CSE 577: Introduction to Medical Imaging

Name: Hritam Basak
SBU ID: 114783055

Question 1:

(a) For heart image, the threshold value is set to 165.
(b) For brain image, the threshold value is set to 155.

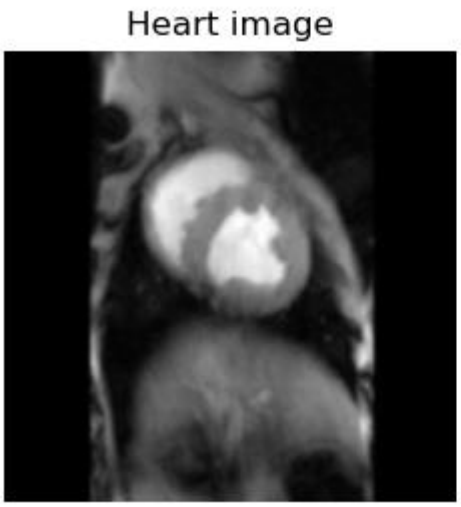
Explanation: For a grayscale image, the pixel intensity values are set between 0 and 255. The goal of thresholding is, if a specific pixel intensity is below threshold value, put it to zero, and if it is above the threshold, put it to one. Thus, the output of thresholding is a binary image.

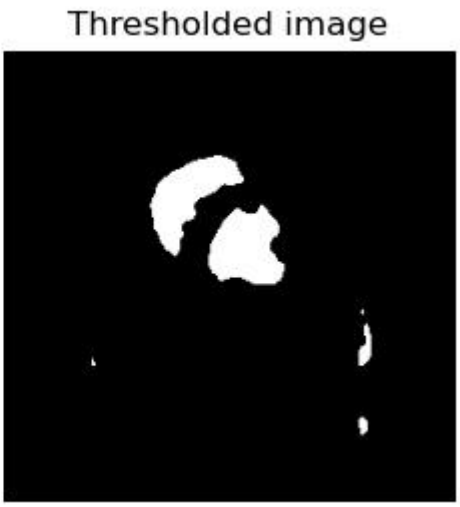
Here is the function that I developed for thresholding:

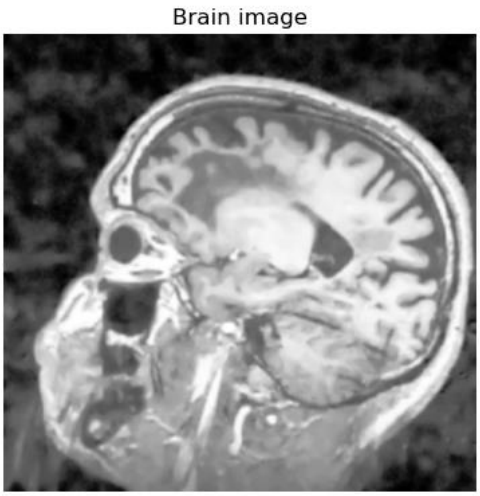
```
# function for image thresholding
def imThreshold(img, threshold, maxVal):
    assert len(img.shape) == 2 # input image has to be gray

    height, width = img.shape   # getting image dimension
    bi_img = np.zeros((height, width), dtype=np.uint8)      # creating a matrix of image size filled with zeros
    for x in range(height):
        for y in range(width):
            if img.item(x, y) > threshold:
                bi_img.itemset((x, y), maxVal)              # If pixel intensity > threshold, set to 255, otherwise 0

    return bi_img
```

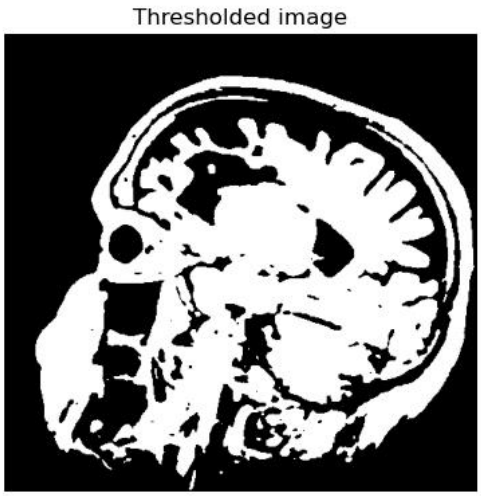
This function has three parameters as input: image: the original input image; threshold: the intended threshold value; maxVal: the maximum intensity value of the image, which is set to 255. In the function, first I am creating a 2D matrix with elements as zero, with same size as the input image. Then, I am checking for every pixel, if the pixel intensity value is more than threshold, I set it to 255, otherwise 0.

The output for the heart and brain images looks like this:









For brain image, I am cropping the image to remove the border. Both the images are smoothed using a gaussian kernel with size (5,5) with zero mean as a pre-processing for better output.

Question 2:

WATERSHED SEGMENTATION:

The watershed segmentation is a classical image segmentation method that is often used to segment objects in an image based on their boundaries.

The basic idea behind the watershed segmentation is to view the image as a topographic surface where the intensity values represent the height of the surface. The goal is to identify the "basins" in this surface that correspond to the different objects in the image. The watershed segmentation method can be broken down into the following steps:

1. First we compute the gradient magnitude of the image, which represents the magnitude of the intensity changes at each pixel location:

$G(x, y) = ||\nabla I(x, y)||$

where $I(x, y)$ is the intensity of the image at pixel (x, y), and $\nabla I(x, y)$ is the gradient of the image at that location.

2. Then we threshold the gradient image to obtain a binary image that highlights the boundaries between different regions:

$B(x, y) = \{1 \text{ if } G(x, y) > T, 0 \text{ otherwise}\}$

where T is the threshold value.

3. Then we compute the distance transform of the binary image, which assigns a distance value to each pixel based on its distance to the nearest boundary:

$D(x, y) = \min\{||p-q||\}$ for all q such that $B(q) = 1$

where $p = (x, y)$ is the current pixel location.

4. Then we compute the watershed transform of the distance image, which assigns a label to each pixel based on its "catchment basin":

W(x, y) = {i if L(p) = i for all p such that D(p) <= D(x, y)}

where L(p) is the label of the pixel p in the binary image.

In this way, the watershed segmentation method assigns each pixel in the image to a particular object label based on the basin in which it falls. The resulting segmented image consists of different objects separated by their boundaries.

Note that the watershed segmentation method can suffer from over-segmentation or under-segmentation, especially in regions with flat or low-contrast regions. Therefore, additional post-processing steps such as merging, splitting, or using prior knowledge can be used to refine the segmentation results.

Here is the following code for watershed segmentation of heart image:

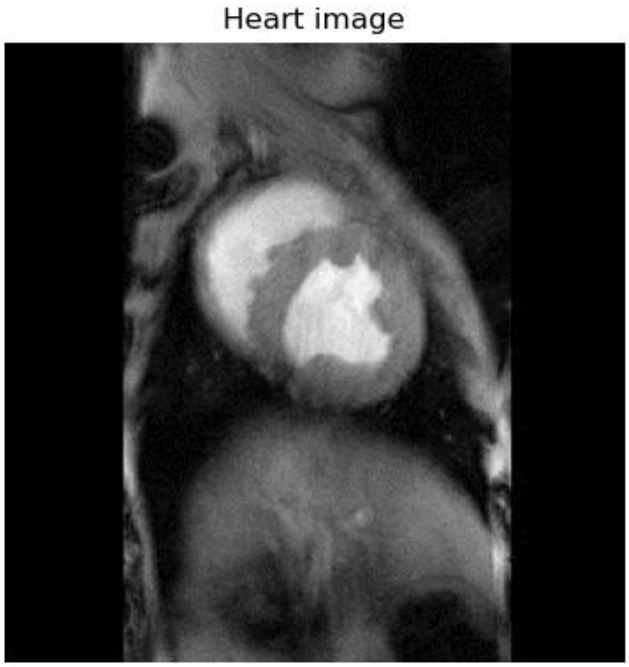
```
# Read image
img = cv2.imread('./heart.jpg')
border = 5
# Convert image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Threshold the image to obtain a binary image
ret, thresh = cv2.threshold(gray, 170, 255, cv2.THRESH_BINARY)

# Perform morphological opening to remove noise and small objects
kernel = np.ones((3,3),np.uint8)
opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations=1)
closing = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel, iterations=1)

# Compute the distance transform of the binary image
dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
ret, sure_fg = cv2.threshold(dist_transform, 0.07*dist_transform.max(), 255, cv2.THRESH_BINARY)

# Compute the watershed transform of the distance image
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(opening, sure_fg)
ret, markers = cv2.connectedComponents(sure_fg)

markers = markers + 1

markers[unknown==255] = 0
markers = cv2.watershed(img, markers)
img[markers == -1] = [255,0,0]
img = img[border:-border, border:-border]
```

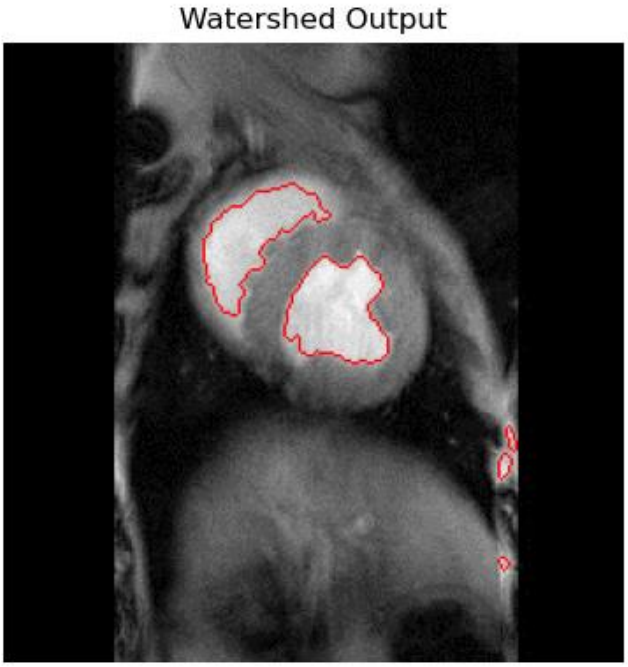
Explanation:

1. The first few lines of code read in an image of a heart and define a border value.
2. The image is then converted to grayscale using cv2.cvtColor() function.
3. The grayscale image is thresholded using cv2.threshold() to obtain a binary image. This binary image is then used for further processing.
4. Next, morphological opening and closing operations are applied to remove noise and small objects from the binary image.
5. The distance transform of the binary image is computed using cv2.distanceTransform(). This computes the distance of each pixel in the binary image to the nearest background pixel.
6. A threshold is applied to the distance transform image to create a sure foreground marker using cv2.threshold().

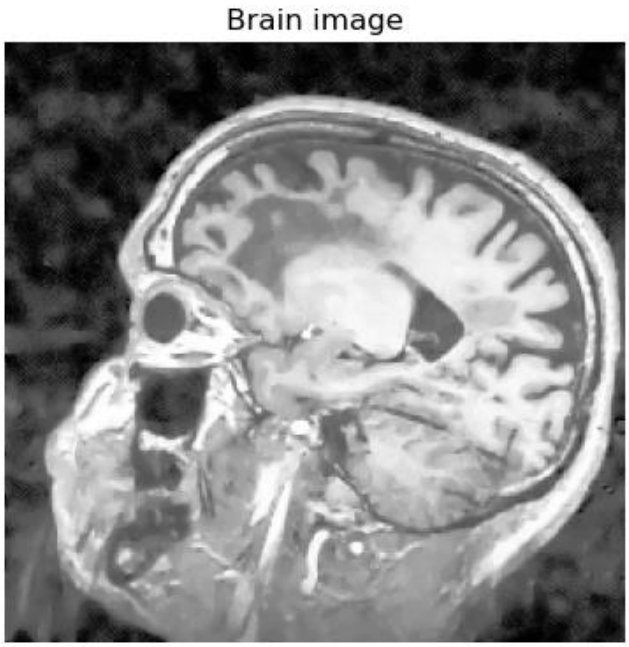
7. The watershed transform of the distance image is computed using cv2.watershed(). This algorithm assigns each pixel to a marker based on its distance to the nearest background pixel.
8. The resulting markers are used to segment the original image using cv2.watershed() again.
9. The segmentation result is displayed in a plot with the original grayscale image.

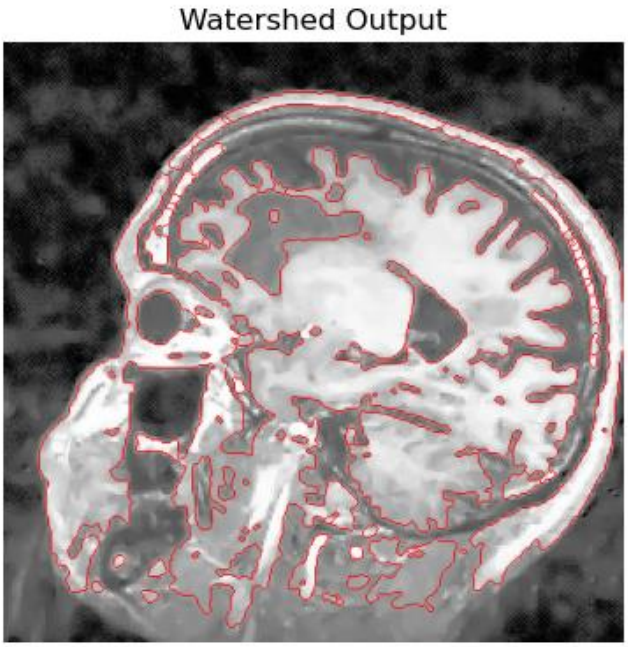
Overall, the code uses several image processing techniques to segment the image into different regions based on the distance to the nearest background pixel, and the result is displayed using matplotlib. The output looks like this:





For Brain image, the result looks like this:





Question 3:

Here we first calculate the Gaussian blurred version of the image, then we take the Laplacian of it, which generates the Laplacian of Gaussian filtered output. Then we check the zero crossing of the output, and whenever it has a zero crossing, we consider it as an edge. Here is the code:

```
def zero_cross_detection(image):
    edges = np.zeros_like(image)
    edges[np.where(np.diff(np.sign(image)))] = 255
    return edges
```

Explanation:

First, I create a function zero_cross_detection() which will be used later. The function applies a zero-crossing edge detection algorithm to the input image and returns the resulting edges as a new image.

The implementation of the algorithm uses NumPy functions to create a new array **edges** with the same shape as the input **image**. Then, the NumPy **np.diff** function is used to calculate the difference between adjacent elements in the **image** along the vertical and horizontal axes. The **np.sign** function returns the sign of each element in the resulting difference array. Then, another NumPy function **np.where** is used to get the coordinates of where the sign of the difference changes from negative to positive, or from positive to negative, indicating the location of a zero crossing.

Finally, the **edges** array is set to 255 (white) at the coordinates of the zero crossings, and the resulting **edges** array is returned.

Then this is the main code:

```
# Load the image
img = cv2.imread('./heart.jpg', 0)

plt.figure(figsize=(5,5))
plt.imshow(img, 'gray')
plt.axis('off')
plt.title('Heart Image')
plt.show()

# Apply the LoG operator to the image
sigmas = [1, 2, 5, 10]
i = 0
for sigma in sigmas:
    i = i+1
    img_gaussian = cv2.GaussianBlur(img, (11, 11), sigma)
    img_log = cv2.Laplacian(img_gaussian, cv2.CV_64F)

    img_edges = zero_cross_detection(img_log)

    # Display the original image and the edges
    plt.figure(figsize=(20, 20))
    plt.subplot(1, 8, i)
    plt.imshow(img_log, 'gray')
    plt.title(f'Laplacian(Sigma = {sigma})')
    plt.axis('off')
    plt.subplot(1, 8, i+1)
    plt.imshow(img_edges, 'gray')
    plt.title(f'Edges(Sigma = {sigma})')
    plt.axis('off')
```

Explanation: Next, the code applies the Laplacian of Gaussian (LoG) edge detection operator to the image for different values of the standard deviation (**sigma**) of the Gaussian blur. A **for** loop is used to iterate through the list of **sigmas** and apply the operator on each iteration.

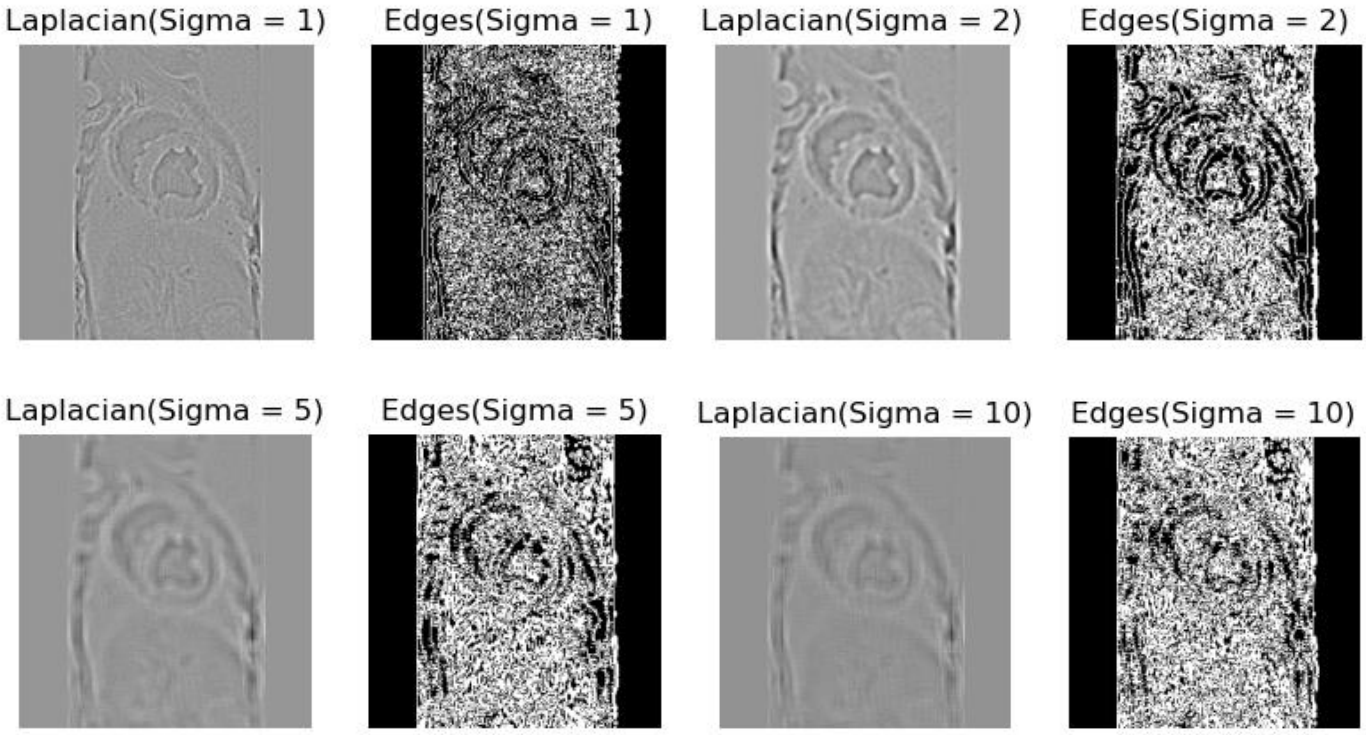
For each **sigma** value, the image is first blurred using OpenCV's **GaussianBlur** function with a kernel size of (11,11) and the current **sigma** value. The LoG operator is then applied to the blurred image using OpenCV's **Laplacian** function with the data type **CV_64F** for the output image.

After the LoG operator is applied, the zero-crossing detection function **zero_cross_detection** is applied to the LoG output image to detect edges.

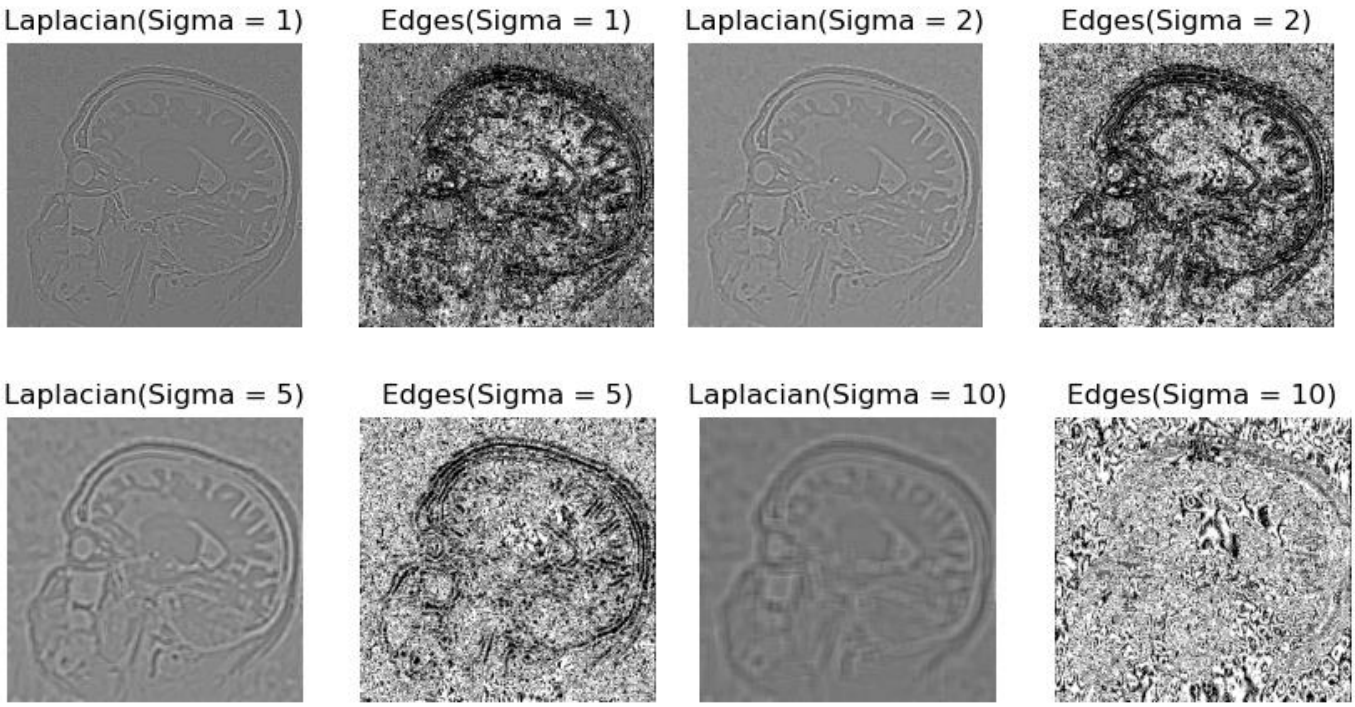
Finally, the Laplacian filtered image, and the detected edges are displayed side-by-side in separate subplots using **plt.subplot** and **plt.imshow** functions with appropriate titles and **axis** off to remove the axes.

The output plots will show a series of subplots with the Laplacian filtered images and the edges detected from each image with varying sigma values.

Here are the output for heart image:



And here are the outputs for brain image:



Question 4:

For histogram equalization, I write a function as below:

```
def histogram_equalization(img):
    hist, _ = np.histogram(img.flatten(), 256,[0,255])     # get the histogram
    H, W = img.shape
    PDF = hist / np.sum(hist)             # generate the probability distribution function
    CDF = PDF.cumsum()                    # get the cumulative distribution function

    # transforming the V channel
    CDF_transform = (CDF - CDF.min()) / (CDF.max() - CDF.min()) * 255
    CDF_transform = CDF_transform.astype('uint8')
    img_transform = CDF_transform[np.asarray(img).flatten()]
    img_transform = img_transform.reshape(H, W)

    return img_transform
```

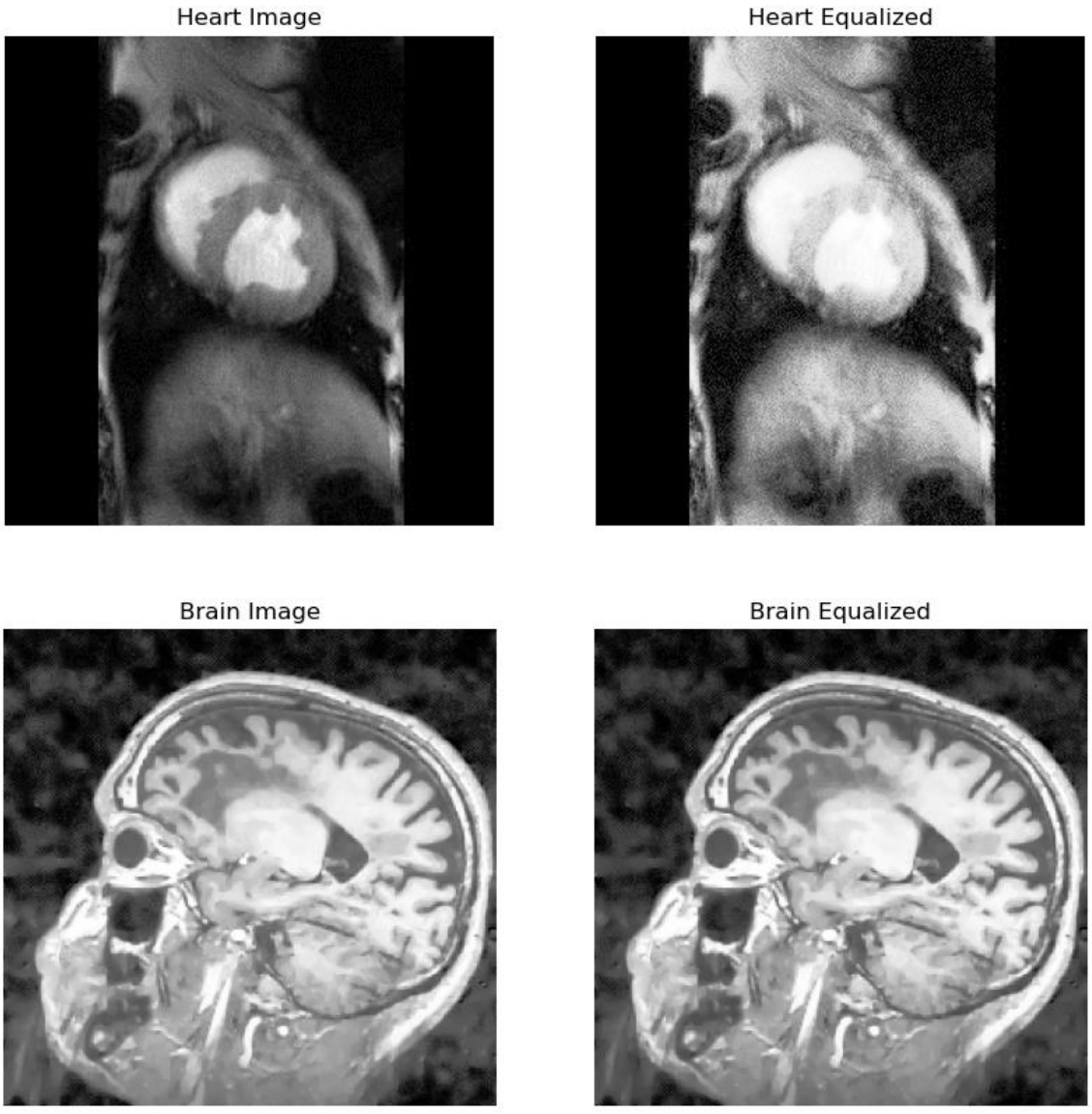
Explanation: This code performs histogram equalization on a given grayscale image. Here is a step-by-step explanation of the code:

1. The **histogram_equalization()** function takes a grayscale image **img** as input.
2. The first line computes the histogram of the image using the **np.histogram()** function. The **flatten()** method is used to convert the 2D image array into a 1D array for easier processing. The histogram has 256 bins (one for each possible pixel value) and spans the range [0, 255].
3. The second line gets the height and width of the image.
4. The third line generates the probability distribution function (PDF) by dividing the histogram by the sum of all its values. This function represents the probability that a pixel takes on a certain value.

5. The fourth line computes the cumulative distribution function (CDF) by taking the cumulative sum of the PDF. The CDF represents the probability that a pixel takes on a value less than or equal to a certain value.
6. The fifth line transforms the CDF so that it spans the range [0, 255] by normalizing it and then multiplying by 255. This step is necessary because the CDF can take on any value between 0 and 1, but the image can only take on integer values between 0 and 255.
7. The sixth line converts the transformed CDF to a data type of unsigned integer (uint8) so that it can be used to index the image.
8. The seventh line applies the transformation to the image by flattening it, indexing it with the transformed CDF, and then reshaping it back into its original dimensions. This step maps the pixel values of the image to new values that have a more uniform distribution across the range of possible values.
9. The final line returns the transformed image.

Overall, the **histogram_equalization()** function enhances the contrast of a grayscale image by redistributing the pixel values to span the entire range of possible values. This is achieved by computing the PDF and CDF of the image and then transforming the CDF to have a more uniform distribution. The transformed CDF is then used to map the pixel values of the image to new values that have a more uniform distribution across the range of possible values.

Here is the output for heart and brain images:



Question 5:

Here is the code:

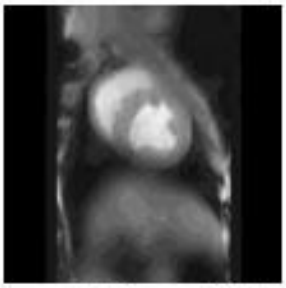
```
def genGaussianKernel(width, sigma):
    kernel = np.zeros([width,width])            # creating a blank matrix of kernel size
    for x in range(width):
      for y in range(width):
          kernel[x,y] = 1/(2*np.pi*sigma**2)*np.exp(-((x-width//2)**2+(y-width//2)**2)/(2*sigma**2))
    kernel_2d = kernel/np.sum(kernel)       # Normalizing the kernel
    return kernel_2d

sigmas = [10, 15, 20]
alphas = [0.1, 0.5, 0.9]

Heart = cv2.imread('./heart.jpg', 0)

plt.imshow(Heart, 'gray')
plt.axis('off')
plt.title('Heart Image')
plt.show()
i = 0
for sigma in sigmas:
   for alpha in alphas:
        i = i+1
        H = genGaussianKernel(2*sigma+1, sigma)
        out = Heart + alpha * (Heart - cv2.filter2D(Heart, -1, H ))
        # plt.figure(figsize=(5,5))

        plt.subplot(3,3,i)
        plt.imshow(out, 'gray')
        plt.title(f'(s={sigma},a={alpha})')
        plt.axis('off')
```

This solution results in a hybrid filtered image. First, I am writing a function genGaussianKernel() that creates a gaussian kernel of a fixed width and given sigma values. Then after reading the image, we perform the specified set of operation for different sigma and alpha values.

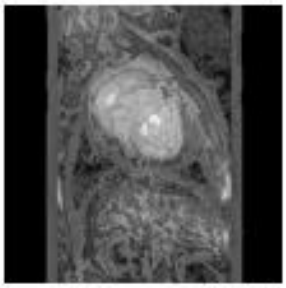
Explanation: This code generates a filtered version of the image with sharpened version of the image. First, we generate a gaussian blurred version of the image F, which reduces the high-frequency components (i.e. low-pass filter), the output is F*H, where H is the gaussian kernel. Then we subtract F*H from the original image F, hence we keep the high-frequency components only (i.e. F – F*H). Now we add a scaled version of this (F – F*H) with the original image F using a scaling factor alpha. This alpha parameter sets the contribution of sharpened edges and high-frequency components to the final output.

Hence, the final output is a sharpened version of the original image, i.e. it performs image sharpening. Here are the outputs with different alpha (a) and sigma (s) values:

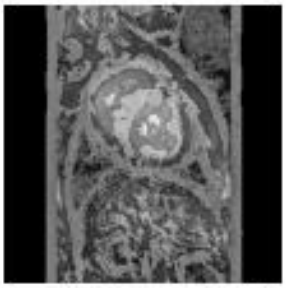
(s=10,a=0.1)



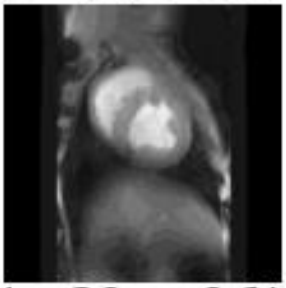
(s=10,a=0.5)



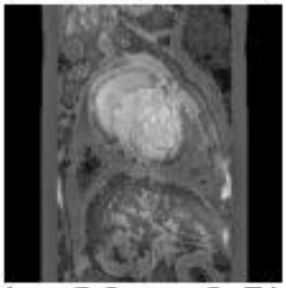
(s=10,a=0.9)



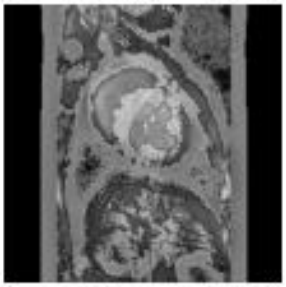
(s=15,a=0.1)



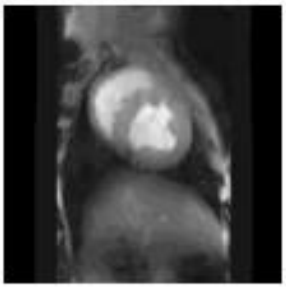
(s=15,a=0.5)



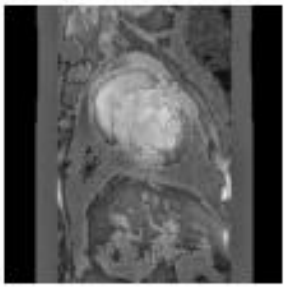
(s=15,a=0.9)



(s=20,a=0.1)



(s=20,a=0.5)



(s=20,a=0.9)