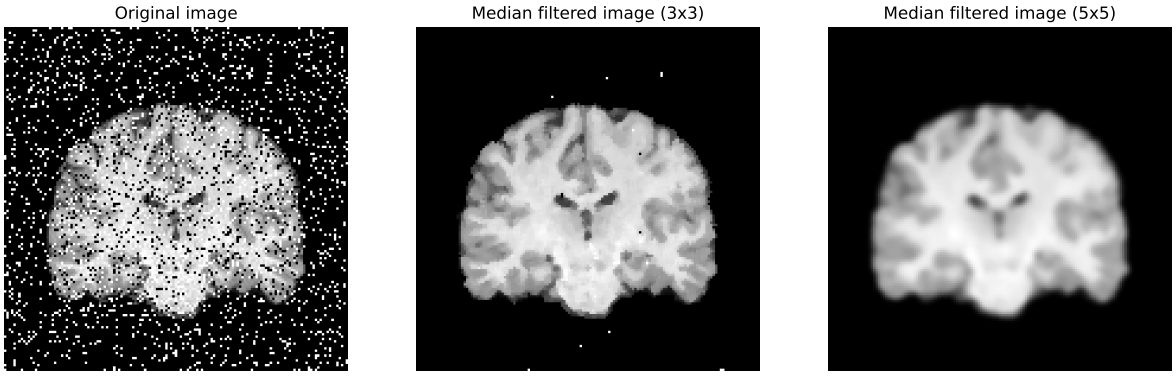
task 3

Task 3

In this task you will need to perform threshold-based image analysis:

Read the greyscale image brain.png, which is provided on the lecture homepage. Reduce the salt and pepper noise in the image using a median filter. (3pts)

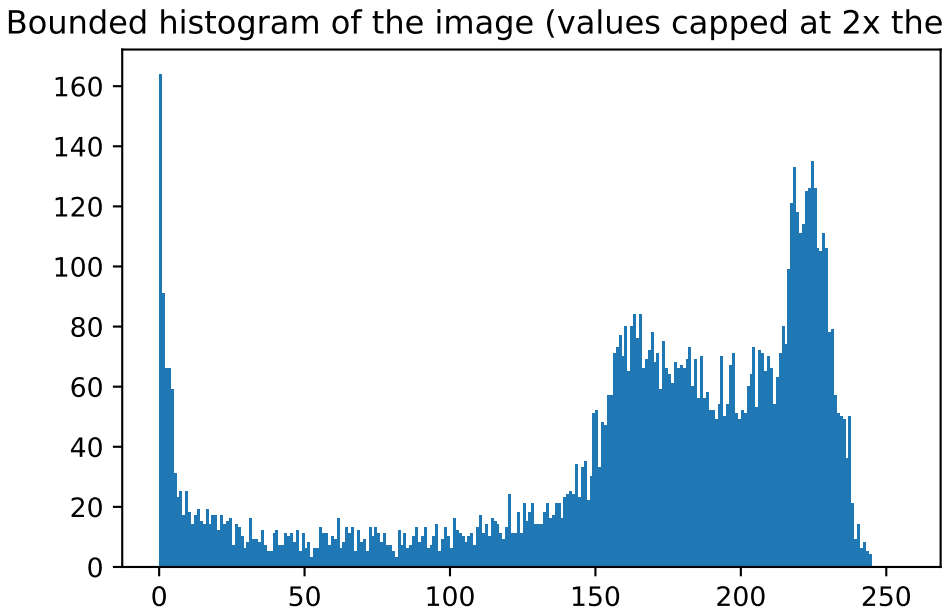
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os


img_noise = cv2.imread("brain-noisy.png", cv2.IMREAD_GRAYSCALE)
if img_noise is None:
    img_noise = cv2.imread("./reports/brain-noisy.png", cv2.IMREAD_GRAYSCALE)
assert img_noise is not None, "Image not found {}".format(os.listdir())
img = cv2.medianBlur(img_noise, 5)
img = cv2.GaussianBlur(img, (5, 5), 0)

fig, ax = plt.subplots(1, 3, figsize=(15, 5))
ax[0].imshow(img_noise, cmap="gray")
ax[0].set_title("Original image")
ax[0].axis("off")
ax[1].imshow(cv2.medianBlur(img_noise, 3), cmap="gray")
ax[1].set_title("Median filtered image (3x3)")
ax[1].axis("off")
ax[2].imshow(img, cmap="gray")
ax[2].set_title("Median filtered image (5x5)")
ax[2].axis("off")

plt.show()
```



As we can see the kernel size of 3×3 is not enough to remove the noise, while the kernel size of 5×5 is sufficient.

Otsu thresholding is a histogram-based method for image segmentation. Use it to find an intensity threshold to segment brain pixels from background. Use Otsu thresholding again to find the threshold only over the brain pixels to segment brain's grey matter from the white matter. Using the two thresholds create three binary masks brain-bg.png, brain-gm.png, brain-wm.png, which should be white in regions of background, grey matter, and white matter, respectively, and black elsewhere. (4pts)

```
values, bin_edge = np.histogram(img, bins=256, range=(0, 256))
bin_centers = (bin_edge[:-1] + bin_edge[1:]) / 2
# values = values[1:]
# bin_centers = bin_centers[1:]
m = values.mean() * 2
values[values > m] = m

plt.bar(bin_centers, values, lw=2)
plt.title("Bounded histogram of the image (values capped at 2x the mean)")
plt.show()
```



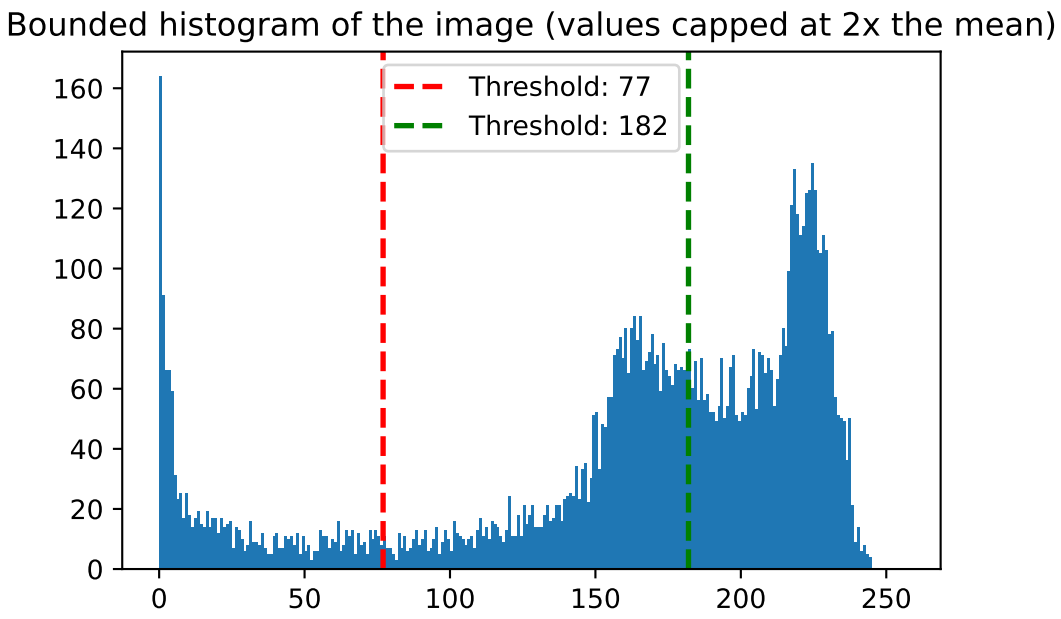
The correct way to use Otsu thresholding with several values is to use [@arora2008multilevel], which is not implemented in OpenCV. However, we can use the implementation in the `skimage` library (which implemented based on [@liao2001fast])

```python
from skimage.filters import threshold_multiotsu


def otsu_threshold(
    img: np.ndarray, classes: int
) -> tuple[list[np.ndarray], np.ndarray]:
    threshold = threshold_multiotsu(img, classes=classes).tolist()
    threshold = [0] + threshold + [255]
    assert (
        len(threshold) == classes + 1
    ), "The number of thresholds should be equal to the number of classes - 1"
    masks = [(img >= t1) & (img < t2) for t1, t2 in zip(threshold, threshold[1:])]
    # masks.append(img >= threshold[-1])
    assert all(mask.dtype == bool for mask in masks), "Masks should be boolean"
    assert (
        len(masks) == classes
    ), "The number of masks should be equal to the number of classes"
    return masks, threshold[1:-1]
```

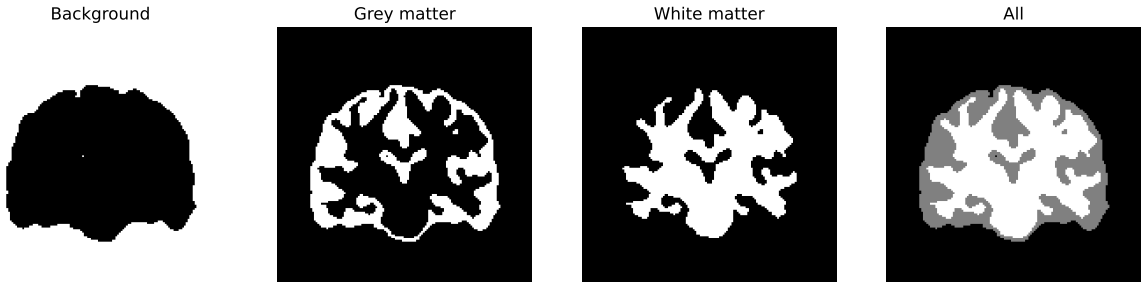
```
(brain_bg, brain_gm, brain_wm), threshold = otsu_threshold(img, 3)
```

```
colors = ["r", "g", "y"]
(brain_bg, brain_gm, brain_wm), threshold = otsu_threshold(img, 3)

print(f"Threshold for the whole image: {threshold}")

values, bin_edge = np.histogram(img, bins=256, range=(0, 256))
bin_centers = (bin_edge[:-1] + bin_edge[1:]) / 2
m = values.mean() * 2
values[values > m] = m

plt.bar(bin_centers, values, lw=2)
for th, color in zip(threshold, colors):
    plt.axvline(th, color=color, lw=2, ls="--", label=f"Threshold: {th}")
plt.legend()
plt.title("Bounded histogram of the image (values capped at 2x the mean)")
plt.show()
```

```
Threshold for the whole image: [77, 182]
```



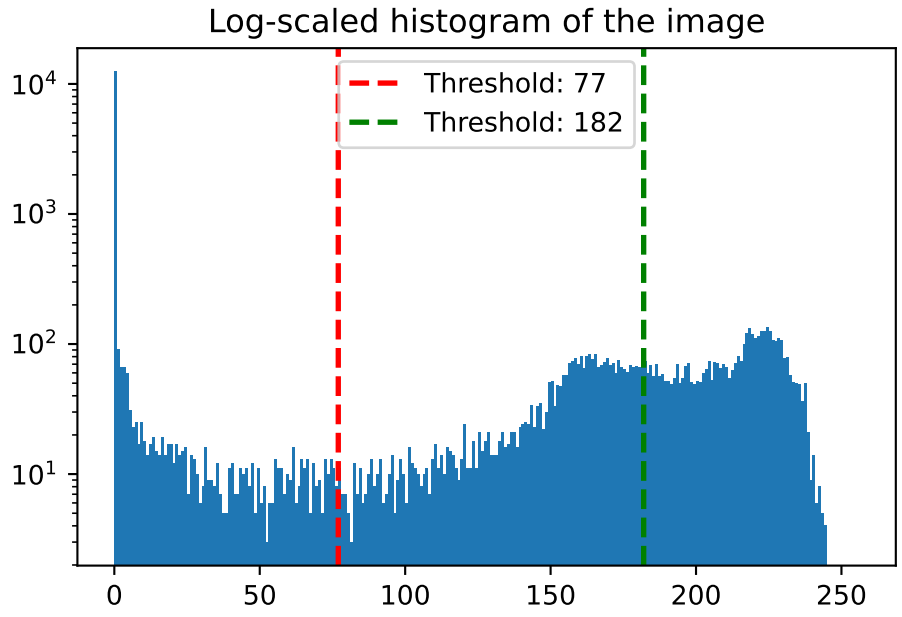
```
fig, ax = plt.subplots(1, 4, figsize=(15, 5))
ax[0].imshow(brain_bg, cmap="gray")
ax[0].set_title("Background")
ax[0].axis("off")

ax[1].imshow(brain_gm, cmap="gray")
ax[1].set_title("Grey matter")
ax[1].axis("off")

ax[2].imshow(brain_wm, cmap="gray")
ax[2].set_title("White matter")
ax[2].axis("off")

ax[3].imshow(brain_bg * 1 + brain_gm * 2 + brain_wm * 3, cmap="gray")
ax[3].set_title("All")
ax[3].axis("off")

plt.show()
```



Plot a log-scaled histogram of the image, which should show how frequently different intensity values occur in the image. How could you roughly estimate the two thresholds you found in the previous task just by looking at the histogram? (3pts)

```
values, bin_edge = np.histogram(img, bins=256, range=(0, 256))
bin_centers = (bin_edge[:-1] + bin_edge[1:]) / 2
plt.bar(bin_centers, values, lw=2)
plt.yscale("log")

for th, color in zip(threshold, colors):
    plt.axvline(th, color=color, lw=2, ls="--", label=f"Threshold: {th}")

plt.legend()
```

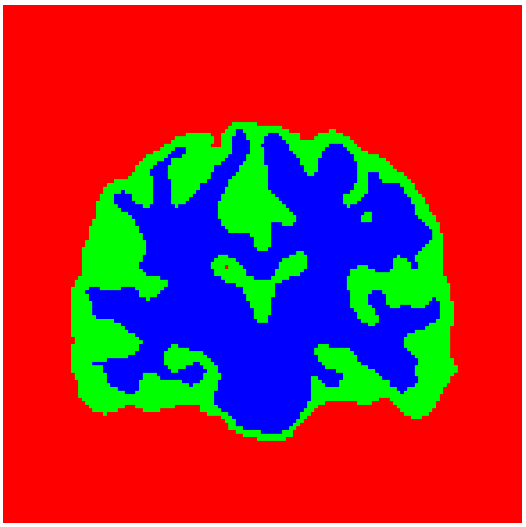
```
plt.title("Log-scaled histogram of the image")
plt.show()
```



As we can see, the histogram has two peaks, which correspond to the grey matter and white matter. The two thresholds can be estimated by finding the two peaks in the histogram. (The purpose of otsu thresholding is to find the optimal threshold for the two peaks)

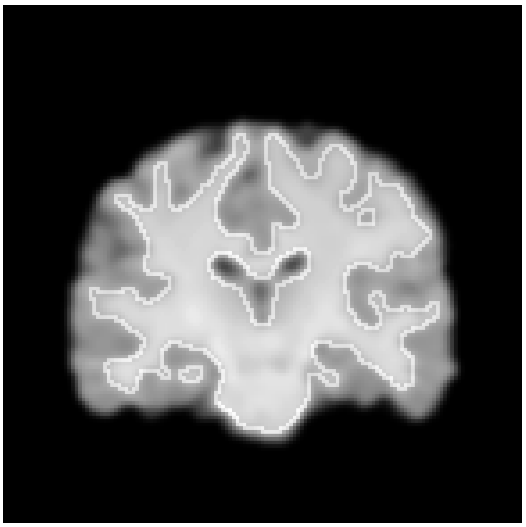
Combine the three masks into a single colour image so that background, grey matter, and white matter are mapped to red, green and blue, respectively. (3pts)

```
combined_brain = np.stack([brain_bg, brain_gm, brain_wm], axis=-1).astype(np.uint8) * 255

plt.imshow(combined_brain)
plt.axis("off")
plt.show()
```



Use erosion (or any other morphological) filter to produce a border between the grey and white matter. Overlay that border on the denoised input image. (3pts)

```
kernel = np.ones((3, 3), np.uint8)
brain_wm_eroded = cv2.erode(brain_wm.astype(np.uint8), kernel, iterations=1)
brain_wm_dilated = cv2.dilate(brain_wm_eroded, kernel, iterations=1)
border = (brain_wm_dilated - brain_wm_eroded) * 255
alpha = 0.85
bordered_img = cv2.addWeighted(img, alpha, border, 1 - alpha, 0)


# plt.imshow(img, cmap="gray")
# plt.imshow(border, cmap="gray", alpha=0.5)
plt.imshow(bordered_img, cmap="gray")
plt.axis("off")
plt.show()
```

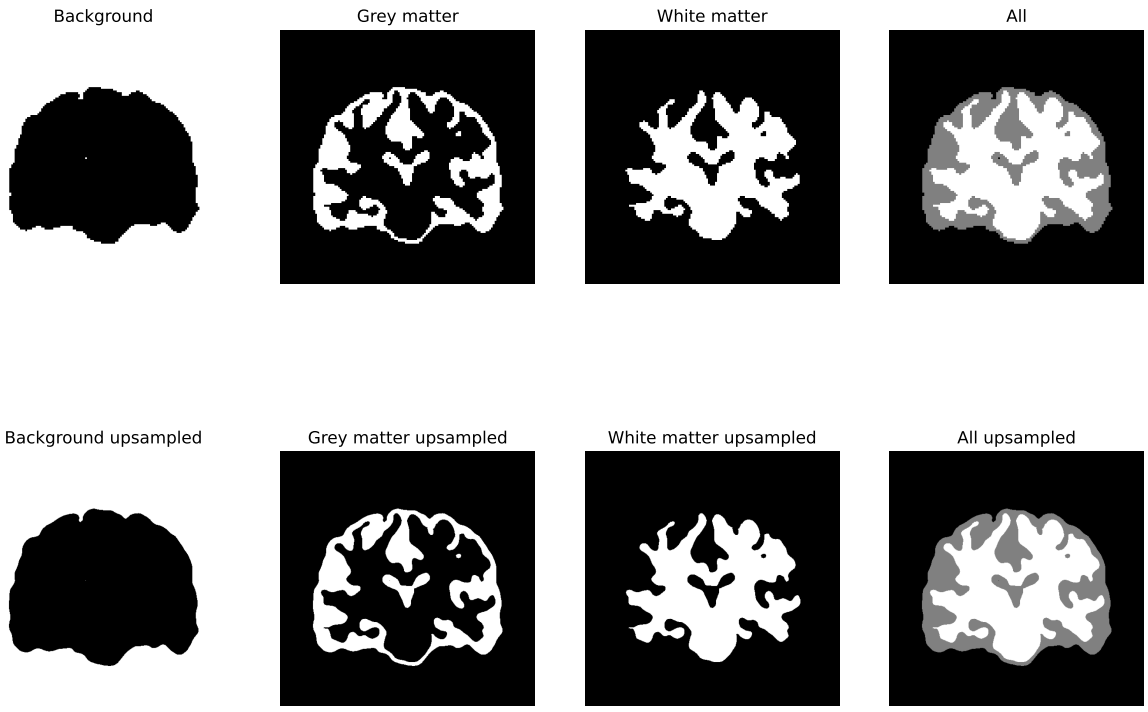


Use bilinear interpolation to up-sample the image by a factor of four along each axis. Apply the same thresholds as in 2) to obtain a segmentation into background, grey matter, and white matter. Up-sample the masks from 2) in the same way and compare the up-sampled masks to the masks from the up-sampled image. Can you see a diference? Why? Repeat the same procedure using nearest neighbour interpolation. Can you see a diference now? (4pts)

```
def upsample(img: np.ndarray, factor: int, interpolation: int) -> np.ndarray:
    return cv2.resize(
        img, (img.shape[1] * factor, img.shape[0] * factor), interpolation=interpolation
    )


masks, threshold = otsu_threshold(img, 3)
img_upsampled = upsample(img, 4, cv2.INTER_LINEAR)
masks_upsampled, threshold_upsampled = otsu_threshold(img_upsampled, 3)

fig, ax = plt.subplots(2, 4, figsize=(15, 10))
fig.suptitle(
    "Comparison of upsampled masks and upsampled image using linear interpolation",
    fontsize=16,
)

titles = ["Background", "Grey matter", "White matter", "All"]
masks.append(masks[0] * 1 + masks[1] * 2 + masks[2] * 3)
masks_upsampled.append(
    masks_upsampled[0] * 1 + masks_upsampled[1] * 2 + masks_upsampled[2] * 3
```

```
)

for i, (mask, mask_upsampled, title) in enumerate(zip(masks, masks_upsampled, titles)):
    ax[0, i].imshow(mask, cmap="gray")
    ax[0, i].set_title(title)
    ax[0, i].axis("off")

    ax[1, i].imshow(mask_upsampled, cmap="gray")
    ax[1, i].set_title(f"{title} upsampled")
    ax[1, i].axis("off")

plt.show()
```

Comparison of upsampled masks and upsampled image using linear interpolation

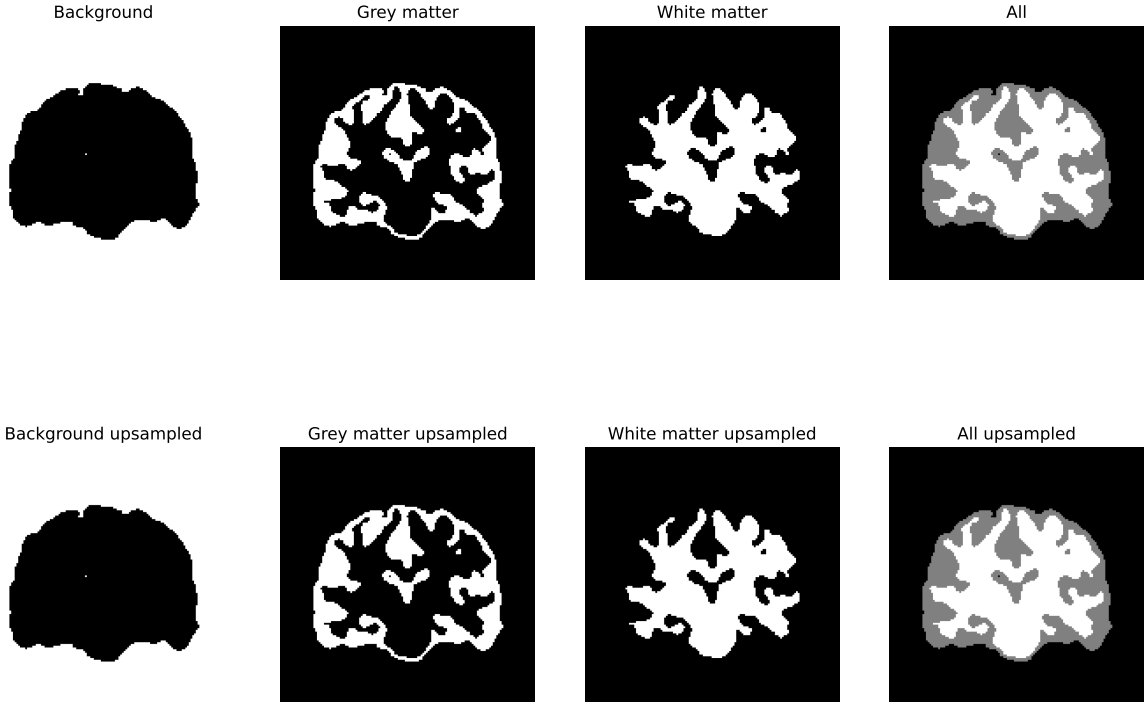


Clearly, we can see much smoother edges in the upsampled masks compared to the upsampled image. This is because the interpolation method used in the up-sampling process is linear, which smooths the edges.

Now, let's repeat the same procedure using the nearest neighbour interpolation method.

```
masks, threshold = otsu_threshold(img, 3)
img_upsampled = upsample(img, 4, cv2.INTER_NEAREST)
masks_upsampled, threshold_upsampled = otsu_threshold(img_upsampled, 3)

fig, ax = plt.subplots(2, 4, figsize=(15, 10))
fig.suptitle(
    "Comparison of upsampled masks and upsampled image using nearest neighbour interpolation"
    fontsize=16,
)

titles = ["Background", "Grey matter", "White matter", "All"]
masks.append(masks[0] * 1 + masks[1] * 2 + masks[2] * 3)
masks_upsampled.append(
    masks_upsampled[0] * 1 + masks_upsampled[1] * 2 + masks_upsampled[2] * 3
)

for i, (mask, mask_upsampled, title) in enumerate(zip(masks, masks_upsampled, titles)):
    ax[0, i].imshow(mask, cmap="gray")
    ax[0, i].set_title(title)
    ax[0, i].axis("off")

    ax[1, i].imshow(mask_upsampled, cmap="gray")
    ax[1, i].set_title(f"{title} upsampled")
    ax[1, i].axis("off")

plt.show()
```

Comparison of upsampled masks and upsampled image using nearest neighbour interpolation



We can see the edges are much sharper in the upsampled masks compared to the upsampled image. This is because the nearest neighbour interpolation method does not smooth the edges.

TODO: Test same thing with pyrUp

```
def upsample_pyramid_(img: np.ndarray) -> np.ndarray:
    # return cv2.resize(
    #     img, (img.shape[1] * factor, img.shape[0] * factor), interpolation=interpolation
    # )
    return cv2.pyrUp(img, dstsize=(img.shape[1] * 2, img.shape[0] * 2))

def upsample_pyramid(img: np.ndarray, factor: int) -> np.ndarray:
    if factor <= 1:
        raise ValueError("Factor should be greater than 1")
    f = 1
    while f < factor:
        img = upsample_pyramid_(img)
        f *= 2
```

```
    return img


print(img.shape)
img_upsampled = upsample_pyramid(img, 4)
print(img_upsampled.shape)
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].imshow(img_upsampled, cmap="gray")
ax[0].set_title("Upsampled image")
ax[0].axis("off")

ax[1].imshow(img, cmap="gray")
ax[1].set_title("Original image")
ax[1].axis("off")

plt.show()
```

```
(145, 145)
(580, 580)
```