

# lab\_01\_segmentation

March 23, 2020

## 1 IAPR 2020: Lab 1 – Image segmentation

**Authors:** Claire Meyer, Nicolas Furrer, Philipp Schuler

**Due date:** 26.03.2020

### 1.1 Extracting relevant data

We first need to extract the lab-01-data.tar.gz archive. To this end, we use the [tarfile](#) module from the Python standard library.

```
[1]: import tarfile
import os

data_base_path = os.path.join(os.pardir, "data")
data_folder = "lab-01-data"
tar_path = os.path.join(data_base_path, data_folder + '.tar.gz')
with tarfile.open(tar_path, mode='r:gz') as tar:
    tar.extractall(path=data_base_path)
```

### 1.2 Part 1: Brain segmentation

Your goal: compute the size of the brain (in pixels) in a 2D image of a human head taken by Magnetic Resonance Imaging (MRI). \* Try as many methods as you can, the more the better. \* At least region growing and contour detection.

#### 1.2.1 1.1 Brain image visualization

```
[2]: import skimage.io
import matplotlib.pyplot as plt
%matplotlib inline

import cv2 as cv
import numpy as np

# Load image with OpenCV
data_path = os.path.join(data_base_path, data_folder)
original_img = cv.imread(os.path.join(data_path, "brain-slice40.tiff"), 0)
```

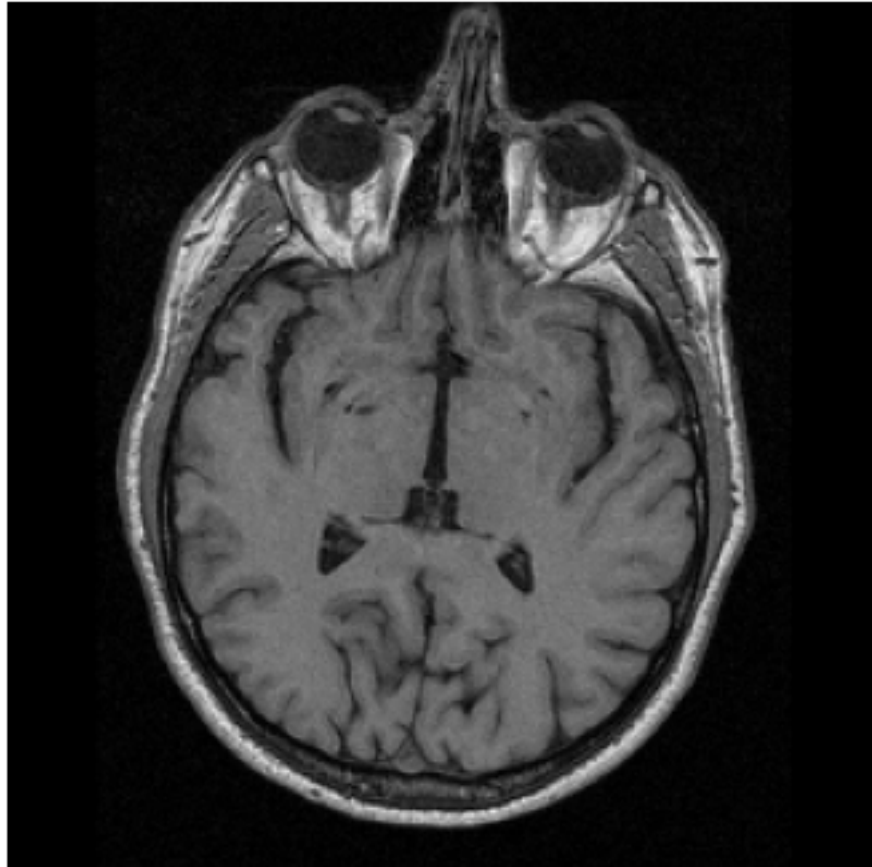
```
height, width = original_img.shape[:2]

# Plot image
n_lin = 1
n_col = 1
size = 6
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size))
ax.imshow(original_img, cmap="gray")
ax.set_title("MRI brain image ({ } px, { } px)".format(height,width))
ax.axis("off")

#End of cell
print("Done")
```

Done

MRI brain image (256 px, 256 px)



### 1.2.2 1.2 Region growing

In order to compute the size of the brain using region growing, we start from the original image,  $I_1$ , and compute the subsequent images : -  $I_2$ , by doing region growing on  $I_1$ , the original image. We obtain the original image superimposed with the mask of the brain surface. The OpenCV function used, `cv.floodFill(image, mask, seedPoint, newVal, loDiff, upDiff)`, also fill the argument mask with the mask of the brain surface, but that mask is bigger than the original image (if original image is  $n \times n$ , then the mask is  $(n+2) \times (n+2)$ , and the padding is filled with the filling value, so here, white (255). -  $I_3$ , by cropping the mask obtained at the previous step. We just need to remove the aforementioned padding. We obtain a mask of the brain surface (brain surface is white on black background).

From there, we just count how many pixels are white in  $I_3$ , the mask, to have the size of the brain in pixels. To verify the quality of the result, we also compute  $I_4$ , by superimposing  $I_3$ , the mask of the brain, on  $I_1$ , the original image. It allows us to assess that the brain surface is indeed correct with visual proof.

All the steps from the original image to the brain mask and the quality control image (from  $I_1$  to  $I_4$ ) are displayed below, along with the implementation :

```
[3]: # Region growing from pixel (150,150)
grown_img = original_img.copy()
seed = (150,150)
t1 = 6
t2 = 15
mask = np.zeros((height+2,width+2), np.uint8)
cv.floodFill(image=grown_img, mask=mask, seedPoint=seed, newVal=255, loDiff=t1,
    ↳upDiff=t2)

# Getting the mask back
brain_mask = mask[1:width+1,1:height+1].copy()*255
cv.imwrite(filename="test.png", img=brain_mask)

# Superimposition
empty_mask = np.zeros((height, width), np.uint8)
color_brain_mask = cv.merge(mv=(empty_mask, empty_mask, brain_mask))
color_original_img = cv.cvtColor(original_img, code=cv.COLOR_GRAY2RGB)
superimposed_img = cv.add(src1=color_brain_mask, src2=color_original_img)

# Number of pixels
n = np.sum(brain_mask == 255)

# Hyperparameters :
print(f"Hyperparamters : ")
print(f"    Seed at {seed}.")
print(f"    Max lower difference : {t1}.")
print(f"    Max upper difference : {t2}.")

# Results
```

```

print(f"Results : ")
print(f"    Size of the brain : {n} px in the {height}x{width} image.")
print(f"    Percentage of area occupied by the brain in the image : {n*100/
    →(width*height):.2f}%.")

# Plots
n_lin = 2
n_col = 2
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size))

ax[0,0].imshow(original_img, cmap = 'gray')
ax[0,0].set_title("$I_1$ : Original image")

ax[0,1].imshow(grown_img, cmap = 'gray')
ax[0,1].set_title("$I_2$ : Region growing result")

ax[1,0].imshow(brain_mask, cmap = 'gray')
ax[1,0].set_title("$I_3$ : Mask of brain surface")

ax[1,1].imshow(superimposed_img, cmap = 'gray')
ax[1,1].set_title("$I_4$ : Control image (mask on original image)")

for a in ax.flatten():
    a.axis("off")

#End of cell
print("Done")

```

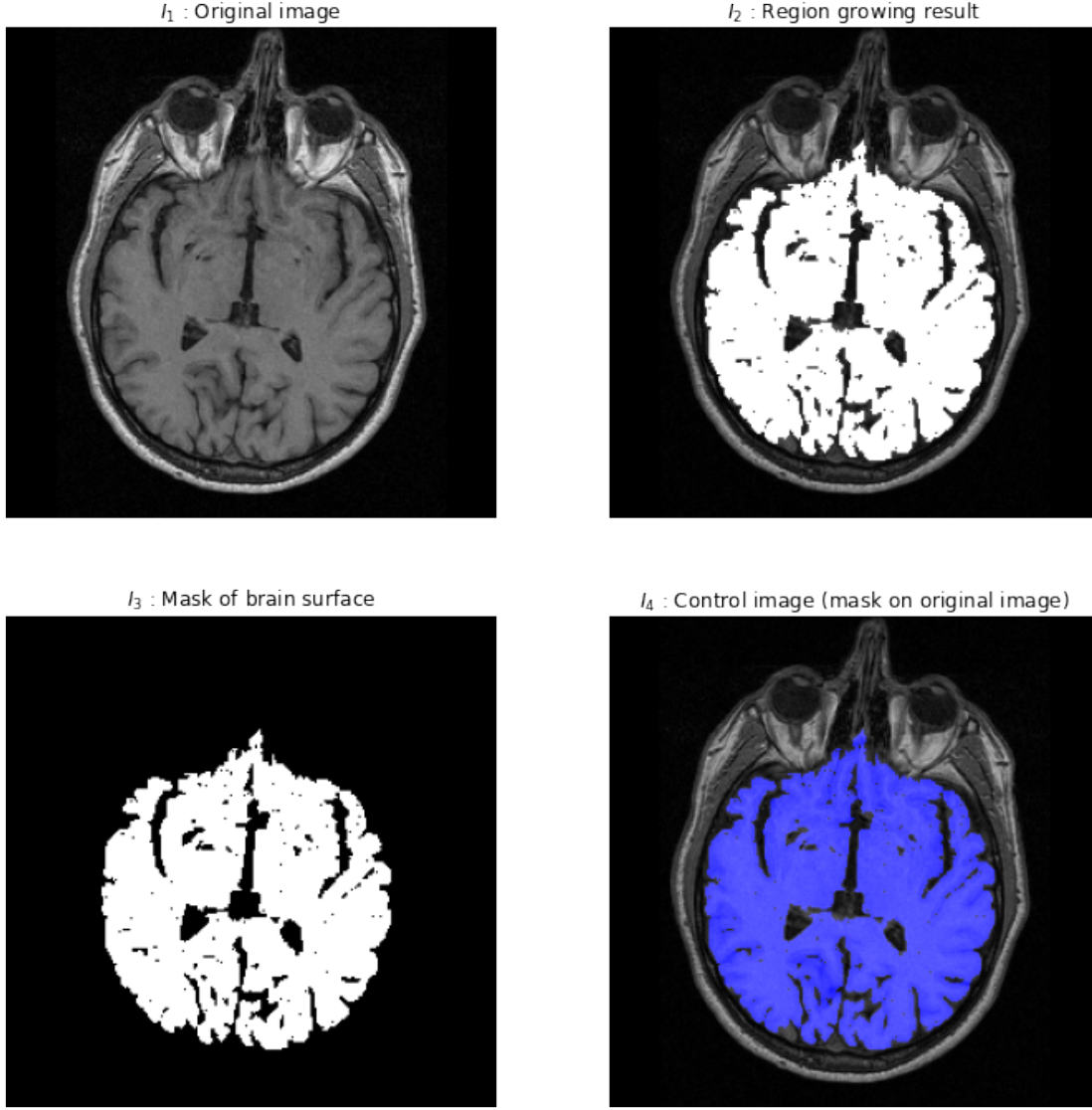
Hyperparamters :

- Seed at (150, 150).
- Max lower difference : 6.
- Max upper difference : 15.

Results :

- Size of the brain : 15667 px in the 256x256 image.
- Percentage of area occupied by the brain in the image : 23.91%.

Done



The result indicates that the surface of the brain is of 15667 px, which corresponds to 23.91% of the image size, as can be seen just above.  $I_4$ , the quality control image, also shows that the segmentation is good, and thus that the result can be trusted.

This method requires three meta parameters : - An initial seed for the region growing, to begin the flood-filling from. In order to find the surface of the brain, that seed needs to be located somewhere within that area. Thus, it can take many values, as long as the coordinates are within the surface of the brain. Here, they were chosen by estimation with the naked eye (trial and error), and  $(x,y)=(150,150)$  is such a coordinate. - A lower and upper brightness difference between the currently observed pixel (pixel at  $(x,y)$ ) and one of its neighbors belonging to the component, or a seed pixel being added to the component (pixel at  $(x',y')$ ). Here, we determine that the next pixel belong to the region if :

$$\text{src}(x',y') - \text{loDiff} \leq \text{src}(x,y) \leq \text{src}(x',y') + \text{upDiff}$$

Here, the lower difference is 6, and the upper difference is 15. It means that a pixel is much more easily added to the growing region if the brightness is higher, than if the brightness is lower, as the empty zones between the brain and the skull are darker.

### 1.2.3 1.3 Contour detection

In order to compute the size of the brain using contour detection, we start from the original image,  $I_1$ , and compute the subsequent images : -  $I_2$ , by applying the Canny algorithm on  $I_1$ , the original image. We obtain the contours of the original image. But those contours are not closed, they are just edges from the original image. To determine a region (here, the brain), we need the contour around that region closed (the other contours don't matter). -  $I_3$ , by dilating  $I_2$ , the contours. We obtained closed contours, because the dilation of the contours make them grow. Thus, the area containing the brain is determined by a closed contour. -  $I_4$ , by flooding  $I_3$ , the closed contours, with a initial seed in the brain. We obtain a compound black image, with the closed dilated contours and the brain surface in white. Indeed, flooding fills a zone on the basis of a continuous color (or here, lack thereof, because we flood an area of pixel value 0). Because we fill from the inside of the brain (black), and the brain is closed by closed contours (white), after flood-filling, the brain is white. It is the same function from OpenCV than the one used in the region growing part; but the difference is that here, we do not set the lower and upper difference. Thus they are the default, which is 0. It means that it is no longer a region growing, but a flooding (only exactly the same color is filled). -  $I_5$ , by performing the "NOT" bitwise operation on  $I_4$ . We obtain a compound white image, with the closed dilated contours and the brain surface in black. The operations to obtain  $I_5$  and  $I_6$  are just logical operations on the images, to obtain what we want. -  $I_6$ , by performing the "OR" bitwise operation on  $I_3$  and  $I_5$ . Since  $I_3$  contains the closed dilated contours, and  $I_5$  anything but the contours and the brain surface, we obtain a white image with the brain surface in black. That brain surface is eroded compared to the real size, because of the previous dilation (of the edges of the brain). -  $I_7$ , by eroding  $I_6$ . We obtain a white image with the brain in black, but this time the correct surface. We need this erosion because we had to dilate  $I_2$ , to close the contours. This dilation "ate away" at the brain, so now we need to "grow it back up". This is done by erosion because the brain is in black, not in white. -  $I_8$ , by performing the "NOT" bitwise operation on  $I_7$ . We obtain a mask of the brain surface (brain surface is white on black background).

From there, we just count how many pixels are white in  $I_8$ , the mask, to have the size of the brain in pixels. To verify the quality of the result, we also compute  $I_9$ , by superimposing  $I_8$ , the mask of the brain, on  $I_1$ , the original image. It allows us to assess that the brain surface is indeed correct with visual proof.

All the steps from the original image to the brain mask and the quality control image (from  $I_1$  to  $I_9$ ) are displayed below, along with the implementation :

```
[4]: # Canny detection
t1 = 85
t2 = 200
edges_img = cv.Canny(image=original_img, threshold1=t1, threshold2=t2,
    ↪apertureSize=3)

# Dilation of contours
kernel = np.zeros((3,3), np.uint8)
```

```

cv.circle(img=kernel, center=(1,1), radius=1, color=255, thickness=-1)
dilated_edges_img = cv.dilate(edges_img, kernel, iterations=1)

# Flood-filling from pixel (150, 150)
mask = np.zeros((height+2, width+2), np.uint8)
flooded_img = dilated_edges_img.copy()
seed = (150,150)
cv.floodFill(flooded_img, mask, seed, 255);

# Inversion of the flood filled image
inverted_flooded_img = cv.bitwise_not(flooded_img)

# Combination of the thresholded image with the inverted flood filled image
→using bitwise OR operation
dilated_segmented_img = cv.bitwise_or(dilated_edges_img, inverted_flooded_img)
segmented_img = cv.erode(dilated_segmented_img, kernel, iterations=1)

# Brain mask
brain_mask = cv.bitwise_not(segmented_img)

# Superimposition
empty_mask = np.zeros((height, width), np.uint8)
color_brain_mask = cv.merge(mv=(empty_mask, empty_mask, brain_mask))
color_original_img = cv.cvtColor(original_img, code=cv.COLOR_GRAY2RGB)
superimposed_img = cv.add(src1=color_brain_mask, src2=color_original_img)

# Number of pixels
n = np.sum(brain_mask == 255)

# Hyperparameters :
print(f"Hyperparamters : ")
print(f"    Low Canny threshold : {t1}.")
print(f"    High Canny threshold : {t2}.")
print(f"    Seed at {seed}.")

# Results
print(f"Results : ")
print(f"    Size of the brain : {n} px in the {height}x{width} image.")
print(f"    Percentage of area occupied by the brain in the image : {n*100/
→(width*height):.2f}%.")

# Plots
n_lin = 3
n_col = 3
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size))

ax[0,0].imshow(original_img, cmap = 'gray')

```

```

ax[0,0].set_title("$I_1$ : Original image")

ax[0,1].imshow(edges_img, cmap = 'gray')
ax[0,1].set_title("$I_2$ : Contours of original image")

ax[0,2].imshow(dilated_edges_img, cmap = 'gray')
ax[0,2].set_title("$I_3$ : Closed, dilated contours")

ax[1,0].imshow(flooded_img, cmap = 'gray')
ax[1,0].set_title("$I_4$ : Closed, dilated contours and brain surface in white")

ax[1,1].imshow(inverted_flooded_img, cmap = 'gray')
ax[1,1].set_title("$I_5$ : Closed, dilated contours and brain surface in black")

ax[1,2].imshow(dilated_segmented_img, cmap = 'gray')
ax[1,2].set_title("$I_6$ : Mask of everything but brain \n(the brain is a little_
    ↳bit eroded away\nbecause of the dilation at the third step)")

ax[2,0].imshow(segmented_img, cmap = 'gray')
ax[2,0].set_title("$I_7$ : Mask of brain surface (inverted)")

ax[2,1].imshow(brain_mask, cmap = 'gray')
ax[2,1].set_title("$I_8$ : Mask of brain surface")

ax[2,2].imshow(superimposed_img, cmap = 'gray')
ax[2,2].set_title("$I_9$ : Control image (mask on original image)")

for a in ax.flatten():
    a.axis("off")

#End of cell
print("Done")

```

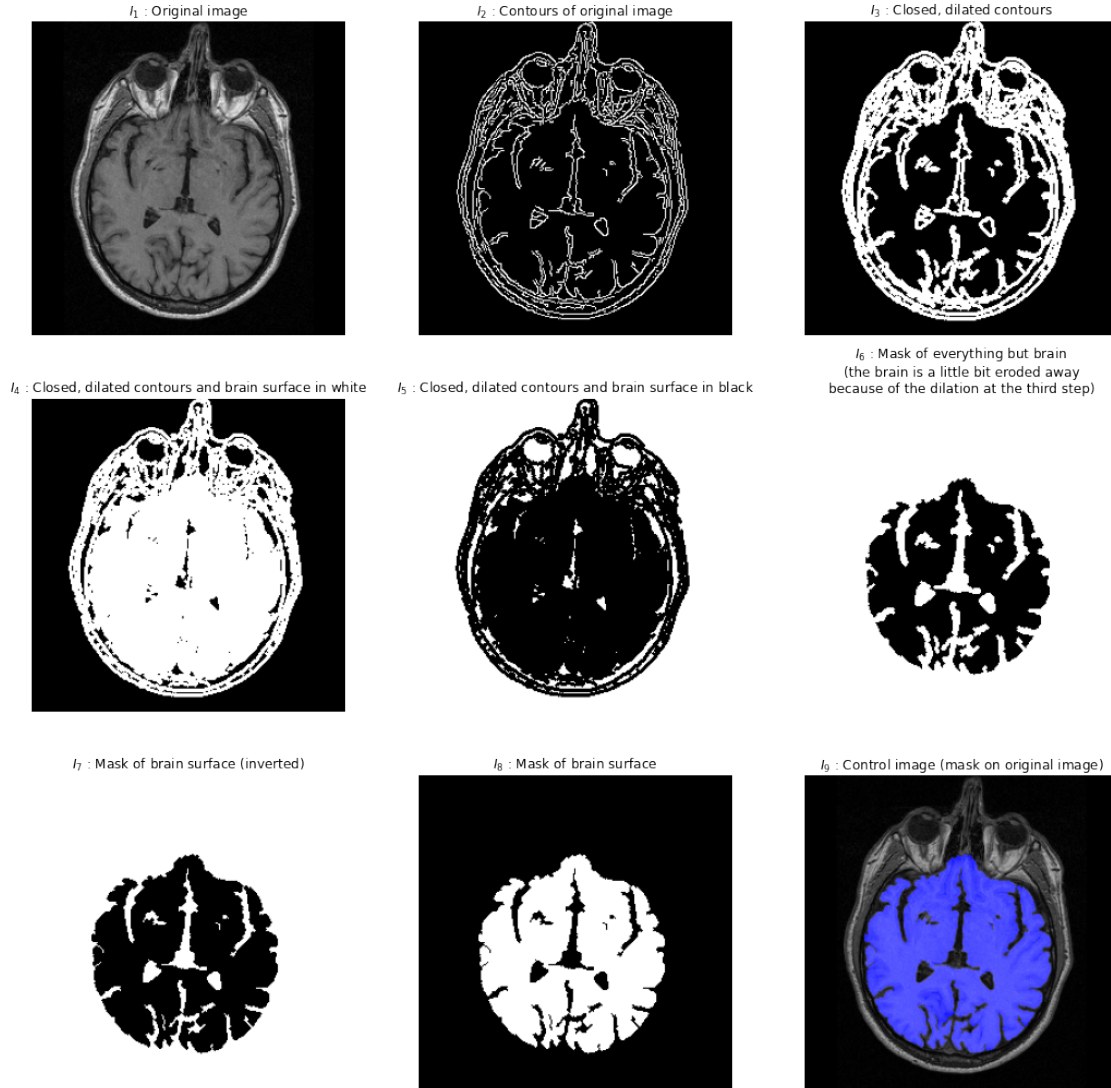
Hyperparamters :

- Low Canny threshold : 85.
- High Canny threshold : 200.
- Seed at (150, 150).

Results :

- Size of the brain : 15742 px in the 256x256 image.
- Percentage of area occupied by the brain in the image : 24.02%.

Done



The result indicates that the surface of the brain is of 15742 px, which corresponds to 24.02% of the image size, as can be seen just above.  $I_9$ , the quality control image, also shows that the segmentation is good, and thus that the result can be trusted.

This method requires a few meta parameters : -  $t_1$  and  $t_2$ , the thresholds for the Canny edges detector. Those two threshold mark the strong and weak edges : anything above the highest threshold is a strong edge, anything below the lowest threshold is deleted, and the rest is a weak edge, whose validity is determined by its link, or lack thereof, to a strong edge. Those values are determined empirically, and influence drastically the result of the edge detection. Here, we chose  $t_1=85$  and  $t_2=200$ . - The dilation kernel is also quite important. If too small, the dilation doesn't close the edges. If too big, some edges that should remain separate become fused. Here, the kernel is a circle of radius 1, which is equivalent to a cross on a 3x3 matrix. To ensure some degree of consistency, the same kernel is used for the subsequent erosion. - Similarly, the dilation iteration number, if too high, can fuse edges that should remain separate. Here, we only iterated the dilation

once; and thus we only eroded once as well, for consistency. - The flooding necessitates an initial seed, to begin the the flooding from. In order to find the surface of the brain, that seed needs to be located somewhere within that area. Thus, it can take many values, as long as the coordinates are within the surface of the brain. Here, they were chosen by estimation with the naked eye (trial and error), and  $(x,y)=(150,150)$  is such a coordinate.

Initially, we tried an histogram equalization on the original image, to see if we could determine the edges more accurately that way. It proved to be severely unhelpful, to the point of being deleterious to the result. When looking at the result of an histogram equalization, we can easily see why. Indeed a lot of noise is enhanced by the equalization, to the point where there are way too many false positive when trying to detect the edges with the Canny algorithm. Below are shown the original image, the histogram equalization of that original image, and the edges detected with the Canny algorithm from that equalized image :

```
[5]: # Histogram equalization
equalized_img = cv.equalizeHist(original_img)

# Canny detection
t1 = 240
t2 = 245
edges_img_2 = cv.
    ↳Canny(image=equalized_img, threshold1=t1, threshold2=t2, apertureSize=3)

# Plots
n_lin = 1
n_col = 3
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size))

ax[0].imshow(original_img, cmap = 'gray')
ax[0].set_title("Original image")

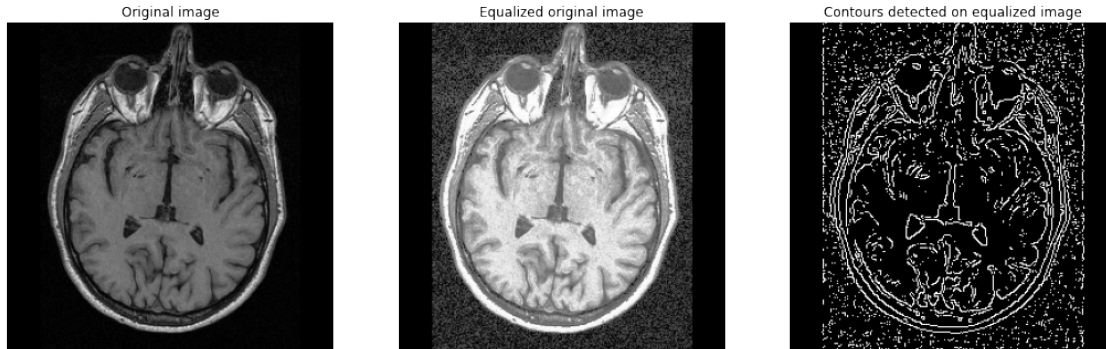
ax[1].imshow(equalized_img, cmap = 'gray')
ax[1].set_title("Equalized original image")

ax[2].imshow(edges_img_2, cmap = 'gray')
ax[2].set_title("Contours detected on equalized image")

for a in ax.flatten():
    a.axis("off")

#End of cell
print("Done")
```

Done



In order to try and detect only the strongest edges, the Canny thresholds are set very high, to keep only the strongest of edges, but there are still too many false positive : the histogram equalization broke the edges on the original image.

#### 1.2.4 1.4 Active contours

In order to compute the size of the brain using active contours, we initialize an initial snake around the brain, and then warp it to closely stick to the brain surface edge. From it, we compute the area within the snake using the Green theorem.

All the steps are displayed below, along with the implementation :

```
[6]: from skimage.filters import gaussian
from skimage.segmentation import active_contour

# Area calculation with green theorem :
# Source: https://stackoverflow.com/questions/22678990/how-can-i-calculate-the-area-within-a-contour-in-python-using-the-matplotlib
def area(vs):
    area = 0
    x0,y0 = vs[0]
    for [x1,y1] in vs[1:]:
        dx = x1-x0
        dy = y1-y0
        area += 0.5*(y0*dx - x0*dy)
        x0 = x1
        y0 = y1
    return area

# Initialisation of the snake as a circle centered around the brain
center = (147, 125)
radius = 75
samples = np.linspace(0, 2*np.pi, 400)
sine = center[0] + radius*np.sin(samples)
cosine = center[1] + radius*np.cos(samples)
```

```

initial_snake = np.array([sine, cosine]).T

# Gaussian blur
blurred_img = gaussian(original_img, 1)

# Active contour detection
beta_val = 4
line_val = -5
edge_val = 10
final_snake_original = active_contour(original_img, initial_snake,
    ↪beta=beta_val, w_line=line_val, w_edge=edge_val, coordinates='rc')

# Number of pixels
n_original = int(area(final_snake_original))

# Hyperparameters :
print(f"Hyperparamters : ")
print(f"    Initial snake center : {center}.")
print(f"    Initial snake radius : {radius}.")
print(f"    Active contour beta value : {beta_val}.")
print(f"    Active contour w line value : {line_val}.")
print(f"    Active contour w edge value : {edge_val}.")

# Results
print(f"Results : ")
print(f"    Size of the brain : {n_original} px in the {height}x{width} image.")
print(f"    Percentage of area occupied by the brain in the image : ↪
    ↪{n_original*100/(width*height):.2f}%.")

# Plots
n_lin = 1
n_col = 2
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size))

ax[0].imshow(original_img, cmap = 'gray')
ax[0].plot(initial_snake[:, 1], initial_snake[:, 0], '--r', lw=3)
ax[0].set_title("$I_1$ : Initial snake")

ax[1].imshow(original_img, cmap = 'gray')
ax[1].plot(final_snake_original[:, 1], final_snake_original[:, 0], '-b', lw=3)
ax[1].set_title("$I_2$ : Final snake")

for a in ax.flatten():
    a.axis("off")

#End of cell
print("Done")

```

Hyperparameters :

Initial snake center : (147, 125).

Initial snake radius : 75.

Active contour beta value : 4.

Active contour w line value : -5.

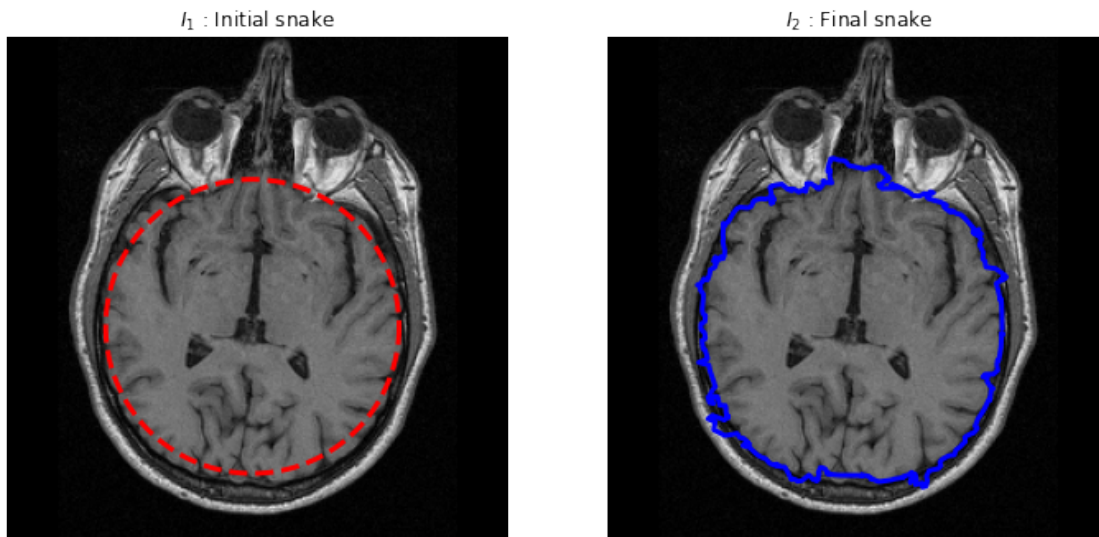
Active contour w edge value : 10.

Results :

Size of the brain : 19259 px in the 256x256 image.

Percentage of area occupied by the brain in the image : 29.39%.

Done



The result indicates that the surface of the brain is of 19259 px, which corresponds to 29.39% of the image size, as can be seen just above. The active contour may not be the best choice in this situation, as we are trying to get the grey zone out of the other different grey zone; on  $I_2$  we can see that the snake doesn't stick as closely to the edge of the brain surface as the previous methods. We also have a lot of holes that we want to detect. That could be done with multiple active contour detections just for the holes.

This method requires a few meta parameters : - The initial snake position; the closer it is to the final snake, the better. - Beta, the smoothness of the line. We don't want to be too smooth as the brain has lots of folds. -  $w_{line}$ , the attraction to brightness. Here we want to keep away from the brighter areas and try to stay in the dark zones between the brain and the skull. -  $w_{edge}$ , the attraction to the edges. We want to stick as much as possible to edges.

Those meta parameters allow to miss very few parts of the brain, but include some that shouldn't be.

### 1.2.5 1.4 Thresholding

In order to compute the size of the brain using contour detection, we start from the original image,  $I_1$ , and compute the subsequent images : -  $I_2$ , by equalizing  $I_1$ 's histogram, the original image. We

obtain an image with values more spread on the dynamic available. -  $I_3$ , by thresholding  $I_2$ , the histogram equalized version of the original image. Ideally, we want at that stage to have the brain as a single region, separate from the other regions on the image. Unfortunately, that's not the case, and on the images below, it is visible that the right ocular region is attached to the brain after thresholding. It doesn't matter, because we can get rid of it at the next step. -  $I_4$ , by eroding  $I_3$ , the thresholded image, with a  $3 \times 3$  kernel presenting a circle of radius 1 (i.e., a cross). We iterate the erosion until the ocular region is cut from the brain, that is three times. We obtain an eroded thresholded image, where the brain is one region, separate from the rest. -  $I_5$ , by flooding  $I_4$ , where the brain was isolated, with a initial seed withing the brain area. The two next steps are bitwise operations similar to the method with contour detection. -  $I_6$ , by performing the "NOT" bitwise operation on  $I_5$ . -  $I_7$ , by performing the "AND" bitwise operation on  $I_6$  and  $I_4$ . We obtain an eroded version of the brain mask. To restore the correct brain mask,  $I_8$ , we just need one last step. -  $I_8$ , by dilating  $I_7$ , the eroded brain mask. We obtain a mask of the brain surface (brain surface is white on black background). We use the same kernel as for the erosion, and the same number of iterations.

From there, we just count how many pixels are white in  $I_8$ , the mask, to have the size of the brain in pixels. To verify the quality of the result, we also compute  $I_9$ , by superimposing  $I_8$ , the mask of the brain, on  $I_1$ , the original image. It allows us to assess that the brain surface is indeed correct with visual proof.

All the steps from the original image to the brain mask and the quality control image (from  $I_1$  to  $I_9$ ) are displayed below, along with the implementation :

```
[7]: # Histogram equalization
equalized_img = cv.equalizeHist(original_img)

# Plain threshold
threshold = 127
_, thresholded_img = cv.threshold(src=equalized_img, thresh=threshold,
    ↳maxval=255, type=cv.THRESH_BINARY)

cv.imwrite("test.png", thresholded_img)

# Erosion to cut the right eye
iterations_num = 3
kernel = np.zeros((3,3), np.uint8)
cv.circle(img=kernel, center=(1,1), radius=1, color=255, thickness=-1)
eroded_thresholded_img = cv.erode(thresholded_img, kernel,
    ↳iterations=iterations_num)

# Flood-filling from pixel (150, 150)
mask = np.zeros((height+2, width+2), np.uint8)
flooded_img = eroded_thresholded_img.copy()
seed = (150,150)
cv.floodFill(flooded_img, mask, seed, 0);

# Inversion of the flood filled image
```

```

eroded_inverted_flooded_img = cv.bitwise_not(flooded_img)

# Combination of the thresholded image with the inverted flood filled image
→using bitwise AND operation
eroded_brain_mask = cv.bitwise_and(eroded_inverted_flooded_img,
→eroded_thresholded_img)

# Dilation to grow back the missing bits
brain_mask = cv.dilate(eroded_brain_mask, kernel, iterations=iterations_num)

# Superimposition
empty_mask = np.zeros((height, width), np.uint8)
color_brain_mask = cv.merge(mv=(empty_mask, empty_mask, brain_mask))
color_original_img = cv.cvtColor(original_img, code=cv.COLOR_GRAY2RGB)
superimposed_img = cv.add(src1=color_brain_mask, src2=color_original_img)

# Number of pixels
n = np.sum(brain_mask == 255)

# Hyperparameters :
print(f"Hyperparamters : ")
print(f"    Threshold value : {threshold}.")
print(f"    Number of erosions / dilations : {iterations_num}.")
print(f"    Seed at {seed}.")

# Results
print(f"Results : ")
print(f"    Size of the brain : {n} px in the {height}x{width} image.")
print(f"    Percentage of area occupied by the brain in the image : {n*100/
→(width*height):.2f}%.")

# Plots
n_lin = 3
n_col = 3
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size))

ax[0,0].imshow(original_img, cmap = 'gray')
ax[0,0].set_title("$I_1$ : Original image")

ax[0,1].imshow(equalized_img, cmap = 'gray')
ax[0,1].set_title("$I_2$ : Histogram equalized image")

ax[0,2].imshow(thresholded_img, cmap = 'gray')
ax[0,2].set_title("$I_3$ : Thresholded image")

ax[1,0].imshow(eroded_thresholded_img, cmap = 'gray')
ax[1,0].set_title("$I_4$ : Eroded thresholded image")

```

```

ax[1,1].imshow(flooded_img, cmap = 'gray')
ax[1,1].set_title("$I_5$ : Flooded eroded thresholded \nimage with 0 in brain")

ax[1,2].imshow(eroded_inverted_flooded_img, cmap = 'gray')
ax[1,2].set_title("$I_6$ : Inverted eroded flooded brain")

ax[2,0].imshow(eroded_brain_mask, cmap = 'gray')
ax[2,0].set_title("$I_7$ : Eroded mask of brain surface ($I_4$ & $I_6$)")

ax[2,1].imshow(brain_mask, cmap = 'gray')
ax[2,1].set_title("$I_8$ : Mask of brain surface")

ax[2,2].imshow(superimposed_img, cmap = 'gray')
ax[2,2].set_title("$I_9$ : Control image (mask on original image)")

for a in ax.flatten():
    a.axis("off")

#End of cell
print("Done")

```

Hyperparamters :

Threshold value : 127.

Number of erosions / dilations : 3.

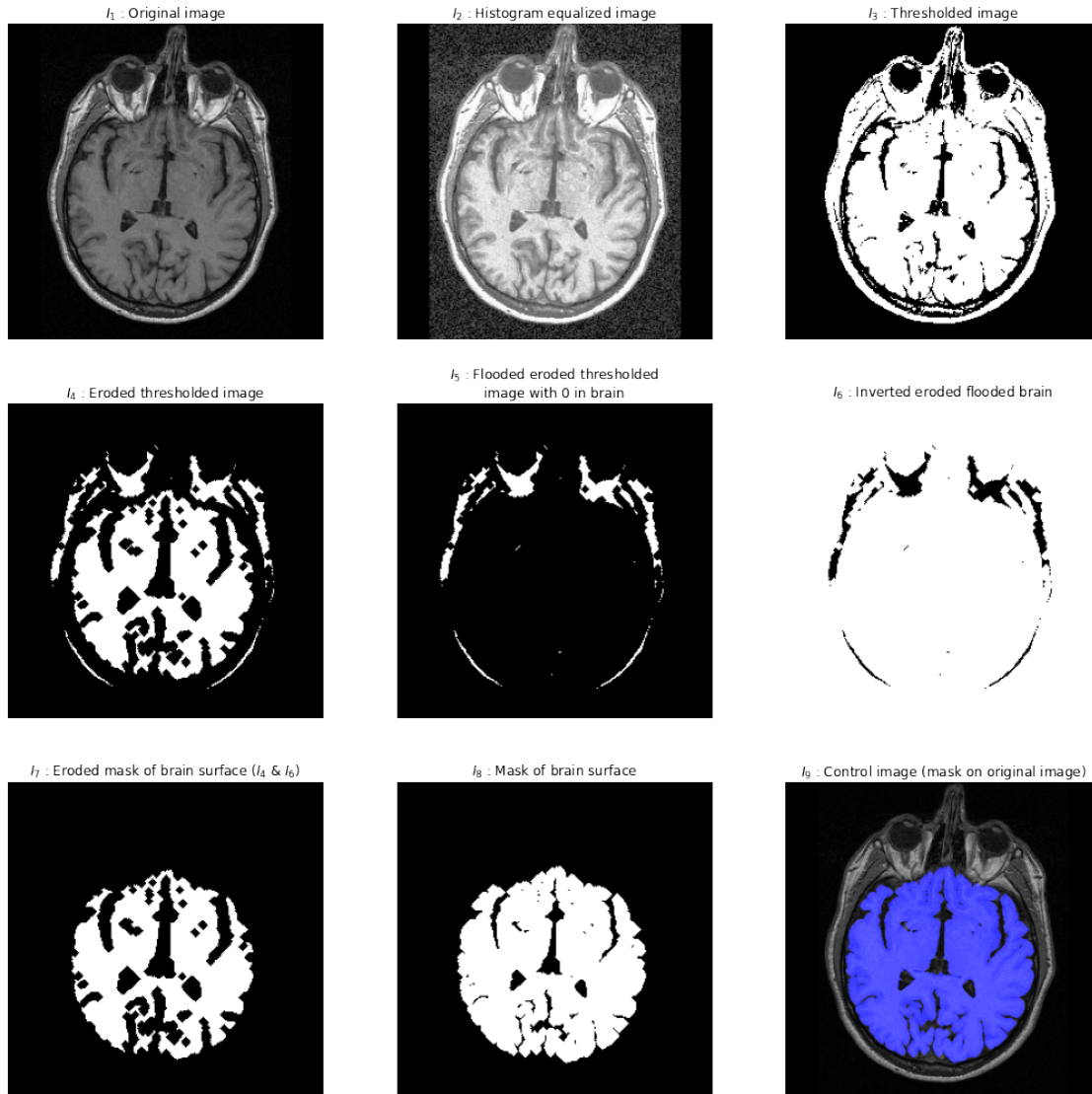
Seed at (150, 150).

Results :

Size of the brain : 16570 px in the 256x256 image.

Percentage of area occupied by the brain in the image : 25.28%.

Done



The result indicates that the surface of the brain is of 16570 px, which corresponds to 25.28% of the image size, as can be seen just above.  $I_9$ , the quality control image, also shows that the segmentation is good, and thus that the result can be trusted.

This method requires a few meta parameters : - The first is the threshold value for the thresholding. Too high, and not all the brain is one piece during this crucial part of the segmentation. Too low, and bit of other cranial structures are attached to the brain. Unfortunately, the “too high” value is lower than the “too low” value, so we have to choose between the brain in its entirety but with bits of ocular region attached, and the brain in several parts. Since it’s easier to get rid of the ocular parts later, we chose a value at which the brain is in one piece. Initially, we tried the Otsu threshold, which maximize inter-class variance, and minimizes intra-class variance; but it’s just not a good threshold for this application. - The erosion kernel is also quite important. If it’s too big, then the risk is to detach parts of the brain. That’s why it’s a cross on a 3 x 3 matrix. The same kernel is later used for the dilation to restore back the brain region. - The number of

iterations to the dilation is dictated by need : it's the minimum number of iterations needed to detach the right ocular region from the brain. The brain region, when segmented, is dilated the same number of time, to restore back the correct zone. - The flood-fill necessitates an initial seed, to begin the the flood-filling from. In order to find the surface of the brain, that seed needs to be located somewhere within that area. Thus, it can take many values, as long as the coordinates are within the surface of the brain. Here, they were chosen by estimation with the naked eye (trial and error), and  $(x,y)=(150,150)$  is such a coordinate.

Initially, the histogram equalization showed that 0-padding wide of 26 px was added to the original image. Because this adds artificial pixels of value 0 to the image, we wondered how it affected the histogram equalization. Thus, we equalized on the whole original image, and on a ROI of the original image, to see the difference :

```
[8]: # Histogram equalization on the whole image
equalized_img = cv.equalizeHist(original_img)

# Histogram equalization with ROI
d = 26
equalized_roi = original_img.copy()
equalized_roi[:,d:width-d] = cv.equalizeHist(original_img[:,d:width-d])

# Difference
print(f"Results : ")
print(f"    Norm of error between with ROI and without : {cv.norm(equalized_img,
    →equalized_roi)}")

# Histograms
hist_original, bins_original = np.histogram(a = original_img , bins = 256,
    →range=[0,256])
hist_equa_img, bins_equa_img = np.histogram(a = equalized_img, bins = 256,
    →range=[0,256])
hist_equa_roi, bins_equa_roi = np.histogram(a = equalized_roi, bins = 256,
    →range=[0,256])

# Normalized cumulative distribution functions
cdf_original = hist_original.cumsum()
cdf_equa_img = hist_equa_img.cumsum()
cdf_equa_roi = hist_equa_roi.cumsum()

cdf_original = cdf_original * hist_original.max() / cdf_original.max()
cdf_equa_img = cdf_equa_img * hist_equa_img.max() / cdf_equa_img.max()
cdf_equa_roi = cdf_equa_roi * hist_equa_roi.max() / cdf_equa_roi.max()

# Plots
n_lin = 2
n_col = 3
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size))
```

```

ax[0,0].imshow(original_img, cmap = 'gray')
ax[0,0].set_title("Original image")

ax[0,1].imshow(equalized_img, cmap = 'gray')
ax[0,1].set_title("Equalized without ROI")

ax[0,2].imshow(equalized_roi, cmap = 'gray')
ax[0,2].set_title("Equalized without ROI")

ax[1,0].plot(cdf_original, color = 'b')
ax[1,0].hist(original_img.flatten(),256,[0,256], color = 'r')
ax[1,0].set_title("Histogram of original image with CDF")
ax[1,0].set_xlim([0,256])

ax[1,1].plot(cdf_equa_img, color = 'b')
ax[1,1].hist(equalized_img.flatten(),256,[0,256], color = 'r')
ax[1,1].set_title("Histogram of equalized image without ROI, with CDF")
ax[1,1].set_xlim([0,256])

ax[1,2].plot(cdf_equa_roi, color = 'b')
ax[1,2].hist(equalized_roi.flatten(),256,[0,256], color = 'r')
ax[1,2].set_title("Histogram of equalized image with ROI, image with CDF")
ax[1,2].set_xlim([0,256])

for a in ax.flatten():
    a.axis("off")

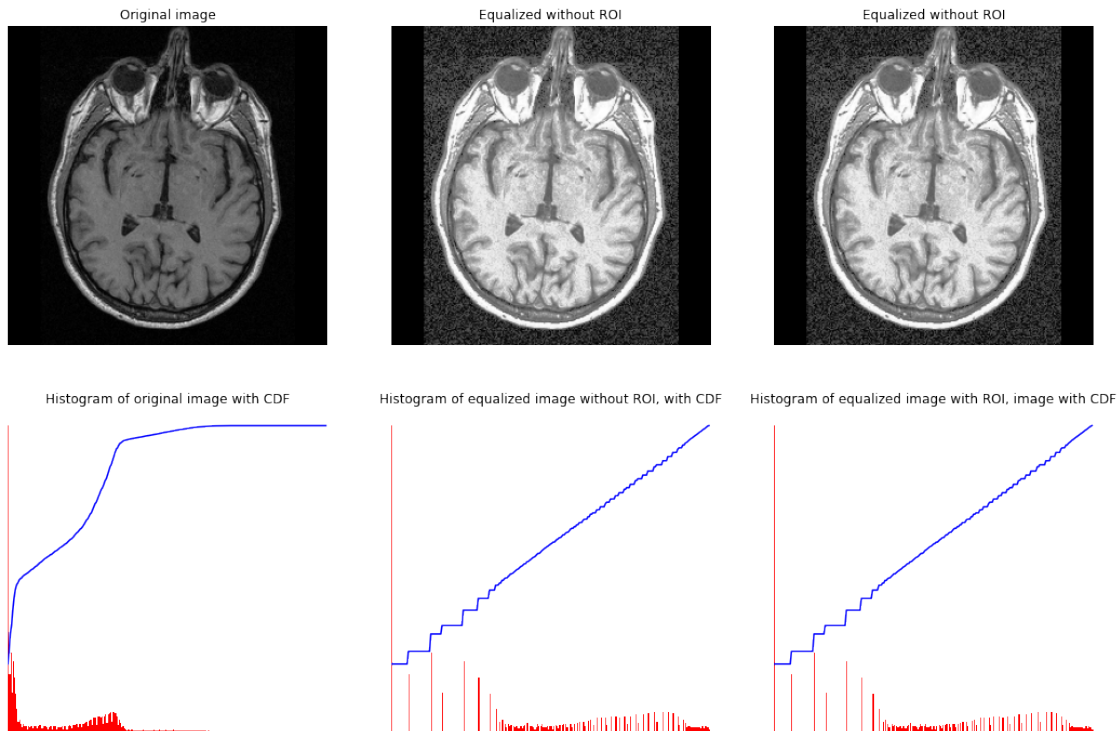
#End of cell
print("Done")

```

Results :

Norm of error between with ROI and without : 0.0

Done



Because the histogram of the original image is skewed on the left, and because of the inherent discrete nature of it, it doesn't impact the equalization, and they both give exactly the same result (the norm of the error between the two equalized histogram is 0). Had the values of the image without the padding been skewed on the right instead, it would have led to very distinct histograms.

Another trial (and error) experimented was an adaptive threshold. Adaptive thresholds work great on pictures which cast shadows, because the threshold is determined automatically from values from neighboring pixels, and maybe that thresholding method would segment better the brain from the skull. The downside is that it requires a new meta-parameter, namely how we determine which pixels are considered neighboring pixels. Here are the result of those investigations, along with the result from plain Otsu thresholding, to illustrate why we didn't keep those implementations :

```
[9]: # Histogram equalization on the whole image
equalized_img = cv.equalizeHist(original_img)

# Otsu thresholding
_, otsu_img = cv.threshold(src=equalized_img, thresh=0, maxval=255, type=(cv.
    THRESH_BINARY | cv.THRESH_OTSU))

# Adaptive thresholding
maxVal=255
constant = 2
```

```

blocksize = 51
img_1 = cv.adaptiveThreshold(src=equalized_img, maxValue=maxVal,
    ↳adaptiveMethod=cv.ADAPTIVE_THRESH_MEAN_C, thresholdType=cv.THRESH_BINARY,
    ↳blockSize=11, C=constant)
img_2 = cv.adaptiveThreshold(src=equalized_img, maxValue=maxVal,
    ↳adaptiveMethod=cv.ADAPTIVE_THRESH_GAUSSIAN_C, thresholdType=cv.THRESH_BINARY,
    ↳blockSize=11, C=constant)

blocksize = 151
img_3 = cv.adaptiveThreshold(src=equalized_img, maxValue=maxVal,
    ↳adaptiveMethod=cv.ADAPTIVE_THRESH_MEAN_C, thresholdType=cv.THRESH_BINARY,
    ↳blockSize=11, C=constant)
img_4 = cv.adaptiveThreshold(src=equalized_img, maxValue=maxVal,
    ↳adaptiveMethod=cv.ADAPTIVE_THRESH_GAUSSIAN_C, thresholdType=cv.THRESH_BINARY,
    ↳blockSize=11, C=constant)

blocksize = 251
img_5 = cv.adaptiveThreshold(src=equalized_img, maxValue=maxVal,
    ↳adaptiveMethod=cv.ADAPTIVE_THRESH_MEAN_C, thresholdType=cv.THRESH_BINARY,
    ↳blockSize=11, C=constant)
img_6 = cv.adaptiveThreshold(src=equalized_img, maxValue=maxVal,
    ↳adaptiveMethod=cv.ADAPTIVE_THRESH_GAUSSIAN_C, thresholdType=cv.THRESH_BINARY,
    ↳blockSize=11, C=constant)

# Plots
n_lin = 4
n_col = 2
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size))

ax[0,0].imshow(otsu_img, cmap = 'gray')
ax[0,0].set_title("Otsu threshold")

# ax[0,1].imshow(equalized_img, cmap = 'gray')
# ax[0,1].set_title("Equalized without ROI")

ax[1,0].imshow(img_1, cmap = 'gray')
ax[1,0].set_title("Adaptive mean threshold, blocksize of 51")

ax[1,1].imshow(img_2, cmap = 'gray')
ax[1,1].set_title("Adaptive gaussian threshold, blocksize of 51")

ax[2,0].imshow(img_3, cmap = 'gray')
ax[2,0].set_title("Adaptive mean threshold, blocksize of 151")

ax[2,1].imshow(img_4, cmap = 'gray')
ax[2,1].set_title("Adaptive gaussian threshold, blocksize of 151")

```

```
ax[3,0].imshow(img_5, cmap = 'gray')
ax[3,0].set_title("Adaptive mean threshold, blocksize of 251")

ax[3,1].imshow(img_6, cmap = 'gray')
ax[3,1].set_title("Adaptive gaussian threshold, blocksize of 251")

for a in ax.flatten():
    a.axis("off")

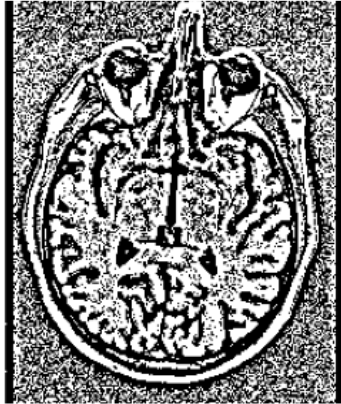
#End of cell
print("Done")
```

Done

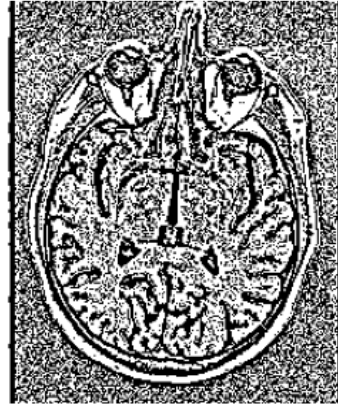
Otsu threshold



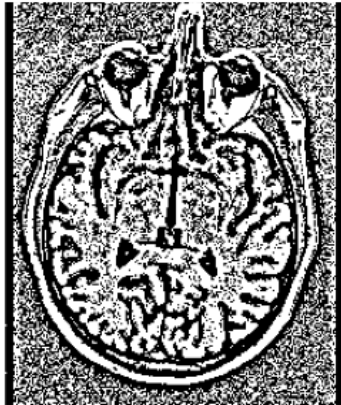
Adaptive mean threshold, blocksize of 51



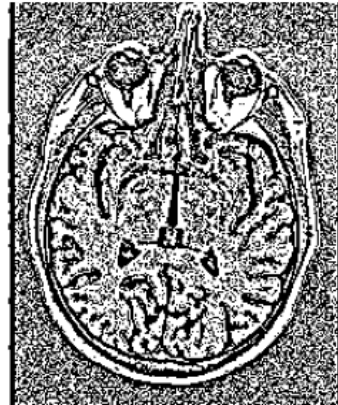
Adaptive gaussian threshold, blocksize of 51



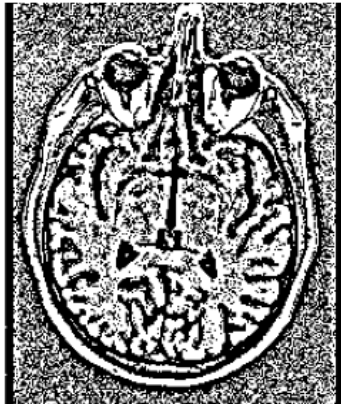
Adaptive mean threshold, blocksize of 151



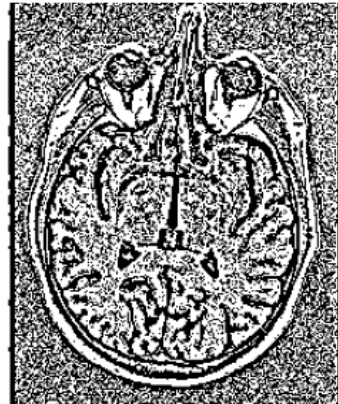
Adaptive gaussian threshold, blocksize of 151



Adaptive mean threshold, blocksize of 251



Adaptive gaussian threshold, blocksize of 251



None of these threshold are as nice as the plain, hand-tuned threshold used (considering our application).

### 1.3 Part 2: Shape/color segmentation

You will find hereafter three pictures taken under three different illuminations, containing some shapes with different colors. We ask you to create a routine to:

1. Count the number of shapes of each color.
2. Compute the total area (in pixels) of each color.

Please note that one specific challenge is to be robust to illumination changes. Therefore some kind of intensity normalization should probably be used.

**Note:** the routine(s) that you will write for this exercise will be useful for the final project as well, so pay special attention to it.

#### 1.3.1 2.1 Visualization

```
[10]: # Load images
img_names = ["arena-shapes-01", "arena-shapes-02", "arena-shapes-03"]
filenames = [os.path.join(data_path, name) + ".png" for name in img_names]
img_files = skimage.io.imread_collection(filenames)
imgs = skimage.io.concatenate_images(img_files)

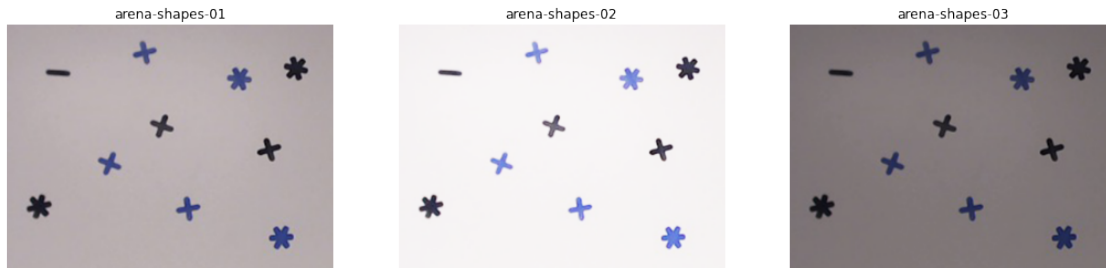
# Print information
img_num = imgs.shape[0]
img_size = (imgs.shape[1], imgs.shape[2])
channel_num = imgs.shape[-1]
print(f"Number of images: {img_num}")
print(f"Image size : {img_size[0]}x{img_size[1]}")
print(f"Number of color channels : {channel_num}")

# Plots
n_lin = 1
n_col = 3
size = 6
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size))

for a, img, name in zip(ax.flatten(), imgs, img_names):
    a.imshow(img)
    a.axis("off")
    a.set_title(name)

# End of cell
print("Done")
```

Number of images: 3  
Image size : 480x640  
Number of color channels : 3  
Done



The idea is to have a robust process that can be applied in exactly the same way to each image to have the same result. For that, we wrote the function `mask`, `grayscale_img`, `equalized_img`, `filtered_img`, `closed_img` = `filterColorFromImg(image, colorRange)`. That function : - Takes an RGB input image - Converts it to grayscale - Equalizes the grayscale image's histogram to attenuate the effect of the light - Filters the image for a color range passed in parameter - Closes the image and then opens it to improve the quality.

All the intermediate steps are returned by the function, in the computed order, except for the final mask which is returned first, because it's the most important returned object of the function.

This function works very well, as can be seen on the steps displayed below for every image, along with the implementation :

```
[11]: def filterColorFromImg(image, colorRange):
    grayscale_img = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    equalized_img = cv.equalizeHist(grayscale_img)
    filtered_img = cv.inRange(equalized_img, colorRange[0], colorRange[1])
    kernel = np.zeros((5,5), np.uint8)
    cv.circle(img=kernel, center=(2,2), radius=2, color=255, thickness=-1)
    closed_img = cv.morphologyEx(filtered_img, cv.MORPH_CLOSE, kernel,
    → iterations=1)
    mask = cv.morphologyEx(closed_img, cv.MORPH_OPEN, kernel, iterations=1)

    return mask, grayscale_img, equalized_img, filtered_img, closed_img

# threshold ranges
blueRange = (4,8)
blackRange = (0,3)

#plot parameters
n_lin = 9
n_col = 3
size = 6
```

```

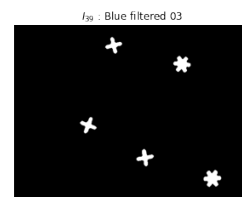
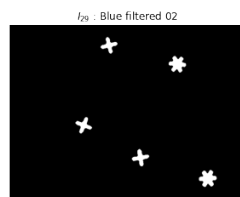
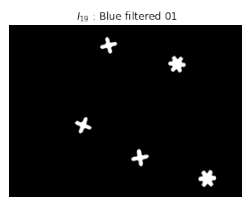
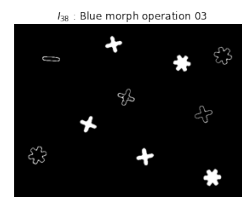
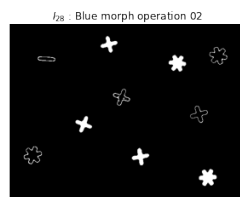
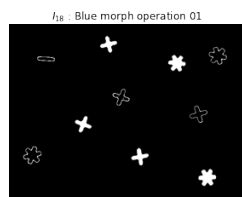
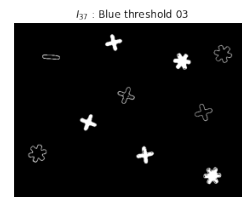
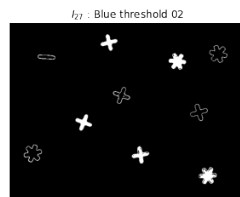
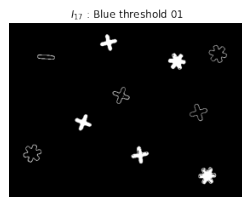
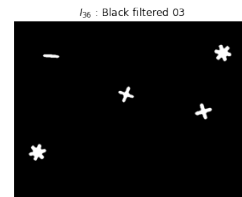
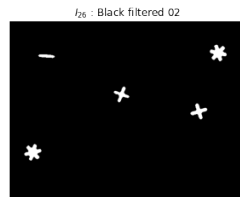
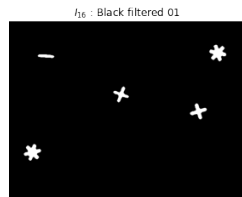
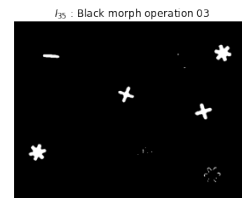
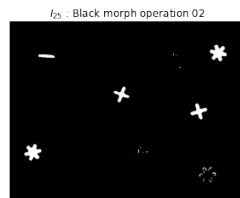
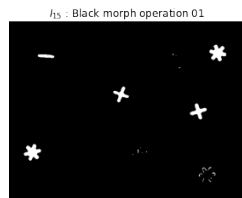
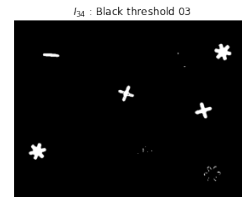
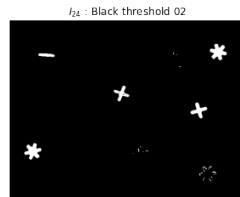
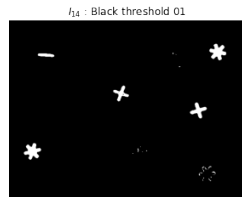
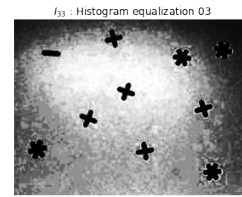
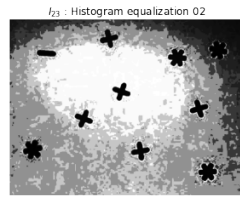
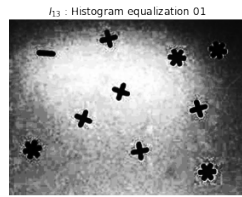
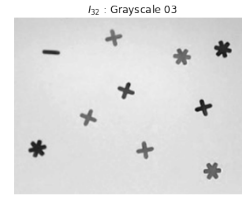
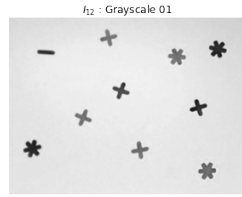
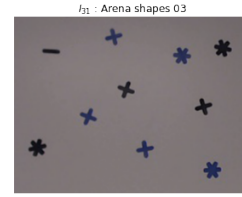
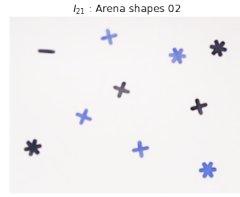
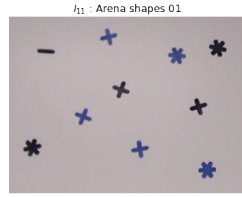
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size*img_size[0]/
→img_size[1]))

#definitions
black = 0
blue = 1
labels = ["Arena shapes", "Grayscale", "Histogram equalization", "Black_
→threshold", "Black morph operation", "Black filtered", "Blue threshold", "Blue_
→morph operation", "Blue filtered"]

# loop to print all different images
for col, im in enumerate(imgs):
    bkf,gs,he,bks,bkm = filterColorFromImg(im,blackRange)
    blf,_,_,bls,blm = filterColorFromImg(im,blueRange)
    for idx,im in enumerate([im,gs,he,bks,bkm,bkf,bls,blm,blf]):
        ax[idx,col].imshow(im, cmap="gray")
        ax[idx,col].set_title("$I_{" + str(col+1) + str(idx+1) + "}$ : " +
→labels[idx] + " 0" + str(col+1))

for a in ax.flatten():
    a.axis("off")

```



The closing is necessary, because we can see some tiny black spots within the shapes before it being performed. The opening is necessary, because we can see the outlines of the shapes of another color before it being performed.

At first, we had tried to filter by color using the RGB colorspace, but to no avail. Then we tried the HSV colorspace, but it didn't work either. Finally, we tried the much simpler approach of a grayscale image, and it works very well.

### 1.3.2 2.2 Number of shapes of each color

Once the filtering is done, counting the number of shapes is very easy with labeling. OpenCV provides a labelling function, which returns the number of objects in the image, including the image itself (thus we print the return value minus 1). It also returns labels, but here we don't care about them, we just want the number of labels.

```
[12]: # Processing, results and plot
n_lin = 3
n_col = 3
size = 6
fig, ax = plt.subplots(n_lin, n_col, figsize=(n_col*size, n_lin*size*img_size[0]/
    →img_size[1]))
print(f"Results : ")
for img, name, i in zip(imgs, img_names, range(img_num)):
    black_img,_,_,_,_ = filterColorFromImg(img,blackRange)
    blue_img,_,_,_,_ = filterColorFromImg(img,blueRange)
    black_shape_num,_ = cv.connectedComponents(image=black_img)
    blue_shape_num,_ = cv.connectedComponents(image=blue_img)

    print(f"    Number of black shapes in {name} : {black_shape_num-1}")
    print(f"    Number of black shapes in {name} : {blue_shape_num-1}")
    print(f"")

    ax[0,i].imshow(img, cmap="gray")
    ax[0,i].set_title(name)

    ax[1,i].imshow(black_img, cmap="gray")
    ax[1,i].set_title("Black filter on " + name)

    ax[2,i].imshow(blue_img, cmap="gray")
    ax[2,i].set_title("Blue filter on " + name)

for a in ax.flatten():
    a.axis("off")

# End of cell
print(f"End of cell")
```

Results :

Number of black shapes in arena-shapes-01 : 5

Number of black shapes in arena-shapes-01 : 5

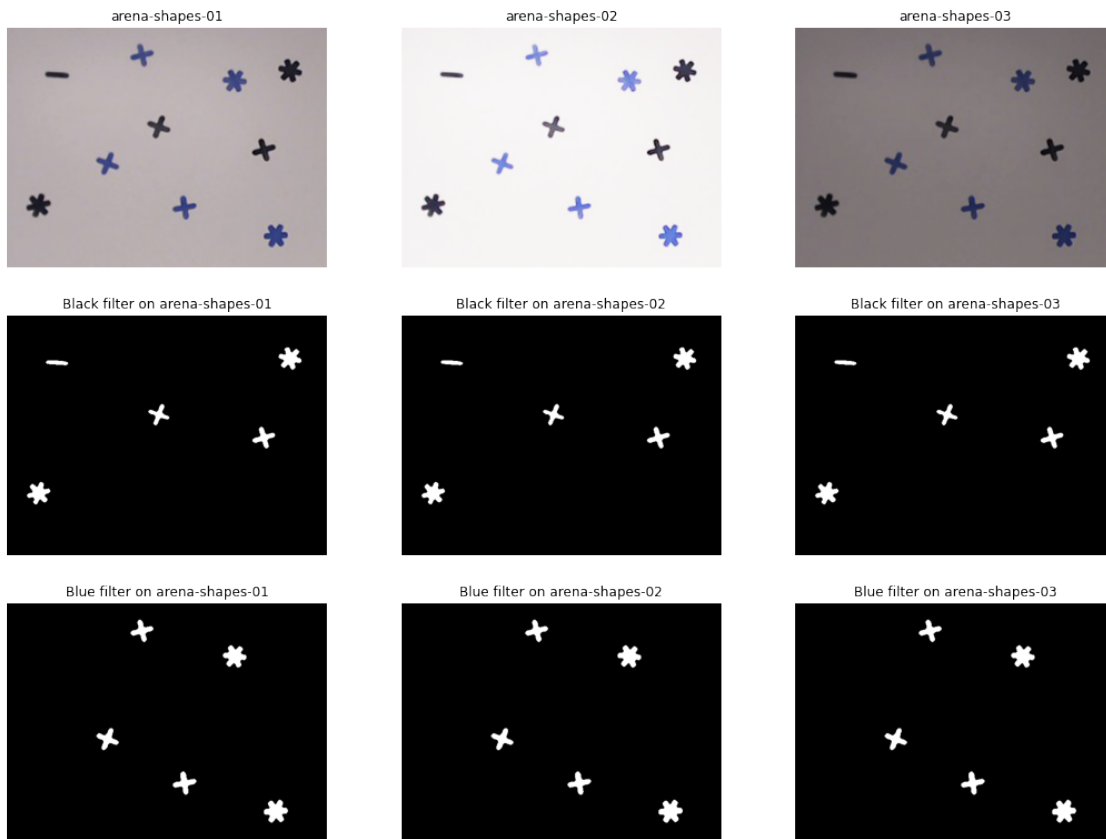
Number of black shapes in arena-shapes-02 : 5

Number of black shapes in arena-shapes-02 : 5

Number of black shapes in arena-shapes-03 : 5

Number of black shapes in arena-shapes-03 : 5

End of cell



### 1.3.3 2.3 Total area (in pixels) of each color

Once the filtering is done, computing the total area in pixels is very easily done by summing the number of pixels with the value 255 (white) in each image.

```
[13]: print(f"Results : ")
      for img, name, i in zip(imgs, img_names, range(img_num)):
          # Extraction
          black_shapes,_,_,_,_ = filterColorFromImg(img,blackRange)
```

```

blue_shapes,_,_,_,_ = filterColorFromImg(img,blueRange)
black_pixel_num = np.sum(black_shapes == 255)
blue_pixel_num = np.sum(blue_shapes == 255)

# Results
print(f"    There are {black_pixel_num} black px and {blue_pixel_num} blue_
→px in {name}.")

```

Results :

There are 4016 black px and 5049 blue px in arena-shapes-01.

There are 3991 black px and 5087 blue px in arena-shapes-02.

There are 3995 black px and 5064 blue px in arena-shapes-03.

We see that the number of shapes found is perfectly the same, but the number of pixels is slightly different. That is normal, since finding different number of shapes would mean that the algorithm is not robust at all; while the slight differences in pixels number (area) are due to the variations within the pictures (lighting within the different takes of the same scene for the three pictures).