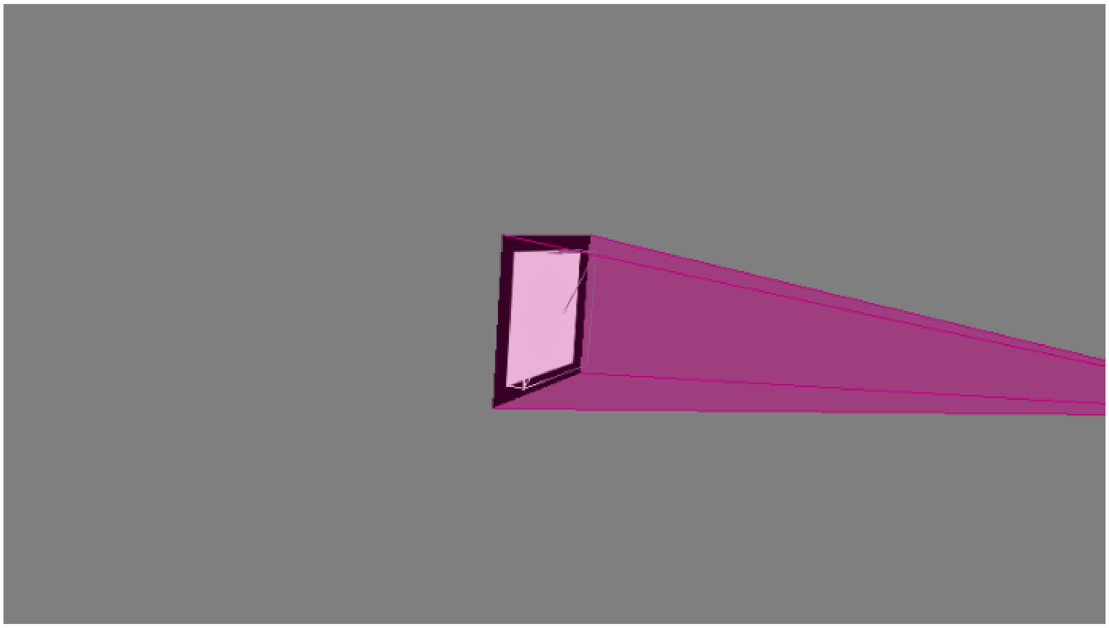
[29]:
```
plt.figure(figsize= (10,10))
plt.title("Screenshot")
plt.imshow(screenshot)
plt.axis('off')

plt.tight_layout()

plt.savefig('plots/PMMA_screenshot-beam-off.pdf')
plt.savefig('plots/PMMA_screenshot-beam-off.png')
```



[30]:
```
gvxr.computeXRayImage()
gvxr.displayScene()
```

[31]:
```
screenshot = gvxr.takeScreenshot()

plt.figure(figsize= (10,10))
plt.title("Screenshot")
plt.imshow(screenshot)
plt.axis('off')

plt.tight_layout()

plt.savefig('plots/PMMA_screenshot-beam-on.pdf')
plt.savefig('plots/PMMA_screenshot-beam-on.png')
```

Screenshot



4 Optimise the position and orientation of the PMMA block, and refine the SDD

4.1 Create an objective function

[32]:
```
def zncc(i1, i2):
    return (np.mean(np.multiply(i1, i2))) / 2.0;
```

[33]:
```
def objectiveFunction(parameters):

    global best_fitness
    global best_fitness_id
    global fitness_function_call_id
    global evolution_zncc
    global evolution_parameters

    # Retrieve the parameters
    x, y, z, r, h, SDD = parameters

    # Update the source position
    source_position = [0.0, 0.0, SDD - block_thickness_in_cm / 2, "cm"]
    gvxr.setSourcePosition(
        source_position[0],
```

```
        source_position[1],
        source_position[2],
        source_position[3]
    );

    # Update the block geometry
    createBlock(x, y, z, r, h)

    # Compute an X-ray image
    x_ray_image = np.array(gvxr.computeXRayImage())

    # Compute the negative image as it is the case for the real image
    x_ray_image *= -1.

    # Crop the image
    x_ray_image = x_ray_image[179:2300,0:2848]

    # Zero-mean, unit-variance normalistion
    x_ray_image = standardisation(x_ray_image)

    # Return the objective
    objective = math.sqrt(mean_squared_error(roi_real_image, x_ray_image))

    # The block below is not necessary for the registration.
    # It is used to save the data to create animations.
    if best_fitness > objective:

        gvxr.displayScene()
        screenshot = (255 * np.array(gvxr.takeScreenshot())).astype(np.uint8)

        # gvxr.saveSTLfile("PMMA block", "gVirtualXRay_output_data/PMMA_block_" + str(best_fitness_id) + ".stl")
        # np.savetxt("gVirtualXRay_output_data/PMMA_block_" + str(best_fitness_id) + ".dat", [x, y, z, r, h, SDD], header='x,y,z,r,h,SDD')
        imwrite("gVirtualXRay_output_data/PMMA_block_xray_" + str(best_fitness_id) + ".tif", x_ray_image.astype(np.single))
        imwrite("gVirtualXRay_output_data/PMMA_block_screenshot_" + str(best_fitness_id) + ".tif", screenshot)

        zncc_value = zncc(roi_real_image, x_ray_image)
        evolution_zncc.append([fitness_function_call_id, zncc_value])
        evolution_parameters.append([fitness_function_call_id, x, y, z, r, h, SDD])

        best_fitness = objective
        best_fitness_id += 1
```

```
    fitness_function_call_id += 1

    return objective
```

[34]:
```
# Zero-mean, unit-variance normalistion
roi_real_image = standardisation(roi_real_image)
imwrite("gVirtualXRay_output_data/PMMA_block_roi_real_image.tif",␣
 ↪roi_real_image)
```

[35]:
```
# The registration has already been performed. Load the results.
if os.path.isfile("gVirtualXRay_output_data/PMMA_block.dat") and \
        os.path.isfile("gVirtualXRay_output_data/PMMA_block_evolution_zncc.
 ↪dat") and \
        os.path.isfile("gVirtualXRay_output_data/
 ↪PMMA_block_evolution_parameters.dat"):

    temp = np.loadtxt("gVirtualXRay_output_data/PMMA_block.dat")
    x = temp[0]
    y = temp[1]
    z = temp[2]
    r = temp[3]
    block_height_in_cm = temp[4]
    source_detector_distance_in_cm = temp[5]

    # Update the source position
    source_position = [0.0, 0.0, source_detector_distance_in_cm -␣
 ↪block_thickness_in_cm / 2, "cm"]
    gvxr.setSourcePosition(
        source_position[0],
        source_position[1],
        source_position[2],
        source_position[3]
    );

    # Update the block geometry
    createBlock(x, y, z, r, block_height_in_cm)

    evolution_zncc = np.loadtxt("gVirtualXRay_output_data/
 ↪PMMA_block_evolution_zncc.dat")
    evolution_parameters = np.loadtxt("gVirtualXRay_output_data/
 ↪PMMA_block_evolution_parameters.dat")

else:
    # Optimise
    opts = cma.CMAOptions()
    opts.set('tolfun', 1e-2)
```

```
    opts['tolx'] = 1e-2
    opts['bounds'] = [[-20, -20, detector_position[2] + block_thickness_in_cm /
↪2, -90, block_height_in_cm * 0.80, source_detector_distance_in_cm * 0.80],
                      [20, 20, source_position[2] - block_thickness_in_cm / 2,
↪90, block_height_in_cm * 1.20, source_detector_distance_in_cm * 1.20]]
    opts['CMA_stds'] = []

    for min_val, max_val in zip(opts['bounds'][0], opts['bounds'][1]):
        opts['CMA_stds'].append(abs(max_val - min_val) * 0.05)

    best_fitness = sys.float_info.max;
    best_fitness_id = 0;
    fitness_function_call_id = 0
    evolution_zncc = []
    evolution_parameters = []

    # Optimise
    res = cma.fmin(objectiveFunction,
              [x, y, z, r, block_height_in_cm, source_detector_distance_in_cm],
              0.5,
              opts)
    x, y, z, r, block_height_in_cm, source_detector_distance_in_cm = res[0]

    # Save the parameters
    np.savetxt("gVirtualXRay_output_data/PMMA_block.dat", [x, y, z, r,
↪block_height_in_cm, source_detector_distance_in_cm], header='x,y,z,r,h,SDD')

    # Update the source position
    source_position = [0.0, 0.0, source_detector_distance_in_cm -
↪block_thickness_in_cm / 2, "cm"]
    gvxr.setSourcePosition(
        source_position[0],
        source_position[1],
        source_position[2],
        source_position[3]
    );

    # Update the block geometry
    createBlock(x, y, z, r, block_height_in_cm)
    gvxr.saveSTLfile("PMMA block", "CAD_models/PMMA_block.stl")

    evolution_zncc = np.array(evolution_zncc)
    np.savetxt("gVirtualXRay_output_data/PMMA_block_evolution_zncc.dat",
↪evolution_zncc, header='t,ZNCC')

    evolution_parameters = np.array(evolution_parameters)
```

```
    np.savetxt("gVirtualXRay_output_data/PMMA_block_evolution_parameters.dat",␣
  ↪evolution_parameters, header='t,x,y,z,r,h,SDD')
```

```
(4_w,9)-aCMA-ES (mu_w=2.8,w_1=49%) in dimension 6 (seed=344565, Wed Mar  2
17:16:03 2022)
Iterat #Fevals   function value  axis ratio  sigma  min&max std  t[m:s]
    1      9 6.032214636078368e-01 1.0e+00 4.51e-01  2e-01  4e+00 0:06.6
    2     18 5.760208449184751e-01 1.2e+00 3.84e-01  2e-01  3e+00 0:12.4
    3     27 6.014607331039005e-01 1.5e+00 3.03e-01  2e-01  3e+00 0:18.0
    4     36 3.983802193250981e-01 1.5e+00 3.56e-01  2e-01  3e+00 0:23.6
    5     45 4.217816002205489e-01 1.9e+00 4.14e-01  3e-01  4e+00 0:29.1
    6     54 4.157273453351011e-01 2.3e+00 4.14e-01  3e-01  3e+00 0:34.4
    8     72 4.237091495155809e-01 2.5e+00 3.32e-01  3e-01  3e+00 0:45.3
   10     90 5.379802510065053e-01 2.7e+00 3.05e-01  2e-01  2e+00 0:56.1
   12    108 4.338366369466534e-01 2.9e+00 2.29e-01  2e-01  2e+00 1:07.0
   14    126 4.190002796597788e-01 2.8e+00 2.17e-01  2e-01  1e+00 1:17.8
   16    144 3.704088013410742e-01 3.0e+00 1.88e-01  1e-01  1e+00 1:29.1
   18    162 3.571304842815214e-01 3.1e+00 1.65e-01  1e-01  9e-01 1:40.3
   21    189 3.233101764641921e-01 3.6e+00 1.66e-01  1e-01  8e-01 1:57.1
   24    216 3.221808929024901e-01 3.6e+00 1.51e-01  1e-01  7e-01 2:13.6
   27    243 2.900561703347026e-01 3.3e+00 1.37e-01  9e-02  6e-01 2:30.5
   30    270 2.889870209996885e-01 4.2e+00 1.18e-01  7e-02  6e-01 2:47.0
   33    297 2.855642743947838e-01 4.8e+00 9.36e-02  5e-02  4e-01 3:03.2
   37    333 2.851854596789206e-01 5.7e+00 6.87e-02  4e-02  3e-01 3:25.1
   41    369 2.812753904325871e-01 5.8e+00 4.96e-02  3e-02  2e-01 3:47.3
   45    405 2.798749636436667e-01 6.7e+00 3.96e-02  2e-02  1e-01 4:10.0
   49    441 2.798959269781965e-01 8.6e+00 3.10e-02  1e-02  1e-01 4:32.0
   53    477 2.796007917034827e-01 1.0e+01 2.00e-02  6e-03  7e-02 4:54.4
   57    513 2.795085105154771e-01 1.4e+01 1.34e-02  4e-03  5e-02 5:16.8
   62    558 2.794581927745673e-01 2.0e+01 1.42e-02  3e-03  8e-02 5:44.5
   64    576 2.794690968313948e-01 2.8e+01 1.51e-02  3e-03  1e-01 5:55.7
termination on tolfun=0.01 (Wed Mar  2 17:21:59 2022)
final/bestever f-value = 2.794387e-01 2.794141e-01
incumbent solution: [-0.06260318692473704, 0.24413465550524388,
0.32135169141381265, -0.021316809587439, 26.07991798039349, 130.4062773731475]
std deviation: [0.0037714830375661333, 0.0029774609575447196,
0.03211960924567781, 0.015391321234395273, 0.007657045026605248,
0.09657648002308826]
```

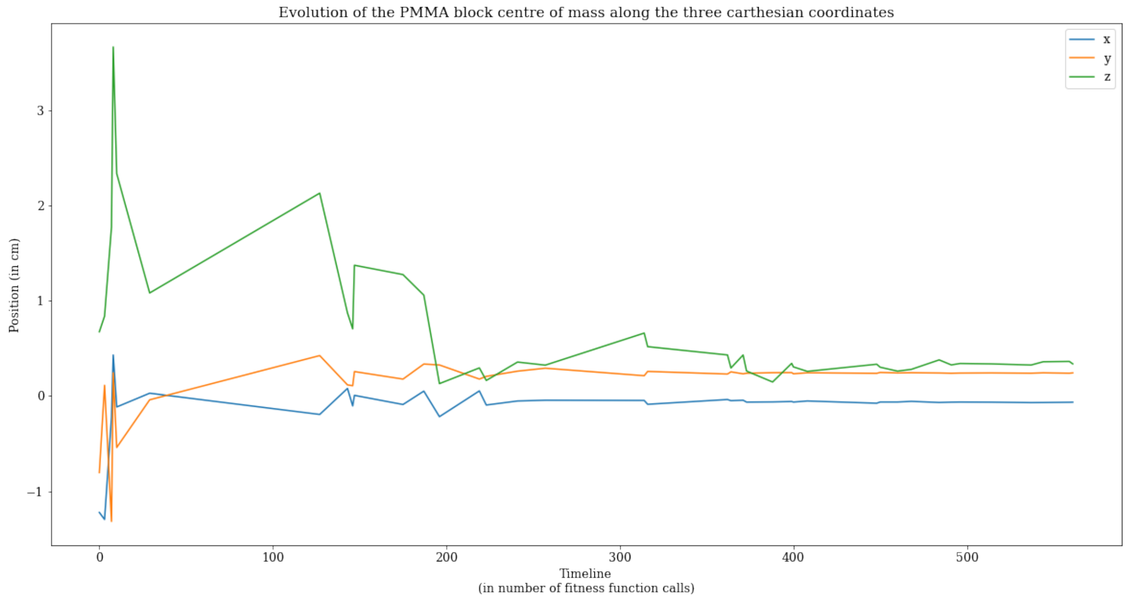
[36]:
```
print("SDD (in cm):", source_detector_distance_in_cm)
print("Block centre (in cm):", x, y, z)
print("Block size (in cm):", block_width_in_cm, block_height_in_cm,␣
  ↪block_thickness_in_cm)
print("Rotation angle (in degrees):", r)
```

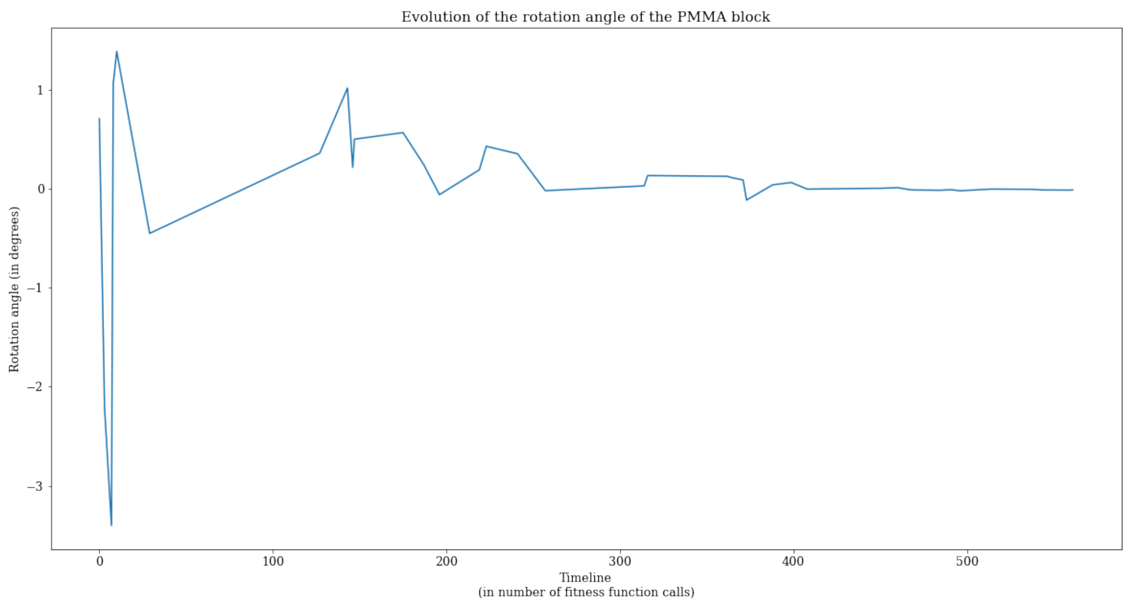
```
SDD (in cm): 130.47880021275753
Block centre (in cm): -0.0641264926281038 0.2433088037596416 0.3378670521654954
```

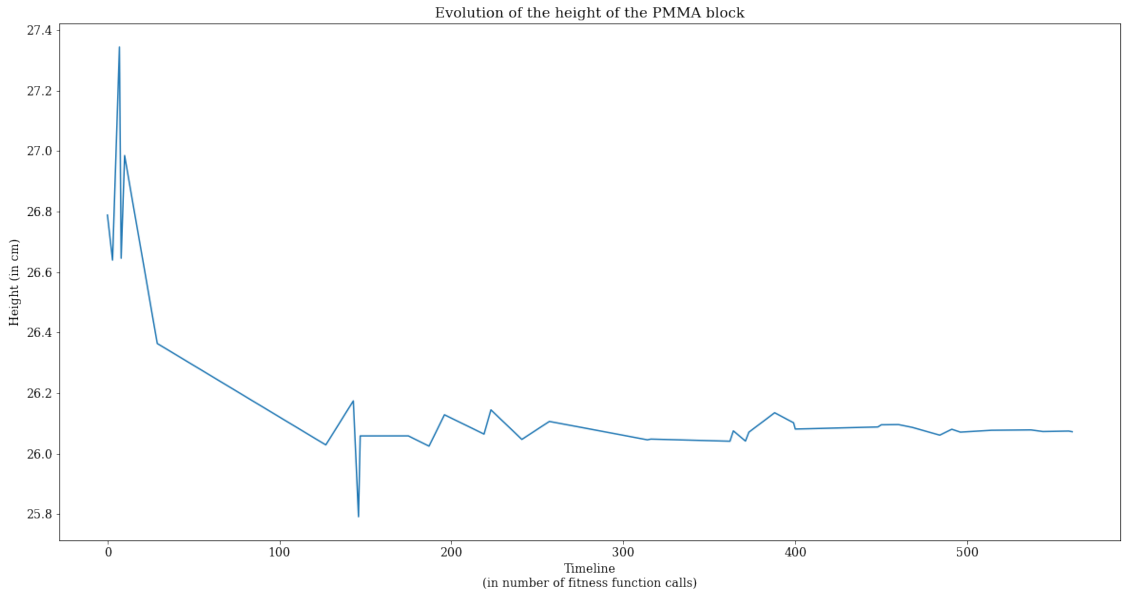
```
Block size (in cm): 32 26.072408952934417 4
Rotation angle (in degrees): -0.013443875732625506
```

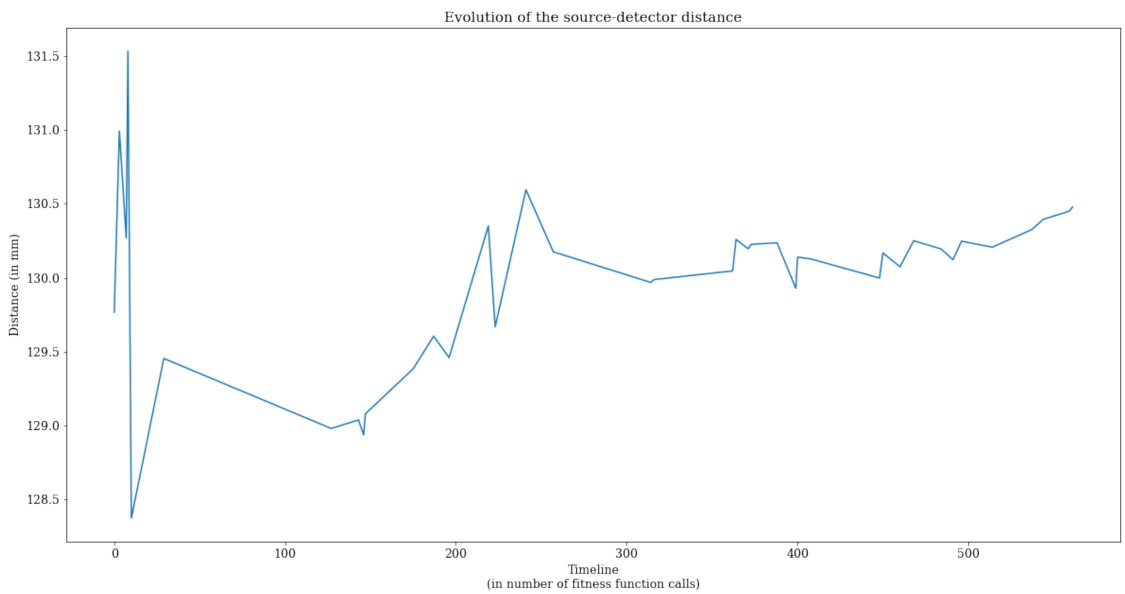
[37]:

```
evolution_parameters = np.array(evolution_parameters)

plt.figure(figsize= (20,10))
plt.title("Evolution of the PMMA block centre of mass along the three␣
↪carthesian coordinates")
plt.plot(evolution_parameters[:,0], evolution_parameters[:,1], label="x")
plt.plot(evolution_parameters[:,0], evolution_parameters[:,2], label="y")
plt.plot(evolution_parameters[:,0], evolution_parameters[:,3], label="z")
plt.xlabel("Timeline\n(in number of fitness function calls)")
plt.ylabel("Position (in cm)")
plt.legend()
plt.savefig('plots/PMMA_block_evolution_centre_of_mass.pdf')
plt.savefig('plots/PMMA_block_volution_centre_of_mass.png')

plt.figure(figsize= (20,10))
plt.title("Evolution of the rotation angle of the PMMA block")
plt.plot(evolution_parameters[:,0], evolution_parameters[:,4], label="angle")
plt.xlabel("Timeline\n(in number of fitness function calls)")
plt.ylabel("Rotation angle (in degrees)")
plt.savefig('plots/PMMA_block_evolution_centre_of_mass.pdf')
plt.savefig('plots/PMMA_block_evolution_centre_of_mass.png')

plt.figure(figsize= (20,10))
plt.title("Evolution of the height of the PMMA block")
plt.plot(evolution_parameters[:,0], evolution_parameters[:,5], label="angle")
plt.xlabel("Timeline\n(in number of fitness function calls)")
plt.ylabel("Height (in cm)")
plt.savefig('plots/PMMA_block_evolution_height.pdf')
plt.savefig('plots/PMMA_block_evolution_height.png')

plt.figure(figsize= (20,10))
plt.title("Evolution of the source-detector distance")
plt.plot(evolution_parameters[:,0], evolution_parameters[:,6])
plt.xlabel("Timeline\n(in number of fitness function calls)")
plt.ylabel("Distance (in mm)")
plt.savefig('plots/PMMA_block_evolution_SDD.pdf')
plt.savefig('plots/PMMA_block_evolution_SDD.png')
```


Evolution of the PMMA block centre of mass along the three carthesian coordinates
x
y
z
Position (in cm)
3
2
1
0
−1
0
100
200
300
400
500
Timeline
(in number of fitness function calls)


Evolution of the rotation angle of the PMMA block
Rotation angle (in degrees)
1
0
−1
−2
−3
0
100
200
300
400
500
Timeline
(in number of fitness function calls)





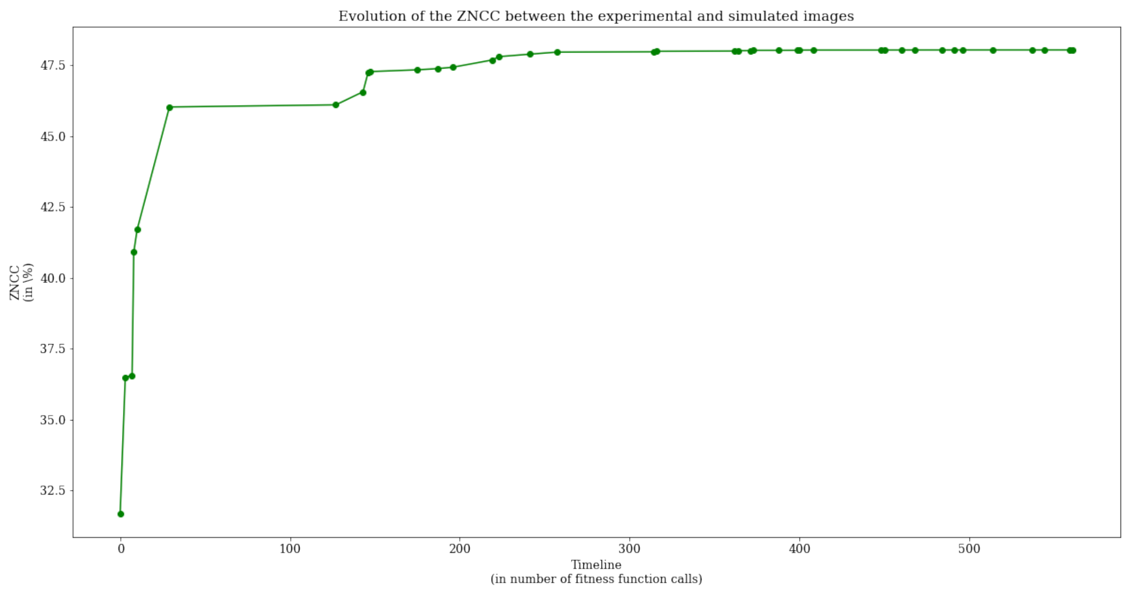
4.2 Apply the result of the optimisation

[38]:
```
# Compute an X-ray image
raw_x_ray_image = np.array(gvxr.computeXRayImage())

gvxr.displayScene()
```

```
# Compute the negative image as it is the case for the real image
x_ray_image = raw_x_ray_image * -1.

# Crop the image
x_ray_image = x_ray_image[179:2300,0:2848]

# Zero-mean, unit-variance normalistion
x_ray_image = standardisation(x_ray_image)
```

4.3 Plot the evolution of the quality of the simulated image

[39]:
```
plt.figure(figsize= (20,10))
plt.title("Evolution of the ZNCC between the experimental and simulated images")
plt.plot(evolution_zncc[:,0], 100.0 * evolution_zncc[:,1], "go-")
plt.xlabel("Timeline\n(in number of fitness function calls)")
plt.ylabel("ZNCC\n(in \%)")
plt.savefig('plots/PMMA_block_evolution_ZNCC1.pdf')
plt.savefig('plots/PMMA_block_evolution_ZNCC1.png')
```

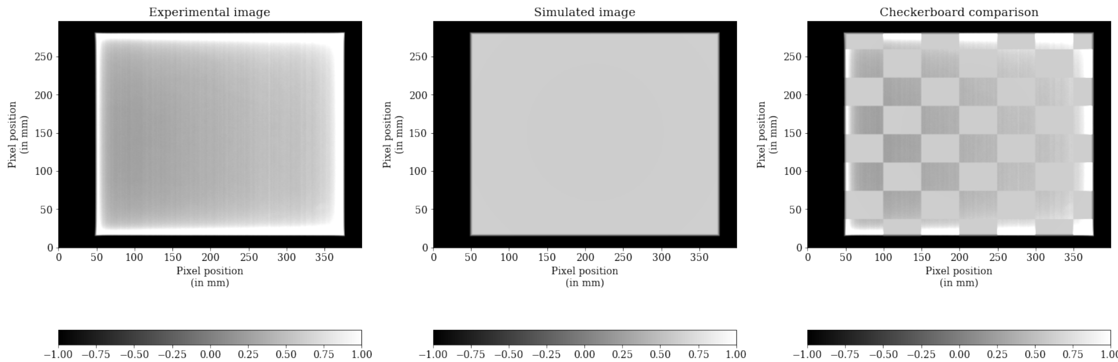


[40]:
```
def compareImages(real_image, x_ray_image, colorbar=True, rows = 1, cols = 3):
    comp_equalized = compare_images(real_image, x_ray_image,␣
 ↪method='checkerboard');

#     plt.figure(figsize= (20,10))
```

```
    # plt.suptitle("Image simulated using gVirtualXRay without the energy␣
↪response of the detector", y=1.02)

    plt.subplot(rows, cols, 1)
    plt.imshow(real_image, cmap="gray", vmin=-1, vmax=1,
               extent=[0,(real_image.shape[1]-1)*spacing[0],0,(real_image.
↪shape[0]-1)*spacing[1]])
    if colorbar:
        plt.colorbar(orientation='horizontal')
    plt.title("Experimental image")
    plt.xlabel("Pixel position\n(in mm)")
    plt.ylabel("Pixel position\n(in mm)")

    plt.subplot(rows, cols, 2)
    plt.imshow(x_ray_image, cmap="gray", vmin=-1, vmax=1,
               extent=[0,(real_image.shape[1]-1)*spacing[0],0,(real_image.
↪shape[0]-1)*spacing[1]])
    if colorbar:
        plt.colorbar(orientation='horizontal')
    plt.title("Simulated image")
    plt.xlabel("Pixel position\n(in mm)")
    plt.ylabel("Pixel position\n(in mm)")

    plt.subplot(rows, cols, 3)
    plt.imshow(comp_equalized, cmap="gray", vmin=-1, vmax=1,
               extent=[0,(real_image.shape[1]-1)*spacing[0],0,(real_image.
↪shape[0]-1)*spacing[1]])
    if colorbar:
        plt.colorbar(orientation='horizontal')
    plt.title("Checkerboard comparison")
    plt.xlabel("Pixel position\n(in mm)")
    plt.ylabel("Pixel position\n(in mm)")

    plt.tight_layout()
```

[41]:

```
plt.figure(figsize= (20,10))
compareImages(roi_real_image, x_ray_image)
plt.savefig('plots/PMMA_block_compare_images1.pdf')
plt.savefig('plots/PMMA_block_compare_images1.png')
```

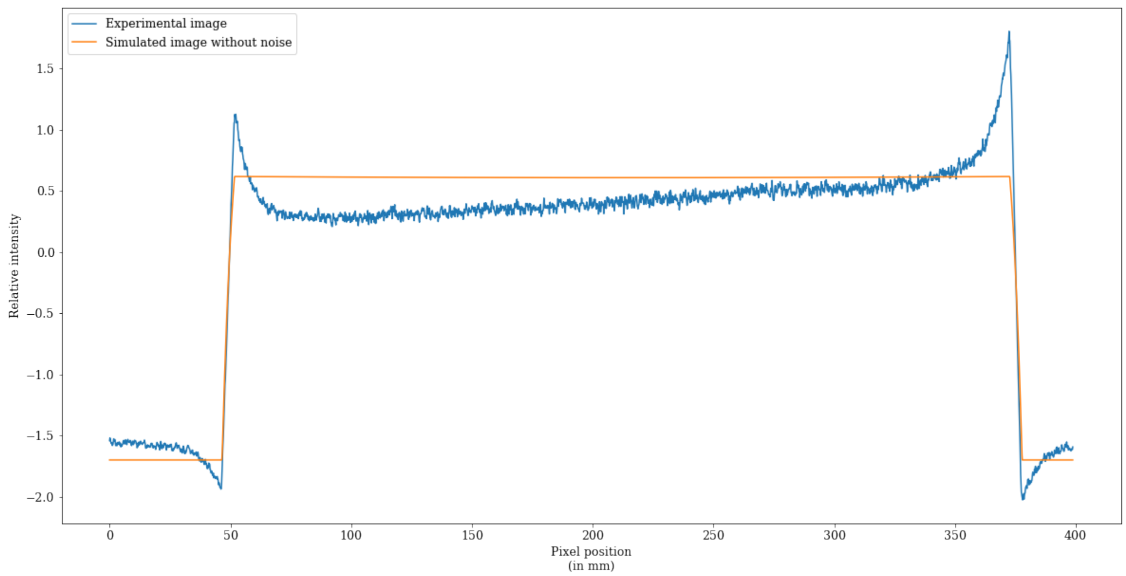


[42]:
```python
def drawHorizontalProfiles(real_image, x_ray_image, x_ray_image_noise=None):

    horizontal_profile_real_image = real_image[real_image.shape[1] // 2]
    horizontal_profile_simulated_image = x_ray_image[x_ray_image.shape[1] // 2]

    x_val = np.linspace(0.0,
                        spacing[0] * real_image.shape[1],
                        real_image.shape[1], endpoint=True)

    plt.plot(x_val, horizontal_profile_real_image, label="Experimental image")

    if x_ray_image_noise is not None:
        horizontal_profile_simulated_image_noise = x_ray_image_noise[x_ray_image_noise.shape[1] // 2]
        plt.plot(x_val, horizontal_profile_simulated_image_noise, label="Simulated image with noise")

    plt.plot(x_val, horizontal_profile_simulated_image, label="Simulated image without noise")

    plt.xlabel("Pixel position\n(in mm)")
    plt.ylabel("Relative intensity")
```

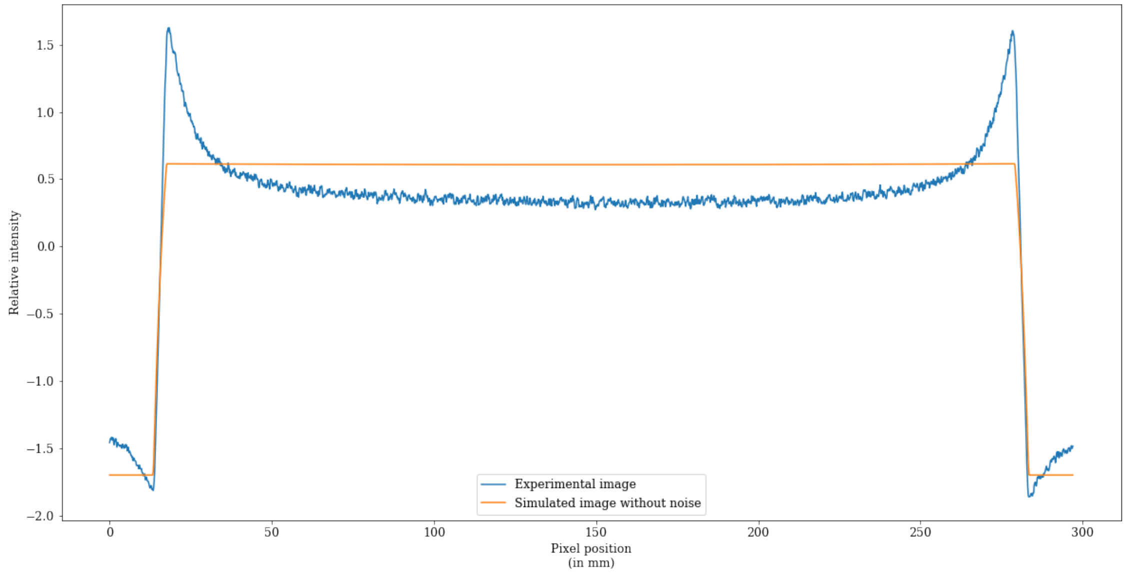
[43]:
```python
def drawVerticalProfiles(real_image, x_ray_image, x_ray_image_noise=None):
    vertical_profile_real_image = real_image[:,real_image.shape[0] // 2]
    vertical_profile_simulated_image = x_ray_image[:,x_ray_image.shape[0] // 2]

    x_val = np.linspace(0.0,
                        spacing[1] * real_image.shape[0],
                        real_image.shape[0], endpoint=True)

    plt.plot(x_val, vertical_profile_real_image, label="Experimental image")

    if x_ray_image_noise is not None:
```

```
        vertical_profile_simulated_image_noise = x_ray_image_noise[:
↪,x_ray_image_noise.shape[0] // 2]
        plt.plot(x_val, vertical_profile_simulated_image_noise,␣
↪label="Simulated image with noise")

    plt.plot(x_val, vertical_profile_simulated_image, label="Simulated image␣
↪without noise")

    plt.xlabel("Pixel position\n(in mm)")
    plt.ylabel("Relative intensity")
```

[44]:
```
plt.figure(figsize= (20,10))
drawHorizontalProfiles(roi_real_image, x_ray_image)
plt.legend()
plt.savefig('plots/PMMA_block_compare_horizontal_profile1.pdf')
plt.savefig('plots/PMMA_block_compare_horizontal_profile1.png')
```



[45]:
```
plt.figure(figsize= (20,10))
drawVerticalProfiles(roi_real_image, x_ray_image)
plt.legend()
plt.savefig('plots/PMMA_block_compare_vertical_profile1.pdf')
plt.savefig('plots/PMMA_block_compare_vertical_profile1.png')
```

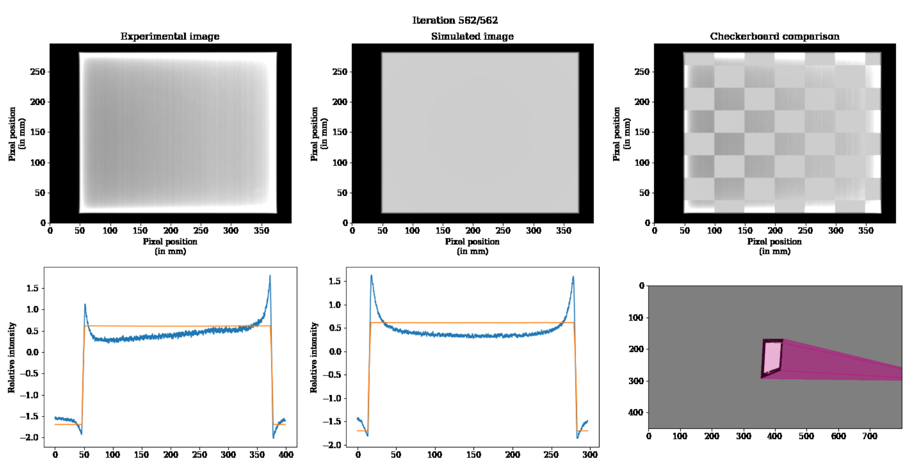


[46]:
```
def createAnimation(evolution_parameters, real_image):
    # Create the GIF file
    with imageio.get_writer("plots/PMMA_block_evolution.gif", mode='I') as␣
↪writer:

        # Store the PNG filenames
        png_filename_set = [];

        # Process all the images
        for i, [t, x, y, z, r, w, SDD] in enumerate(evolution_parameters):
            t = int(t)

            x_ray_image = imread("gVirtualXRay_output_data/PMMA_block_xray_" +␣
↪str(i) + ".tif")
            screenshot = imread("gVirtualXRay_output_data/
↪PMMA_block_screenshot_" + str(i) + ".tif")

            # Create the figure
            fig, axs = plt.subplots(nrows=2, ncols=3, figsize= (20,10))
            plt.suptitle("Iteration " + str(t+1) + "/" +␣
↪str(int(evolution_parameters[-1][0]+1)))
            compareImages(real_image, x_ray_image, False, 2, 3)

            plt.subplot(234)
            drawHorizontalProfiles(real_image, x_ray_image)

            plt.subplot(235)
            drawVerticalProfiles(real_image, x_ray_image)
```

```
            plt.subplot(236)
            plt.imshow(screenshot)

            # Save the figure as a PNG file
            plt.savefig("temp.png")

            # Close the figure
            plt.close()

            # Open the PNG file with imageio and add it to the GIF file
            image = imageio.imread("temp.png")
            writer.append_data(image)

            # Delete the PNG file
            os.remove("temp.png");

        for i in range(15):
            writer.append_data(image)
```

[47]:
```
if not os.path.exists("plots/PMMA_block_evolution.gif"):
    createAnimation(evolution_parameters, roi_real_image)
```

[48]:
```
with open('./plots/PMMA_block_evolution.gif','rb') as f:
    display(Image(data=f.read(), format='png', width=1000))
```



5 Post-processing using image sharpening

We can see from the real image that an image sharpening filter was applied. We will implement one and optimise its parameters.

[49]:
```python
def sharpen(image, ksize, alpha, shift, scale):
    details = image - gaussian(image, ksize)

    return scale * (shift + image) + alpha * details
```

[50]:
```python
def postProcessing(raw_x_ray_image, sigma1, sigma2, alpha, shift, scale):
    # Compute an X-ray image
    x_ray_image = sharpen(raw_x_ray_image, [sigma1, sigma2], alpha, shift,␣
 ↪scale)

    # Compute the negative image as it is the case for the real image
    x_ray_image *= -1.

    # Crop the image
    x_ray_image = x_ray_image[179:2300,0:2848]

    # Zero-mean, unit-variance normalistion
    x_ray_image = standardisation(x_ray_image)

    return x_ray_image
```

5.1 Define an objective function

[51]:
```python
def objectiveFunctionSharpen(parameters):

    global best_fitness
    global best_fitness_id
    global fitness_function_call_id
    global sharpened_evolution_zncc
    global sharpened_evolution_parameters

    global raw_x_ray_image

    # Retrieve the parameters
    sigma1, sigma2, alpha, shift, scale = parameters

    # Compute an X-ray image
    x_ray_image = postProcessing(raw_x_ray_image, sigma1, sigma2, alpha, shift,␣
 ↪scale)

    # Return the objective
```

```
    objective = math.sqrt(mean_squared_error(roi_real_image, x_ray_image))

    # The block below is not necessary for the registration.
    # It is used to save the data to create animations.
    if best_fitness > objective:

        imwrite("gVirtualXRay_output_data/PMMA_block_sharpened_xray_" +␣
↪str(best_fitness_id) + ".tif", x_ray_image.astype(np.single))

        zncc_value = zncc(roi_real_image, x_ray_image)
        sharpened_evolution_zncc.append([fitness_function_call_id, zncc_value])
        sharpened_evolution_parameters.append([fitness_function_call_id,␣
↪sigma1, sigma2, alpha, shift, scale])

        best_fitness = objective
        best_fitness_id += 1

    fitness_function_call_id += 1

    return objective
```

5.2 Minimise the objective function

[52]:
```
old_zncc = evolution_zncc[-1][1]
```

[53]:
```
best_fitness = sys.float_info.max;
best_fitness_id = 0;
fitness_function_call_id = 0
sharpened_evolution_zncc = []
sharpened_evolution_parameters = []

sigma1 = 100
sigma2 = 100
alpha = 2.5
shift = 0
scale = 1

xl = [3, 3, 0, -5, 0]
xu = [200, 200, 15, 5, 2]
x_init = [sigma1, sigma2, alpha, shift, scale]
```

[54]:
```
# The registration has already been performed. Load the results.
if os.path.isfile("gVirtualXRay_output_data/PMMA_block_sharpen.dat") and \
        os.path.isfile("gVirtualXRay_output_data/
↪PMMA_block_sharpen_evolution_zncc.dat") and \
```

```python
        os.path.isfile("gVirtualXRay_output_data/
↪PMMA_block_sharpen_evolution_parameters.dat"):

    temp = np.loadtxt("gVirtualXRay_output_data/PMMA_block_sharpen.dat")
    sigma1 = temp[0]
    sigma2 = temp[1]
    alpha = temp[2]
    shift = temp[3]
    scale = temp[4]

    sharpened_evolution_zncc = np.loadtxt("gVirtualXRay_output_data/
↪PMMA_block_sharpen_evolution_zncc.dat")
    sharpened_evolution_parameters = np.loadtxt("gVirtualXRay_output_data/
↪PMMA_block_sharpen_evolution_parameters.dat")

else:
    # Optimise
    timeout_in_sec = 20 * 60 # 20 minutes
    opts = cma.CMAOptions()
    opts.set('tolfun', 1e-5)
    opts['tolx'] = 1e-5
    opts['timeout'] = timeout_in_sec
    opts['bounds'] = [xl, xu]
    opts['CMA_stds'] = []

    for min_val, max_val in zip(opts['bounds'][0], opts['bounds'][1]):
        opts['CMA_stds'].append(abs(max_val - min_val) * 0.05)

    # Optimise
    es = cma.CMAEvolutionStrategy(x_init, 0.5, opts)
    es.optimize(objectiveFunctionSharpen)

    # Save the parameters
    sigma1, sigma2, alpha, shift, scale = es.result.xbest
    np.savetxt("gVirtualXRay_output_data/PMMA_block_sharpen.dat", [sigma1,␣
↪sigma2, alpha, shift, scale], header='sigma,alpha,shift,scale')

    sharpened_evolution_zncc = np.array(sharpened_evolution_zncc)
    np.savetxt("gVirtualXRay_output_data/PMMA_block_sharpen_evolution_zncc.
↪dat", sharpened_evolution_zncc, header='t,ZNCC')

    sharpened_evolution_parameters = np.array(sharpened_evolution_parameters)
    np.savetxt("gVirtualXRay_output_data/
↪PMMA_block_sharpen_evolution_parameters.dat",␣
↪sharpened_evolution_parameters, header='t,sigma,alpha')
```

```
# Release memory
del es;
```

```
(4_w,8)-aCMA-ES (mu_w=2.6,w_1=52%) in dimension 5 (seed=259176, Wed Mar  2
17:22:46 2022)
Iterat #Fevals   function value  axis ratio  sigma  min&max std  t[m:s]
    1      8 2.498434291739727e-01 1.0e+00 4.19e-01  4e-02  4e+00 0:32.9
    2     16 2.252714433709117e-01 1.2e+00 4.11e-01  4e-02  4e+00 1:06.5
    3     24 2.079673871082384e-01 1.5e+00 5.14e-01  5e-02  5e+00 1:39.2
    4     32 1.834984491429550e-01 1.7e+00 6.90e-01  7e-02  7e+00 2:10.3
    5     40 1.714324564099746e-01 1.7e+00 7.53e-01  7e-02  7e+00 2:41.1
    6     48 1.725303607376057e-01 1.5e+00 7.95e-01  8e-02  8e+00 3:11.3
    7     56 1.707547319299024e-01 1.4e+00 7.48e-01  7e-02  7e+00 3:40.3
    8     64 1.718454931680271e-01 1.4e+00 7.42e-01  7e-02  7e+00 4:09.4
    9     72 1.723681981409343e-01 1.5e+00 5.86e-01  5e-02  5e+00 4:38.4
   10     80 1.713725081418439e-01 1.5e+00 5.34e-01  5e-02  5e+00 5:08.0
   11     88 1.705631782333739e-01 1.7e+00 5.06e-01  5e-02  4e+00 5:39.2
   12     96 1.707695938523146e-01 1.9e+00 4.11e-01  4e-02  3e+00 6:12.2
   13    104 1.704705959136210e-01 1.8e+00 3.53e-01  3e-02  3e+00 6:43.6
   14    112 1.703732021454532e-01 2.0e+00 3.27e-01  3e-02  3e+00 7:15.3
   15    120 1.704048877327923e-01 2.4e+00 2.87e-01  3e-02  2e+00 7:47.2
   16    128 1.704522875975855e-01 3.0e+00 2.42e-01  2e-02  2e+00 8:19.0
   17    136 1.703711085137480e-01 3.1e+00 2.08e-01  2e-02  1e+00 8:51.1
   18    144 1.704025540118071e-01 3.2e+00 1.94e-01  2e-02  1e+00 9:23.3
   19    152 1.702325928758017e-01 3.6e+00 2.13e-01  2e-02  2e+00 9:56.1
   20    160 1.701110137975442e-01 3.9e+00 2.61e-01  3e-02  2e+00 10:28.7
   21    168 1.700548167152631e-01 4.2e+00 2.92e-01  3e-02  3e+00 11:02.2
   22    176 1.700099190805450e-01 4.5e+00 3.03e-01  3e-02  3e+00 11:36.1
   23    184 1.700025564239929e-01 4.8e+00 3.48e-01  3e-02  3e+00 12:10.5
   24    192 1.698880490802685e-01 4.9e+00 3.91e-01  4e-02  3e+00 12:44.9
   25    200 1.699289835615872e-01 6.1e+00 3.56e-01  3e-02  3e+00 13:19.6
   26    208 1.698517819236453e-01 6.5e+00 3.70e-01  3e-02  3e+00 13:54.3
   27    216 1.698676987584920e-01 9.4e+00 3.71e-01  4e-02  3e+00 14:29.3
   28    224 1.698756867697323e-01 1.1e+01 3.70e-01  4e-02  3e+00 15:03.7
   29    232 1.698763446537101e-01 1.0e+01 4.03e-01  5e-02  4e+00 15:38.1
   30    240 1.698389234532609e-01 1.1e+01 3.93e-01  5e-02  3e+00 16:13.5
   31    248 1.698472034327853e-01 1.2e+01 3.30e-01  4e-02  3e+00 16:48.8
   32    256 1.698330614223806e-01 1.4e+01 2.87e-01  3e-02  3e+00 17:24.7
   33    264 1.698556803976207e-01 1.5e+01 2.46e-01  3e-02  2e+00 18:00.7
   34    272 1.698342130066243e-01 1.5e+01 2.54e-01  3e-02  2e+00 18:36.7
   35    280 1.698341417183026e-01 1.7e+01 2.78e-01  4e-02  2e+00 19:12.8
   36    288 1.698444836959503e-01 2.2e+01 2.57e-01  4e-02  2e+00 19:48.7
   37    296 1.698316904785758e-01 2.5e+01 2.16e-01  3e-02  2e+00 20:23.9
```

[55]:
```
new_zncc = sharpened_evolution_zncc[-1][1]
```

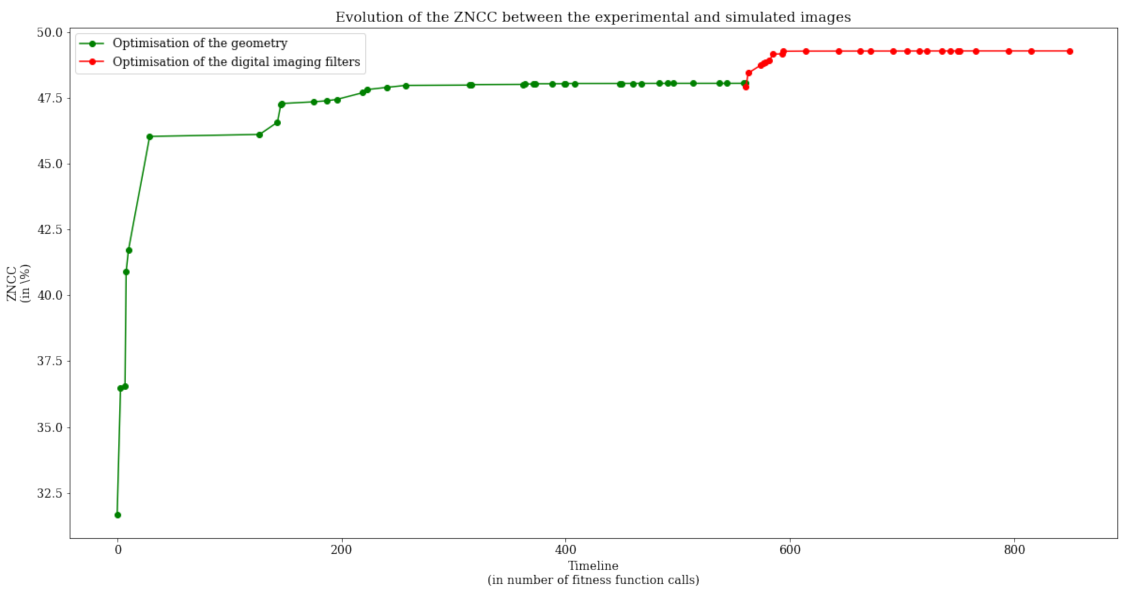
[56]:
```
print("ZNCC before sharpening:", str(100 * old_zncc) + "%")
print("ZNCC after sharpening:", str(100 * new_zncc) + "%")
```

```
ZNCC before sharpening: 48.04822504520416%
ZNCC after sharpening: 49.27895963191986%
```

[57]:
```
print(evolution_zncc[-1,0])
```

```
561.0
```

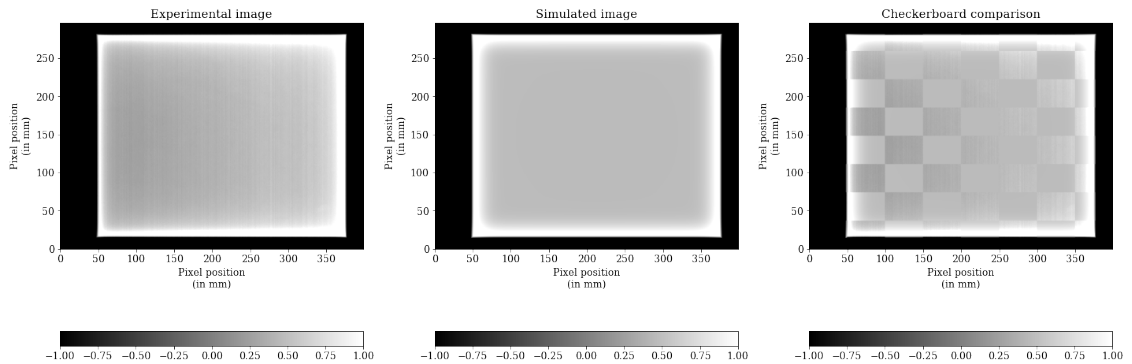
[58]:
```
plt.figure(figsize= (20,10))
plt.title("Evolution of the ZNCC between the experimental and simulated images")
plt.plot(evolution_zncc[:,0], 100 * evolution_zncc[:,1], "go-",␣
↪label="Optimisation of the geometry")
plt.plot(sharpened_evolution_zncc[:,0] + evolution_zncc[-1,0], 100 *␣
↪sharpened_evolution_zncc[:,1], "ro-", label="Optimisation of the digital␣
↪imaging filters")
plt.xlabel("Timeline\n(in number of fitness function calls)")
plt.ylabel("ZNCC\n(in \%)")
plt.legend()
plt.savefig('plots/PMMA_block_evolution_ZNCC2.pdf')
plt.savefig('plots/PMMA_block_evolution_ZNCC2.png')
```



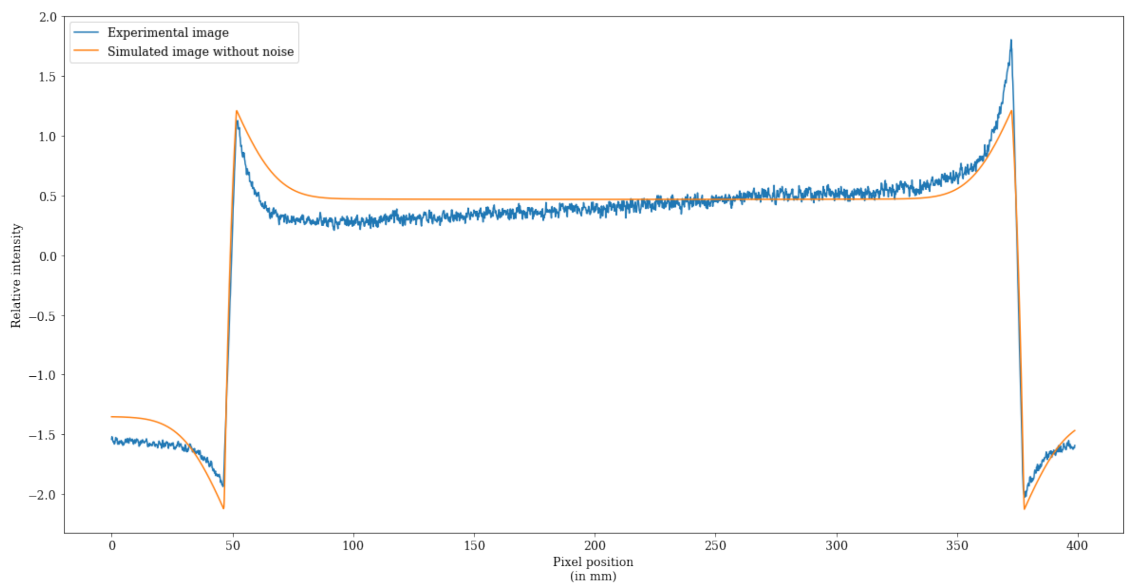
5.3 Apply the result of the optimisation

[59]:
```
x_ray_image = postProcessing(raw_x_ray_image, sigma1, sigma2, alpha, shift,␣
↪scale)
```

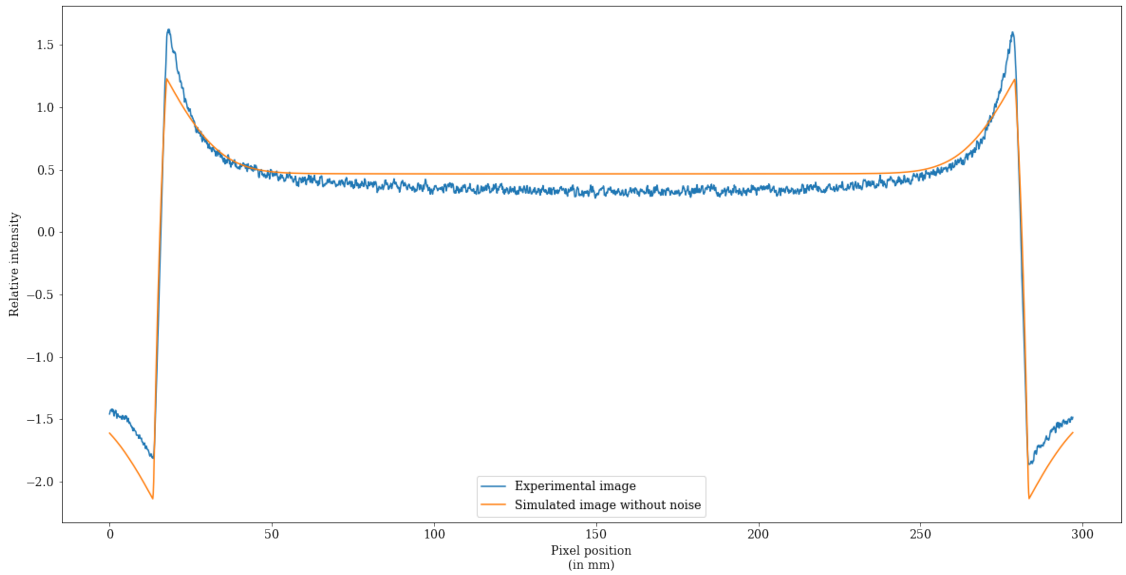
[60]:
```
plt.figure(figsize= (20,10))
compareImages(roi_real_image, x_ray_image)
plt.savefig('plots/PMMA_block_compare_images2.pdf')
plt.savefig('plots/PMMA_block_compare_images2.png')
```



[61]:
```
plt.figure(figsize= (20,10))
drawHorizontalProfiles(roi_real_image, x_ray_image)
plt.legend()
plt.savefig('plots/PMMA_block_compare_horizontal_profile2.pdf')
plt.savefig('plots/PMMA_block_compare_horizontal_profile2.png')
```



[62]:
```python
plt.figure(figsize= (20,10))
drawVerticalProfiles(roi_real_image, x_ray_image)
plt.legend()
plt.savefig('plots/PMMA_block_compare_vertical_profile2.pdf')
plt.savefig('plots/PMMA_block_compare_vertical_profile2.png')
```



[63]:
```python
print([sigma1, sigma2], alpha)
```

```
[103.59032031134267, 109.26224767704798] 1.014424469101441
```

[64]:
```python
print(shift, scale)
```

```
1.6671140281800274 1.0561354748958112
```

6 Poisson noise

gVirtualXRay exploits the Beer-Lambert law. It does not take into account photon noise. We can, however, estimate the amount of noise in the real X-ray radiograph and replicate it. First we need to build a model of the Poisson noise:

6.1 Extract noise properties from the ground truth

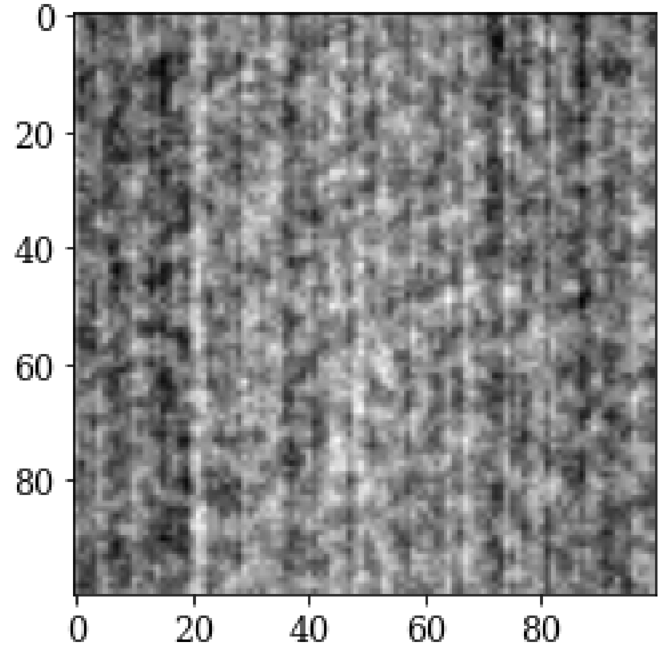
[65]:
```
def applyNoise(img, bias, gain, scale):

    # Poisson noise
    temp_xray_image = (img + (bias + 1)) * gain
    temp = np.random.poisson(temp_xray_image).astype(np.single)
    temp /= gain
    temp -= bias + 1

    # Noise map
    noise_map = img - temp
    noise_map *= scale;
    noisy_image = img + noise_map

    return noisy_image, noise_map
```

We compute the amplitude of the pixel values in an homogeneous region of interest (ROI) extracted from the ground truth.

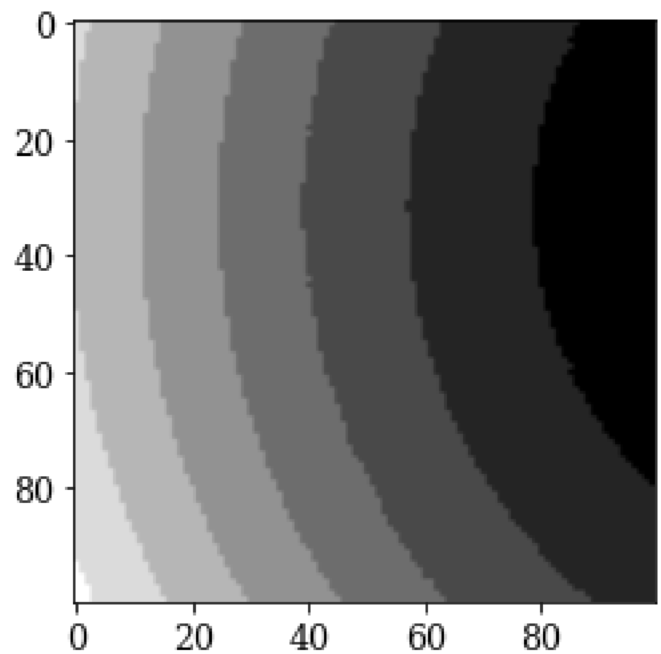
[66]:
```
centre = [int(roi_real_image.shape[0]/2), int(roi_real_image.shape[1]/2)]
quarter_size = [int(50), int(50)]

reference_noise_ROI = copy.deepcopy(roi_real_image)[centre[0]-quarter_size[0]:
 ↪centre[0]+quarter_size[0], centre[1]-quarter_size[1]:
 ↪centre[1]+quarter_size[1]]
test_noise_ROI = copy.deepcopy(x_ray_image)[centre[0]-quarter_size[0]:
 ↪centre[0]+quarter_size[0], centre[1]-quarter_size[1]:
 ↪centre[1]+quarter_size[1]]
```

[67]:
```
plt.figure(figsize= (20,10))
plt.figure()
plt.imshow(reference_noise_ROI, cmap="gray")
plt.savefig('plots/PMMA_block_ROI_noise_groundtruth.pdf')
plt.savefig('plots/PMMA_block_ROI_noise_groundtruth.png')

plt.figure(figsize= (20,10))
plt.figure()
plt.imshow(test_noise_ROI, cmap="gray")
plt.savefig('plots/PMMA_block_ROI_noise_simulation.pdf')
plt.savefig('plots/PMMA_block_ROI_noise_simulation.png')
```

```
<Figure size 1440x720 with 0 Axes>
```



<Figure size 1440x720 with 0 Axes>



[68]:
```
min_noise_value = np.min(reference_noise_ROI)
max_noise_value = np.max(reference_noise_ROI)
mean_noise_value = np.mean(reference_noise_ROI)
stddev_noise_value = np.std(reference_noise_ROI)
roi_noise_range = max_noise_value - min_noise_value

print("min_noise_value", min_noise_value)
print("max_noise_value", max_noise_value)
print("mean_noise_value", mean_noise_value)
print("stddev_noise_value", stddev_noise_value)
print("roi_noise_range", roi_noise_range)
```

```
min_noise_value 0.2985593
max_noise_value 0.4805349
mean_noise_value 0.38856715
stddev_noise_value 0.028201768
roi_noise_range 0.1819756
```

6.2 Define an objective function

[69]:
```
def fitnessFunctionNoise(x):
    global roi_stddev
    global x_ray_image_response

    bias = x[0]
    gain = x[1]
    scale = x[2]

    # Extract a ROI from the test image where no object is
    test_noise_ROI = x_ray_image   #copy.
 ↪deepcopy(standardisation(x_ray_image)[1000:2000,1000:2000])
    test_noise_ROI = copy.deepcopy(x_ray_image)[centre[0]-quarter_size[0]:
 ↪centre[0]+quarter_size[0], centre[1]-quarter_size[1]:
 ↪centre[1]+quarter_size[1]]

    # Apply the noise model
    noisy_image, noise_map = applyNoise(test_noise_ROI, bias, gain, scale)

    # Compute the amplitude of the noise
    noise_range = noise_map.max() - noise_map.min()

    # Square difference
    diff = roi_noise_range - noise_range

    objective = diff * diff
```

```
    return objective
```

6.3 Minimise the objective function

We minimise the objective value using a global optimisation algorithm

[70]:
```
# The registration has already been performed. Load the results.
if os.path.isfile("gVirtualXRay_output_data/PMMA_block_noise.dat"):

    noise_bias, noise_gain, noise_scale = np.loadtxt("gVirtualXRay_output_data/
 ↪PMMA_block_noise.dat")

# Optimise
else:

    # Initialise the values
    test_noise_ROI = x_ray_image  #copy.deepcopy(x_ray_image[1000:2000,1000:
 ↪2000])
    bias = -x_ray_image.min();
    gain = 1.0;
    scale = 1;

    x0 = [bias, gain, scale];

    bounds = [
        [-x_ray_image.min(),   0.0001, -10.0],
        [ 15.0, 15.0, 10.0]
    ];

    opts = cma.CMAOptions()
    opts.set('tolfun', 1e-10);
    opts['tolx'] = 1e-10;
    opts['bounds'] = bounds;
    opts['CMA_stds'] = [0.25, 0.25, 0.25];

    es = cma.CMAEvolutionStrategy(x0, 0.25, opts);
    es.optimize(fitnessFunctionNoise);

    # Save the parameters
    noise_bias = es.result.xbest[0];
    noise_gain = es.result.xbest[1];
    noise_scale = es.result.xbest[2];

    np.savetxt("gVirtualXRay_output_data/PMMA_block_noise.dat", [noise_bias,␣
 ↪noise_gain, noise_scale], header='bias,gain,scale')
```

```
    # Release memory
    del es;
```

```
(3_w,7)-aCMA-ES (mu_w=2.3,w_1=58%) in dimension 3 (seed=311834, Wed Mar  2
17:43:18 2022)
Iterat #Fevals   function value  axis ratio  sigma  min&max std  t[m:s]
    1      7 1.066550979614258e+02 1.0e+00 2.66e-01  6e-02  8e-02 0:00.0
    2     14 9.096156311035156e+01 1.5e+00 3.68e-01  7e-02  1e-01 0:00.0
    3     21 4.803373336791992e+01 1.7e+00 4.93e-01  1e-01  2e-01 0:00.1
  100    700 9.521744459561887e-08 1.1e+03 1.62e-01  2e-04  2e-02 0:02.5
  200   1400 1.131974514123613e-10 2.0e+03 2.56e-02  2e-06  3e-04 0:04.9
  249   1743 6.417089082333405e-14 7.6e+03 4.98e-03  7e-08  2e-05 0:06.1
```

6.4 Apply the result of the optimisation

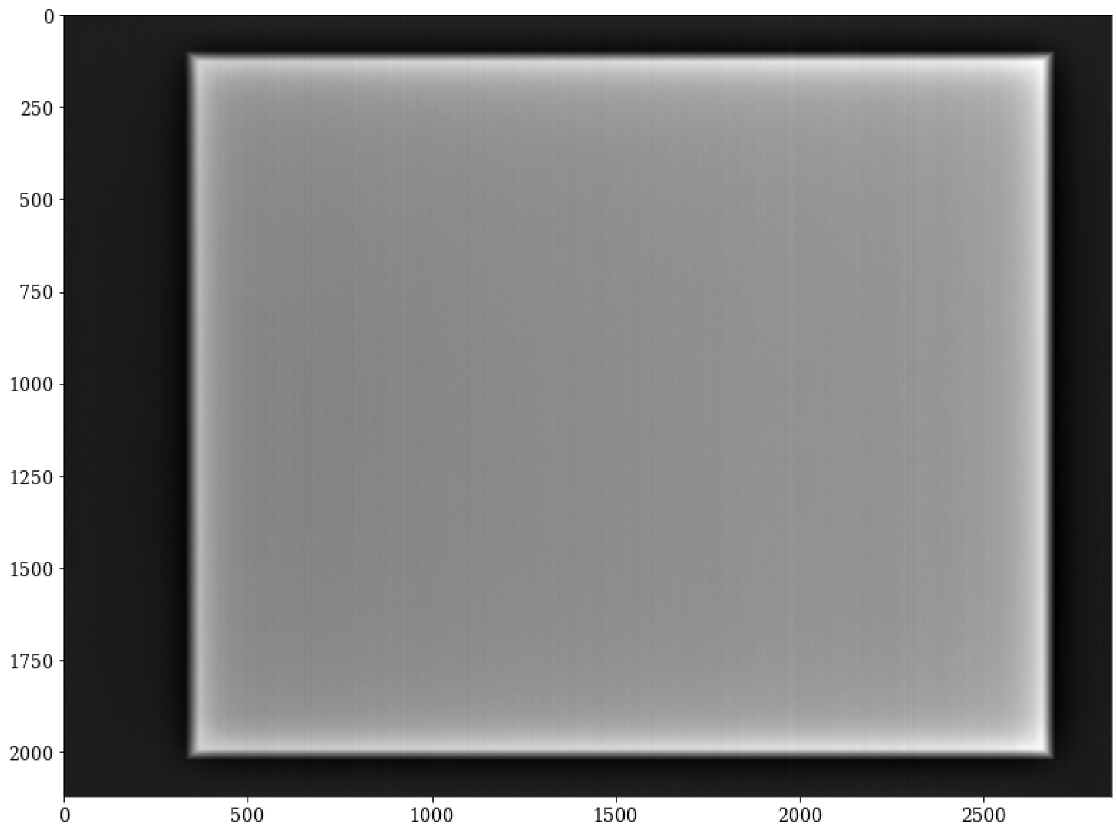
[71]:
```
print(noise_bias, noise_gain, noise_scale)
x_ray_image_noise, noise_map = applyNoise(x_ray_image, noise_bias, noise_gain,␣
 ↪noise_scale)

min_noise_value = np.min(reference_noise_ROI)
max_noise_value = np.max(reference_noise_ROI)
mean_noise_value = np.mean(reference_noise_ROI)
stddev_noise_value = np.std(reference_noise_ROI)

print("min_noise_value", min_noise_value)
print("max_noise_value", max_noise_value)
print("mean_noise_value", mean_noise_value)
print("stddev_noise_value", stddev_noise_value)
print("range", max_noise_value - min_noise_value)

min_noise_value = np.min(noise_map)
max_noise_value = np.max(noise_map)
mean_noise_value = np.mean(noise_map)
stddev_noise_value = np.std(noise_map)

print("min_noise_value", min_noise_value)
print("max_noise_value", max_noise_value)
print("mean_noise_value", mean_noise_value)
print("stddev_noise_value", stddev_noise_value)
print("range", max_noise_value - min_noise_value)
```

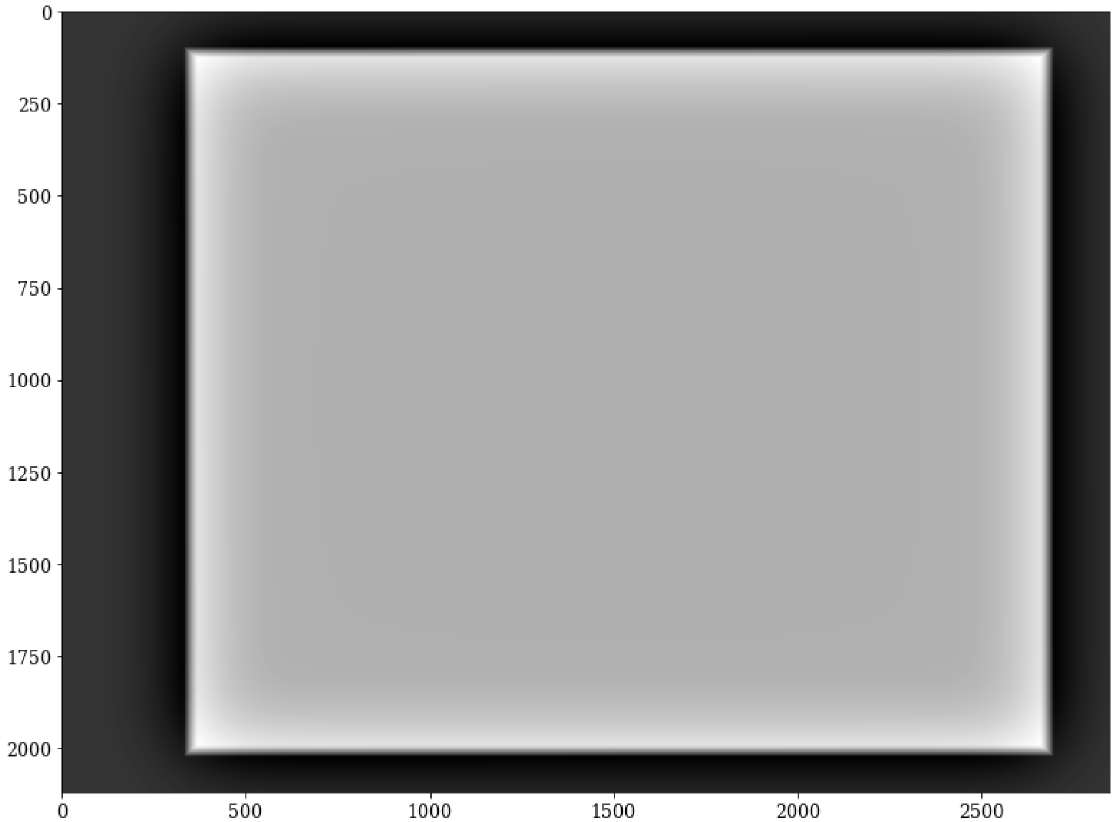
```
2.150670225693114 1.4811896904422113 -0.016846119703515877
min_noise_value 0.2985593
max_noise_value 0.4805349
mean_noise_value 0.38856715
stddev_noise_value 0.028201768
```

```
range 0.1819756
min_noise_value -0.08003755
max_noise_value 0.1672303
mean_noise_value -1.739534e-05
stddev_noise_value 0.0245761
range 0.24726784
```

Plot the images

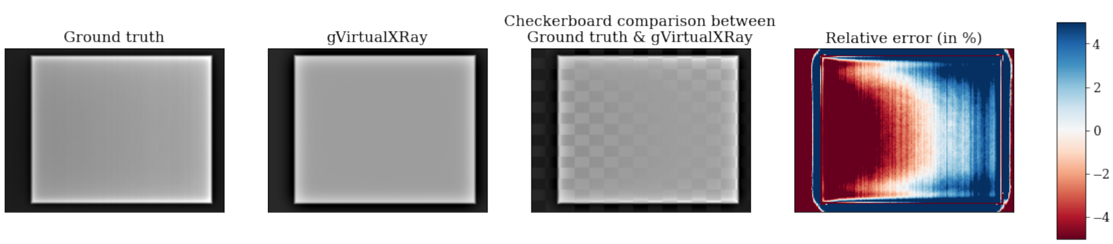
[72]:
```
plt.figure(figsize= (20,10))
plt.imshow(roi_real_image, cmap="gray")

plt.figure(figsize= (20,10))
plt.imshow(x_ray_image, cmap="gray")

plt.figure(figsize= (20,10))
plt.imshow(x_ray_image_noise, cmap="gray")
plt.savefig('plots/PMMA_block_simulated_image_with_noise.pdf')
plt.savefig('plots/PMMA_block_simulated_image_with_noise.png')
```




0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500

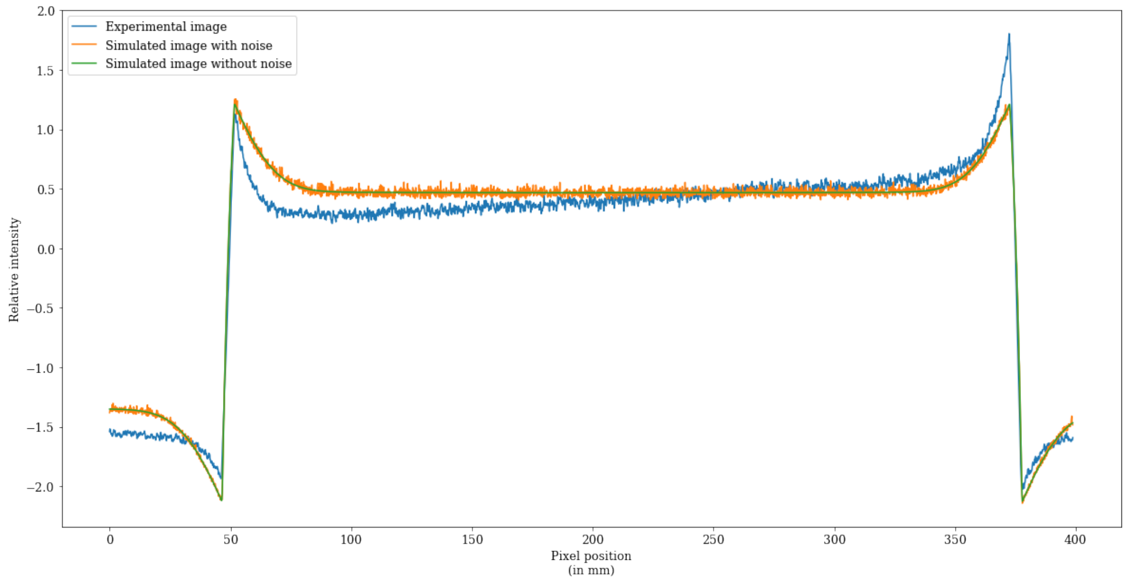

```
[73]: font = {'size'   : 12.5
       }
matplotlib.rc('font', **font)
```

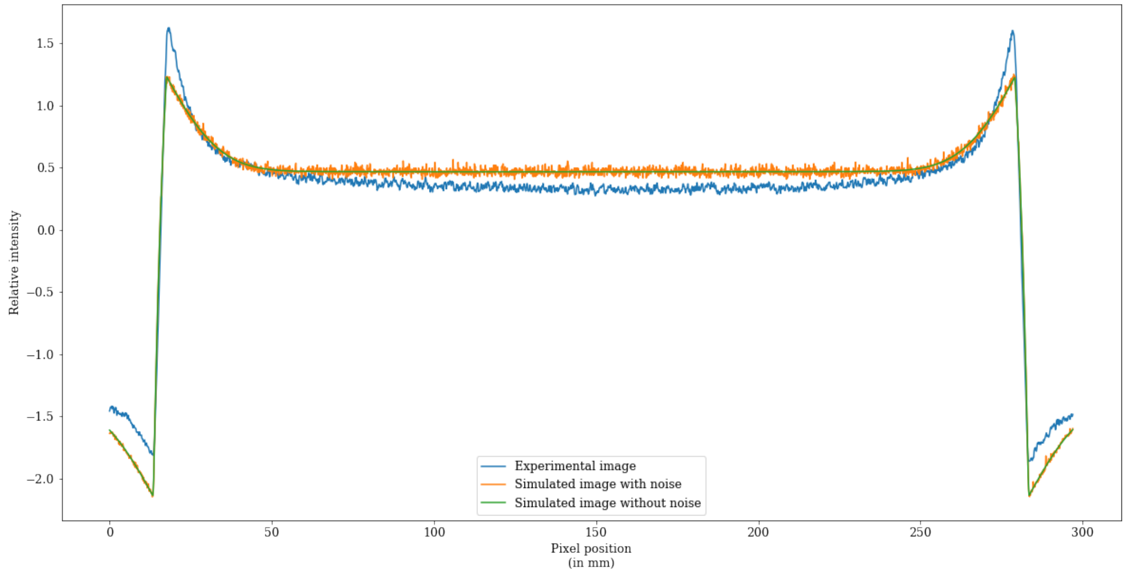
```
[74]: fullCompareImages(roi_real_image,
                  x_ray_image_noise,
                  "gVirtualXRay\n with integration on GPU",
                  "plots/PMMA_block_full_comparison_integration_GPU", vmin=-2,␣
  ↪vmax = 2,
                 avoid_div_0=True)
```



Plot the profiles

[75]:
```
roi_raw_real_image = standardisation(roi_raw_real_image)

plt.figure(figsize= (20,10))
drawHorizontalProfiles(roi_real_image, x_ray_image, x_ray_image_noise)

plt.legend()
plt.savefig('plots/PMMA_block_compare_horizontal_profile3.pdf')
plt.savefig('plots/PMMA_block_compare_horizontal_profile3.png')

plt.figure(figsize= (20,10))
drawVerticalProfiles(roi_real_image, x_ray_image, x_ray_image_noise)
plt.legend()
plt.savefig('plots/PMMA_block_compare_horizontal_profile3.pdf')
plt.savefig('plots/PMMA_block_compare_horizontal_profile3.png')
```





Quantify the similarities and differences

[76]:
```
# Avoid div by 0
offset1 = min(roi_real_image.min(), x_ray_image_noise.min())
offset2 = 0.01 * (roi_real_image.max() - roi_real_image.min())
offset = offset2 - offset1

MAPE = mape(roi_real_image + offset, x_ray_image_noise + offset)
ZNCC = np.mean((roi_real_image - roi_real_image.mean()) / roi_real_image.std()␣
↪* (x_ray_image_noise - x_ray_image_noise.mean()) / x_ray_image_noise.std())
SSIM = ssim(roi_real_image, x_ray_image_noise, data_range=roi_real_image.max()␣
↪- roi_real_image.min())

print("MAPE with noise:", "{0:0.2f}".format(100 * MAPE) + "%")
print("ZNCC with noise", "{0:0.2f}".format(100 * ZNCC) + "%")
print("SSIM with noise:", "{0:0.2f}".format(SSIM))

# Avoid div by 0
offset1 = min(roi_real_image.min(), x_ray_image_noise.min())
offset2 = 0.01 * (roi_real_image.max() - roi_real_image.min())
offset = offset2 - offset1

MAPE = mape(roi_real_image + offset, x_ray_image + offset)
ZNCC = np.mean((roi_real_image - roi_real_image.mean()) / roi_real_image.std()␣
↪* (x_ray_image - x_ray_image.mean()) / x_ray_image.std())
SSIM = ssim(roi_real_image, x_ray_image, data_range=roi_real_image.max() -␣
↪roi_real_image.min())
```

```
print("MAPE without noise:", "{0:0.2f}".format(100 * MAPE) + "%")
print("ZNCC without noise", "{0:0.2f}".format(100 * ZNCC) + "%")
print("SSIM without noise:", "{0:0.2f}".format(SSIM))
```

```
MAPE with noise: 10.75%
ZNCC with noise 98.53%
SSIM with noise: 0.90
MAPE without noise: 10.69%
ZNCC without noise 98.56%
SSIM without noise: 0.93
```

7 Estimate the simulation time

Get the total number of triangles

[77]:
```
number_of_triangles = gvxr.getNumberOfPrimitives("PMMA block")
```

Compute an X-ray image 25 times (to gather performance statistics)

[78]:
```
# gvxr.enableArtefactFilteringOnCPU()
gvxr.enableArtefactFilteringOnGPU()
# gvxr.disableArtefactFiltering() # Spere inserts are missing with GPU␣
↪integration when a outer surface is used for the matrix

runtimes = []

for i in range(25):
    start_time = datetime.datetime.now()

    raw_x_ray_image = np.array(gvxr.computeXRayImage())

    # Compute an X-ray image
    x_ray_image = sharpen(raw_x_ray_image, [sigma1, sigma2], alpha, shift,␣
↪scale)

    # Compute the negative image as it is the case for the real image
    x_ray_image *= -1.

    # Crop the image
    x_ray_image = x_ray_image[179:2300,0:2848]

    # Zero-mean, unit-variance normalistion
    x_ray_image = standardisation(x_ray_image)

    # Apply the noise model
```

```
    noisy_image, noise_map = applyNoise(x_ray_image, noise_bias, noise_gain,␣
↪noise_scale)

    end_time = datetime.datetime.now()
    delta_time = end_time - start_time
    runtimes.append(delta_time.total_seconds() * 1000)
```

[79]:
```
runtime_avg = round(np.mean(runtimes))
runtime_std = round(np.std(runtimes))
```

Print a row of the table for the paper

[80]:
```
print("Registration PMMA block & Real image & " +
      "{0:0.2f}".format(100 * MAPE) + "\\%    &    " +
      "{0:0.2f}".format(100 * ZNCC) + "\\%    &    " +
      "{0:0.2f}".format(SSIM) + "    &    $" +
      str(raw_x_ray_image.shape[1]) + " \\times " + str(raw_x_ray_image.
↪shape[0]) + "$    &    " +
      str(number_of_triangles) + "    &    " +
      "N/A    &    " +
      "$" + str(runtime_avg) + " \\pm " + str(runtime_std) + "$ \\\\")
```

```
Registration PMMA block & Real image & 10.69\%    &    98.56\%    &    0.93    &
$3040 \times 2442$    &    12    &    N/A    &    $5611 \pm 75$ \\
```