CSE 577: Introduction to Medical Imaging

Name: Hritam Basak
SBU ID: 114783055

1. **Testing the functionality of image segmentation techniques and comparing their accuracy requires us to work with images for which the correct segmentation is known. Develop some test images by:**
   (a) Create an image SEG1 containing artificial objects on a background of constant gray-level. Generate simple geometric objects such as squares, rectangles, diamonds, stars, circles, etc., each having a constant gray-level different from that of the background, some of them darker and some brighter than the background. Determine the area of each object and store it in an appropriate form.

Ans: For this question, I have set the background grayscale value as 127. On top of this, I have created multiple objects (circle, square, rectangle, star, diamond) with different intensity values. Some of them have intensity more than the background and some has less than the background. Here is the code:

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


# Define image dimensions and background gray level
# Define image dimensions and background gray level
width = 520
height = 520
bg_gray_level = 128


def draw_square(img, size, x, y, gray_level):
    """
    Draws a square on the image with the specified size, position, and gray level.
    """
    half_size = size // 2
    for i in range(-half_size, half_size+1):
        for j in range(-half_size, half_size+1):
            img[y+i, x+j] = gray_level

def draw_rectangle(img, width, height, x, y, gray_level):
    """
    Draws a rectangle on the image with the specified width, height, position, and gray
level.
    """
    half_width = width // 2
    half_height = height // 2
    for i in range(-half_height, half_height+1):
        for j in range(-half_width, half_width+1):
            img[y+i, x+j] = gray_level
```

```python
def draw_diamond(img, size, x, y, gray_level):
    """
    Draws a diamond on the image with the specified size, position, and gray level.
    """
    half_size = size // 2
    for i in range(-half_size, half_size+1):
        for j in range(-half_size, half_size+1):
            if abs(i) + abs(j) <= half_size:
                img[y+i, x+j] = gray_level


def draw_star(img, size, x, y, gray_level):
    """
    Draws a filled star on the image with the specified size, position, and gray level.
    """
    R = size // 2
    r = R // 2.1
    points = []
    for i in range(5):
        theta = i * 4 * math.pi / 5
        x1 = x + R * math.cos(theta)
        y1 = y + R * math.sin(theta)
        points.append((int(x1), int(y1)))
        theta += 2 * math.pi / 5
        x2 = x + r * math.cos(theta)
        y2 = y + r * math.sin(theta)
        points.append((int(x2), int(y2)))
    pts = np.array(points, np.int32)
    mask = np.zeros(img.shape[:2], np.uint8)
    cv2.fillPoly(mask, [pts], gray_level)
    img[mask == gray_level] = gray_level

    # Draw a small filled circle at the center of the star
    center_size = size // 4
    center_x = x
    center_y = y
    cv2.circle(img, (center_x, center_y), center_size, gray_level, -1)


def draw_circle(img, size, x, y, gray_level):
    """
    Draws a circle on the image with the specified size, position, and gray level.
    """
    radius = size // 2
    cv2.circle(img, (x, y), radius, gray_level, -1)

def create_object(shape, size, x, y, gray_level):
    """
    Creates an object with the specified shape, size, position, and gray level.
    """
    if shape == "square":
        return draw_square, size, x, y, gray_level
    elif shape == "rectangle":
```

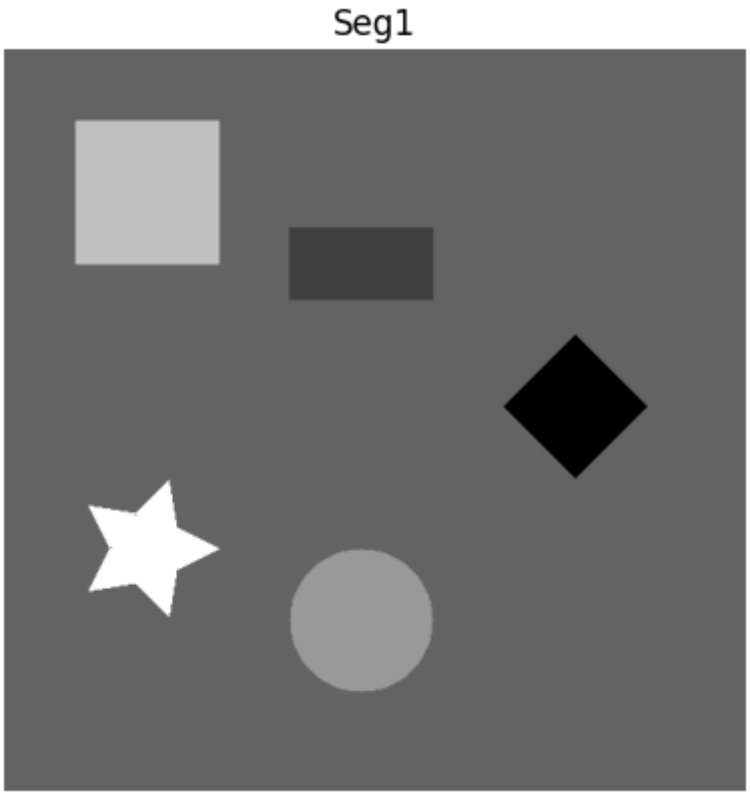
```
        return draw_rectangle, size, size // 2, x, y, gray_level
    elif shape == "diamond":
        return draw_diamond, size, x, y, gray_level
    elif shape == "star":
        return draw_star, size, x, y, gray_level
    elif shape == "circle":
        return draw_circle, size, x, y, gray_level

# Create image
img = np.ones((height, width), dtype=np.uint8) * bg_gray_level

# Define objects
objects = [
    ("square", 100, 100, 100, 200),
    ("rectangle", 100, 250, 150, 100),
    ("diamond", 100, 400, 250, 50),
    ("star", 100, 100, 350, 250),
    ("circle", 100, 250, 400, 170)
]

# Draw objects and store their areas
areas = []
for obj in objects:
    shape, size, x, y, gray_level = obj
    draw_func, *draw_args = create_object(shape, size, x, y, gray_level)
    draw_func(img, *draw_args)
    contours, hierarchy = cv2.findContours((img == gray_level).astype(np.uint8),
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    area = cv2.contourArea(contours[0])
    # area = size ** 2 if shape in ("square", "rectangle") else math.pi * (size/2)**2 if
shape == "circle" else size**2/2
    # area = np.count_nonzero(img == gray_level)
    areas.append((shape, area))

# Display image and print areas
plt.imshow(img, 'gray')
plt.title("Seg1")
plt.axis('off')
plt.show()

cv2.imwrite('/content/drive/MyDrive/Medical Imaging Final/Output/SEG1.png', img)

print("Areas:")
for shape, area in areas:
    print(f"{shape}: {area}")
```

The code is almost self-explanatory. First, the code defines the dimensions of the image and background gray level. Then, it defines functions to draw different shapes on the image such as square, rectangle, diamond, star, and circle.

The **create_object()** function takes the shape, size, position, and gray level of an object and returns a tuple with the corresponding drawing function and its arguments.

After defining the objects, the code creates the image using NumPy and fills it with the background gray level. The **draw_objects()** function is called for each object, which draws the object on the image using the corresponding drawing function obtained from **create_object()**. It also calculates the area of the object using the **cv2.findContours()** and **cv2.contourArea()** functions.

Finally, the code displays the generated image using matplotlib, saves the image to a file, and prints the areas of the different shapes. Also, I have printed the areas of individual shapes. Here is the output of the above code:



Areas:

square: 10000.0

rectangle: 5000.0

diamond: 5000.0

star: 3753.5

circle: 7704.0

(b) **Superimpose additive Gaussian noise with a given standard deviation, thus creating an image SEG2**

Ans: For this problem, we can use cv2.randn() to generate gaussian noise. We have to define the mean and standard deviation first, and then we can generate the gaussian noise, followed by adding it to the original image. Here is the code:

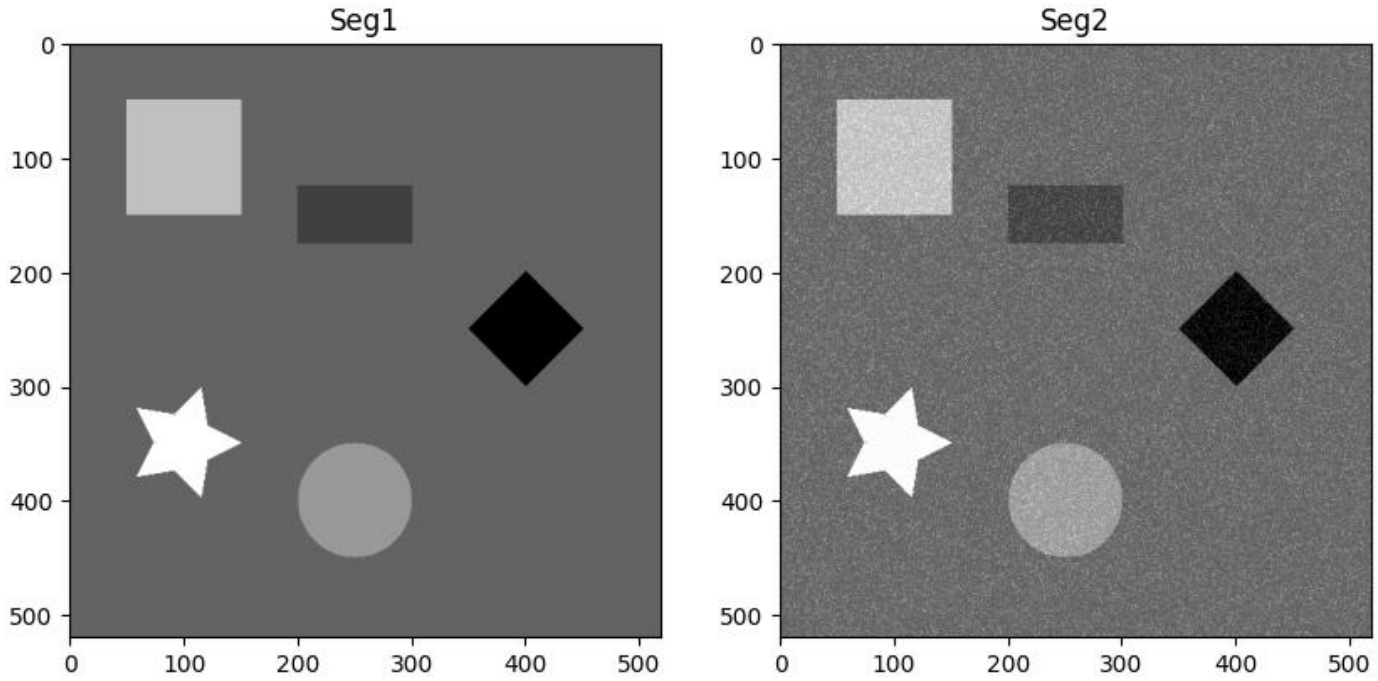
```
mean = 0
stddev = 25
noise = np.zeros_like(img)
```

```
cv2.randn(noise, mean, stddev)
noisy_img = cv2.add(img, noise, dtype=cv2.CV_8UC1)

_, axis = plt.subplots(1, 2, figsize = (10,10))
axis[0].imshow(img, 'gray')
axis[0].set_title('Seg1')

axis[1].imshow(noisy_img, 'gray')
axis[1].set_title('Seg2')

plt.show()
```

Here is a step-by-step explanation:

1. **mean = 0** and **stddev = 25**: These lines define the mean and standard deviation of the Gaussian distribution that will be used to generate the noise.
2. **noise = np.zeros_like(img)**: This line creates a numpy array of zeros with the same shape as the input image, **img**.
3. **cv2.randn(noise, mean, stddev)**: This line uses the **cv2.randn()** function from the OpenCV library to fill the **noise** array with random values drawn from a Gaussian distribution with the specified mean and standard deviation.
4. **noisy_img = cv2.add(img, noise, dtype=cv2.CV_8UC1)**: This line adds the generated noise to the original image, **img**, using the **cv2.add()** function. The **dtype=cv2.CV_8UC1** parameter specifies that the resulting noisy image should have pixel values represented as unsigned 8-bit integers.

Then we show the output, which looks like this:



Clearly, the Seg2 image has some random gaussian noise as compared to Seg1.

c) **Superimpose random impulse noise of a given severity over the image SEG2, thus creating an image SEG3.**

Ans: Here is the function that I wrote for generating impulse noise.

```
def add_impulse_noise(image, prob):
    h, w = image.shape[:2]
    output = np.copy(image)

    # Generate a random matrix with probabilities for each pixel
    prob_matrix = np.random.rand(h, w)

    # Add salt noise
    output[prob_matrix < prob / 2] = 0

    # Add pepper noise
    output[prob_matrix > 1 - prob / 2] = 255

    return output
```

The function takes two parameters:

1. **image**: a 2D numpy array representing the input image.
2. **prob**: a float value between 0 and 1 representing the probability of adding noise to each pixel.

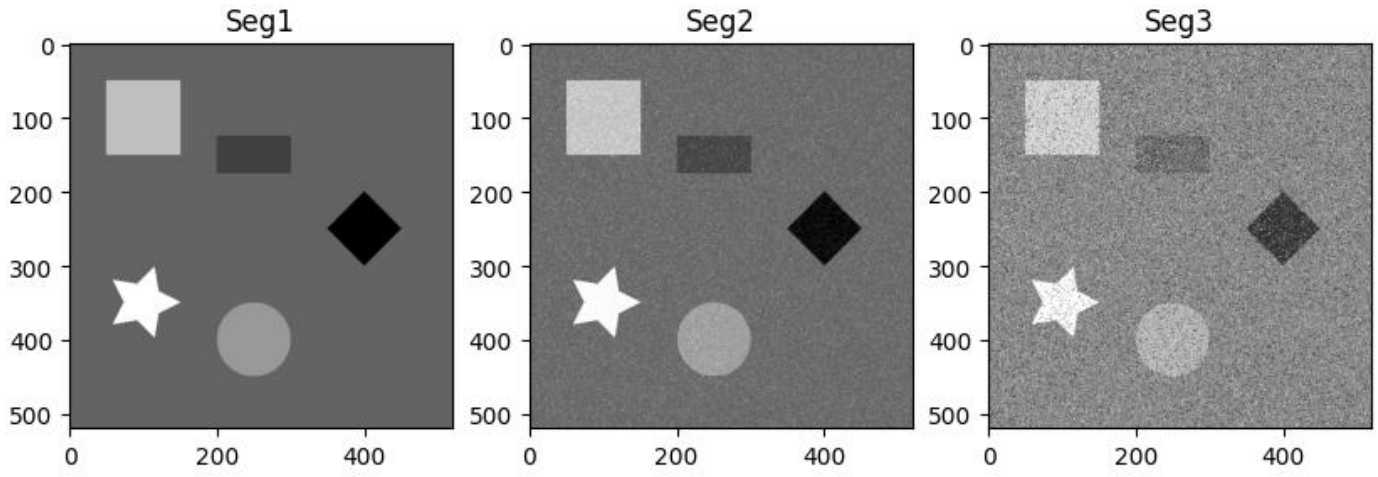
The function first copies the input image into a new variable named **output**. Then it generates a random matrix of the same size as the input image with values between 0 and 1 representing the probability of adding noise to each pixel.

Next, the function adds salt noise to the **output** variable by setting the pixel values to 0 where the corresponding value in the **prob_matrix** is less than **prob / 2**. Similarly, it adds pepper noise by setting the pixel values to 255 where the corresponding value in the **prob_matrix** is greater than **1 - prob / 2**.

Finally, the function returns the output image with added noise. Now I call this function like this:

```
impulse_img = add_salt_pepper_noise(noisy_img, 0.1)
```

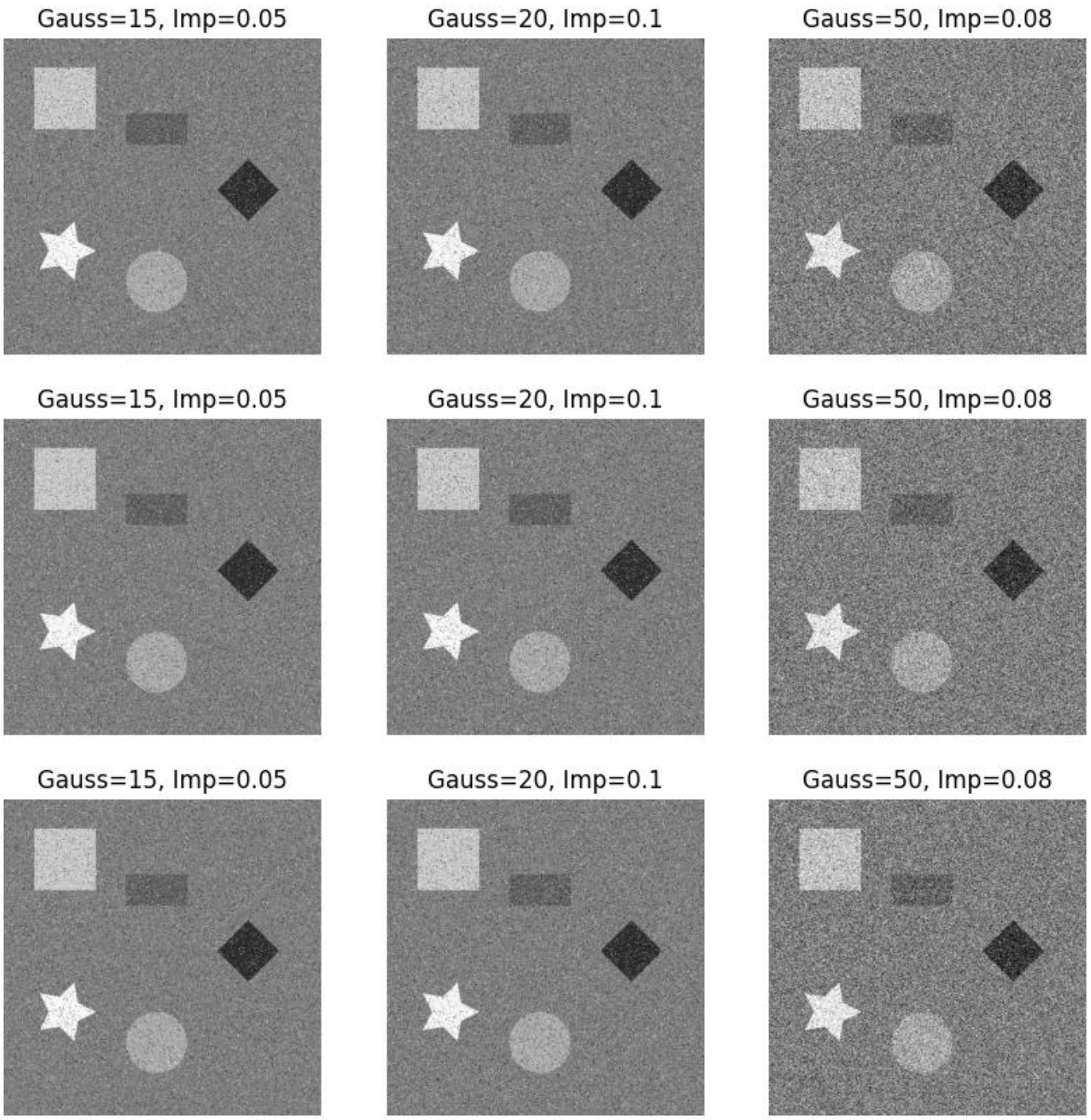
And the final output looks like this:



With different SD values of gaussian noise and different impulse noise proportions, we can generate different images. Here I use these three sets:

```
gaussian_noise_levels = [15, 20, 50]
impulse_noise_levels = [0.03, 0.04, 0.05]
```

And I get the following outputs.



2. a) **"Relative signed area error" function:**

Ans: For this, we first identify the largest connected components from the noisy image and find the area of those regions. Similarly, we do the same for ground truth file also. Now, based on the equation, we can calculate the error using the true area and computed area as follows:

$$A_{error} = \frac{\sum_{i=1}^{N} T_i - \sum_{j=1}^{M} A_j}{\sum_{i=1}^{N} T_i} \times 100$$

where Ti is the true area of the i-th object and Aj is the measured area of the j-th object, N is the number of objects in the image, M is the number of objects after segmentation. Areas may be expressed in pixels. Here is my implementation.

```
def relative_area_error(gt, out):
     # Get the areas of all the objects in output
    num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(out)
    areas = stats[:, cv2.CC_STAT_AREA]
    # Sort the areas in descending order and get the indices
    sorted_indices = np.argsort(areas)[::-1]
    # Get the areas and centroids of the top 5 objects
    area = 0
    num_objects = min(5, num_labels - 1)  # Ensure we don't go out of bounds
    for i in range(num_objects):
        index = sorted_indices[i + 1]  # Skip the background label (0)
        area += areas[index]


    # Get the areas of all the objects in gt
    num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(gt)
    areas = stats[:, cv2.CC_STAT_AREA]
    # Sort the areas in descending order and get the indices
    sorted_indices = np.argsort(areas)[::-1]
    # Get the areas and centroids of the top 5 objects
    area_gt = 0
    num_objects = min(5, num_labels - 1)  # Ensure we don't go out of bounds
    for i in range(num_objects):
        index = sorted_indices[i + 1]  # Skip the background label (0)
        area_gt += areas[index]

    # print(area_gt, area)
    error = (area_gt - area) * 100 /area_gt
    return error
```

This code defines a function **relative_area_error** that calculates the relative area error between the ground truth (**gt**) and the output (**out**). The function uses the **connectedComponentsWithStats** function from the **cv2** library to get the areas of all the objects in both the **gt** and **out** images.

It first calls **connectedComponentsWithStats** on the **out** image to get the areas of all the objects in it. It then sorts the areas in descending order and gets the areas and centroids of the top 5 objects (excluding the background label). It adds up the areas of these top 5 objects and stores the sum in the variable **area**.

It then repeats the same process for the **gt** image to get the ground truth areas and stores the sum in the variable **area_gt**.

Finally, the function calculates the relative area error by taking the difference between **area_gt** and **area**, dividing it by **area_gt**, multiplying it by 100, and returning the result.

2.b) **Labelling error**

Ans: L_error is defined as the ratio of the number of incorrectly labeled pixels (object pixels labeled as background as vice versa) and the number of pixels of true objects, and is expressed as percent. The implementation looks like this:

```
def labeling_error(gt, out):
    count = 0
    count_GT = 0
    H, W = out.shape[:2]
    for i in range (H):
      for j in range (W):
        if (gt[i][j] == 255 and out[i][j] != 255) or (gt[i][j] == 0 and out[i][j] != 0):
          count += 1
        if gt[i][j] == 255:
          count_GT += 1
    return (count / count_GT)*100
```

This code defines a function `labeling_error` that takes in two images `gt` and `out` as inputs and returns the labeling error between them. The labeling error is a measure of the difference between the binary labels in the ground truth image `gt` and the predicted image `out`.

The function first initializes two variables `count` and `count_GT` to zero. These variables will be used to count the number of pixels that are mislabeled and the total number of pixels in the ground truth image, respectively.

The function then iterates over each pixel in the images using a nested for loop. For each pixel, if the pixel in the ground truth image is white (255) and the corresponding pixel in the predicted image is not white or if the pixel in the ground truth image is black (0) and the corresponding pixel in the predicted image is not black, then the `count` variable is incremented. This counts the number of pixels that are mislabeled.

The `count_GT` variable is incremented for each white pixel in the ground truth image, which counts the total number of pixels in the ground truth image.

Finally, the function returns the percentage of pixels that are mislabeled by dividing `count` by `count_GT` and multiplying by 100.

3) **Basic Thresholding and Chane-Vase method**

Ans) This is a bit tricky. Some of the regions in the image has lower intensity than the background and some has greater than the background. So, to perform thresholding, we need to perform thresholding twice. As the background value is set to 127, we perform thresholding once based on threshold value 128, and then perform inverse thresholding below threshold value 126. We add these two outputs to get the final output. This is the thresholding code.

```
def threshold_image(img):
    thresh_below = cv2.threshold(img, 128, 255, cv2.THRESH_BINARY)[1]
    thresh_above = cv2.threshold(img, 126, 255, cv2.THRESH_BINARY_INV)[1]
    threshold = cv2.add(thresh_below, thresh_above)
    return threshold
```

For Chane-Vase, we define the following function:

```
def chanVeseSegmentation(origImage, lambda1=1, lambda2=1):
  grayscale_orig = origImage
  contour = np.zeros_like(grayscale_orig)
  rr1, cc1 = skimage.draw.disk((90,200),20)
  rr2, cc2 = skimage.draw.disk((120,110),13)
  rr3, cc3 = skimage.draw.disk((200,200), 13)
  contour[rr1, cc1] = 1
  contour[rr2, cc2] = 1
  contour[rr3, cc3] = 1
  cv = chan_vese(grayscale_orig, mu=1, lambda1=lambda1, lambda2=lambda2, tol=1e-3,
              max_num_iter=1000, dt=0.5, init_level_set=contour,
              extended_output=True)
  return cv[0]
```

This code defines a function **chanVeseSegmentation** that implements the Chan-Vese segmentation algorithm using the **chan_vese** function from the scikit-image library. The Chan-Vese segmentation algorithm is a method for image segmentation that is based on the level set method, where the boundaries of the objects in the image are represented as level sets of a scalar function.

The function takes an input image **origImage** and two optional parameters **lambda1** and **lambda2**. The **lambda1** and **lambda2** parameters control the trade-off between the two terms in the Chan-Vese functional, which corresponds to the inside and outside regions of the segmented objects, respectively. The default values for **lambda1** and **lambda2** are both 1.

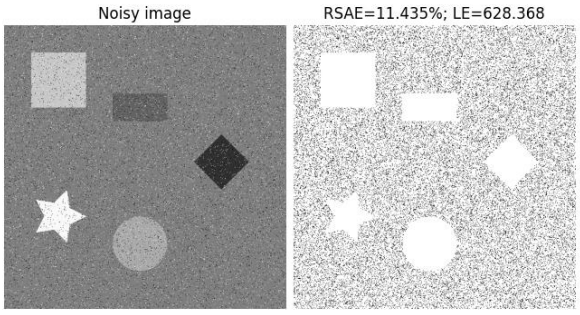
The first step of the function is to convert the input image to grayscale and create an initial contour for the segmentation using three circular regions defined by the **skimage.draw.disk** function.

The **chan_vese** function is then called with the input image, the initial contour, and the specified parameters. The output of the function is a tuple containing the final segmentation mask and other information about the algorithm's convergence.

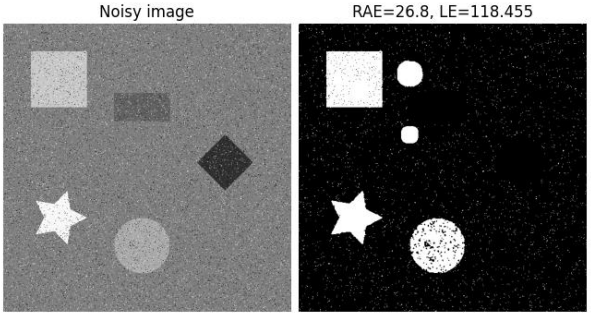
Finally, the function returns the final segmentation mask as the output.

Here are the outputs from these two methods:

BASIC THRESHOLDING

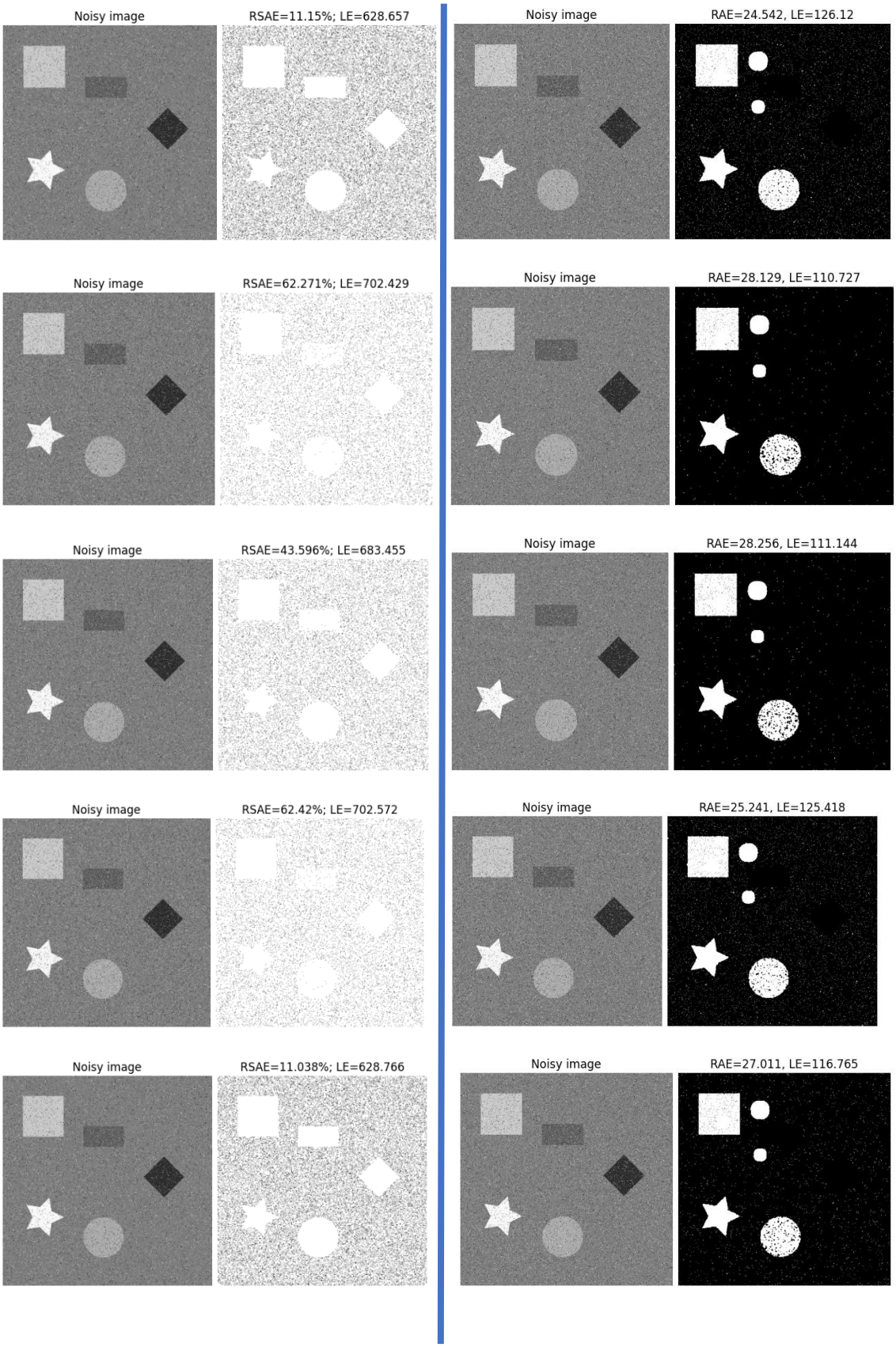


CHANE_VASE

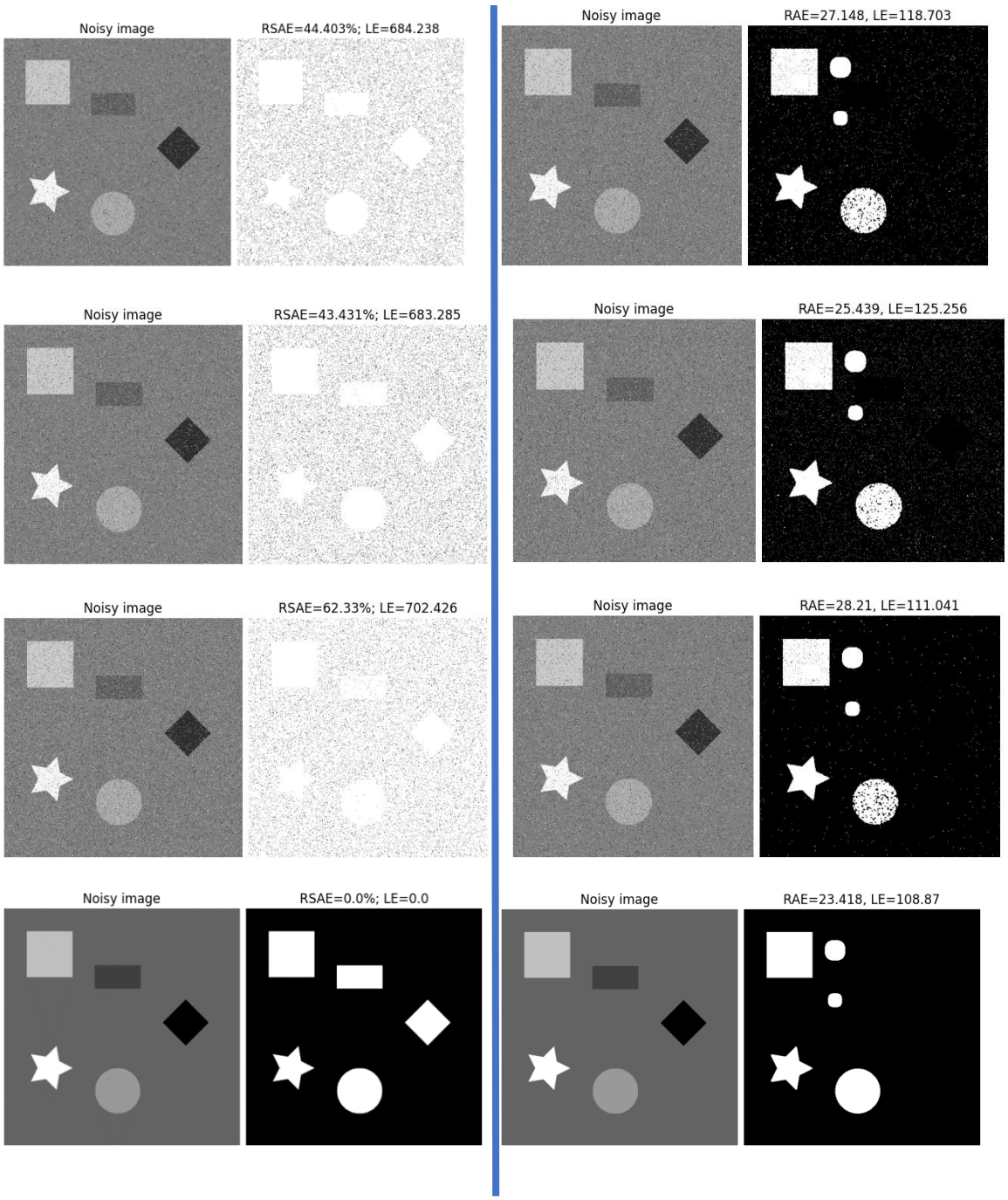


BASIC THRESHOLDING

CHANE_VASE



BASIC THRESHOLDING **CHANE_VASE**



So, here are the qualitative and quantitative comparison of both the methods. In most of the cases, Chane Vase performs well, although it consistently misses some region (rectangle or square) in the image.

4. **Image Registration**

Ans: First, I rotate the image 15 degree anticlockwise using the following code:

```
import math

img = cv2.imread('/content/drive/MyDrive/Medical Imaging Final/heart.jpg', 0)
h, w = img.shape
_, axs = plt.subplots(1, 2, layout='constrained')
```

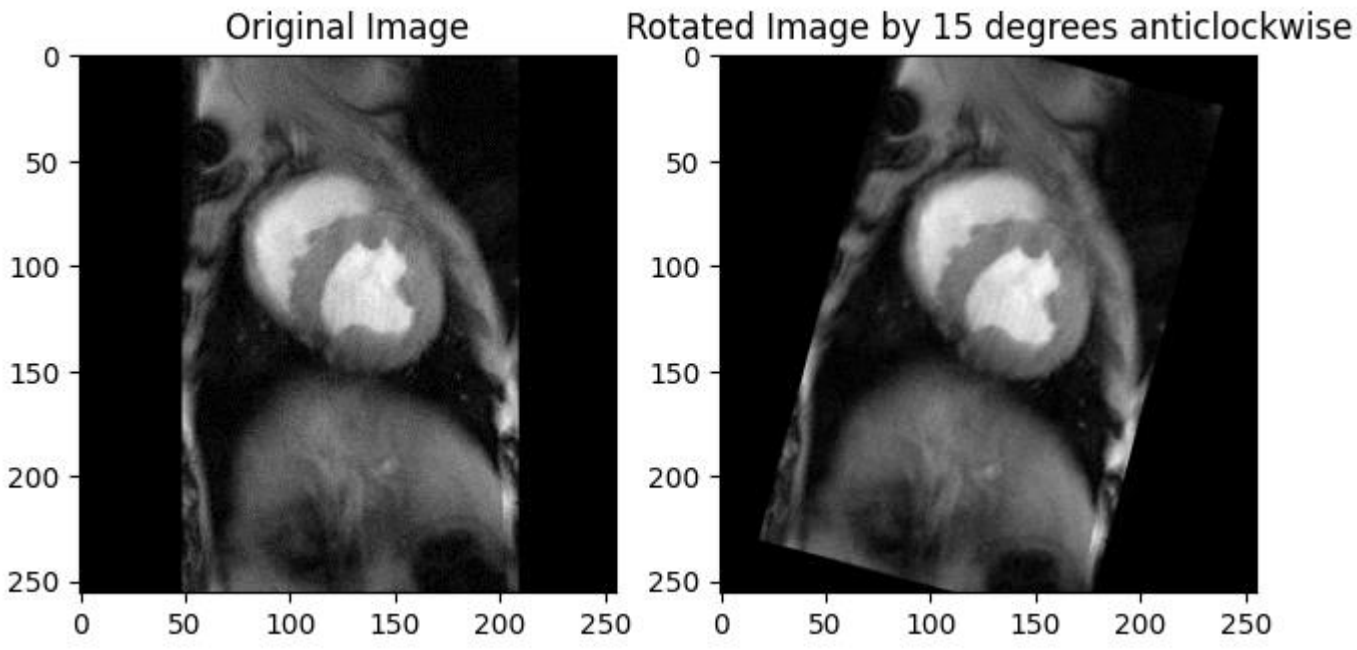
```
axs[0].imshow(img, cmap='gray')
axs[0].set_title('Original Image')
# plt.show()

angle = -15.0
rotation_matrix = cv2.getRotationMatrix2D((h/2, w/2), angle, 1)

rotated_img = cv2.warpAffine(img, rotation_matrix, (w, h))
axs[1].imshow(rotated_img, cmap='gray')
axs[1].set_title('Rotated Image by 15 degrees anticlockwise')
plt.show()

cv2.imwrite('heart.15.jpg', rotated_img)
```

This code loads an image of a heart in grayscale using OpenCV and displays it using Matplotlib. It then applies a rotation of 15 degrees in the anticlockwise direction to the image using **cv2.getRotationMatrix2D** to obtain a rotated image. The rotated image is also displayed using Matplotlib. Finally, the rotated image is saved to the file **heart.15.jpg** using **cv2.imwrite**.

cv2.warpAffine() is a function in OpenCV that applies an affine transformation to an image. In the provided code, it takes in the original image **img** and the rotation matrix **rotation_matrix**, and returns a new image that is the result of rotating the original image by the specified angle.

The **(w, h)** argument specifies the dimensions of the output image after the rotation. The resulting image will have the same size as the original image, but some parts of the image may be cut off or become black due to the rotation.

After rotation, the image looks like this:



Then, we try to perform registration using SIFT operation. SIFT (Scale-Invariant Feature Transform) is a computer vision algorithm that is used to detect and describe local features in images. It is robust to changes in scale, rotation, illumination, and affine distortion. SIFT works by first detecting scale-space extrema, which are points where the difference of Gaussian function takes a maximum or minimum value over a scale-space pyramid. Then, keypoints are selected at these extrema based on their stability and repeatability across multiple images. Each keypoint is described by a feature vector that encodes information about its orientation and local image structure.

SIFT can be used for image registration by first detecting SIFT features in both the reference image and the image to be registered. Then, a matching algorithm is used to match the SIFT features between the two images. In the code above, the Brute-Force Matcher is used to match the SIFT features between the two images. The matched keypoints are used to estimate a homography matrix that describes the transformation between the two images. Finally, the warpPerspective function is used to warp the reference image using the homography matrix to obtain an aligned image. This aligned image can be used for further analysis, such as image fusion or object detection.

Here is the code:

```
# Load the images
img1 = cv2.imread('/content/drive/MyDrive/Medical Imaging Final/heart.jpg', 0)
img2 = cv2.imread('/content/drive/MyDrive/Medical Imaging Final/Output/heart.15.jpg', 0)

# Create a SIFT detector object
sift = cv2.SIFT_create()

# Find keypoints and descriptors in the images
kp1, des1 = sift.detectAndCompute(img1, None)
kp2, des2 = sift.detectAndCompute(img2, None)

# Create a Brute-Force Matcher object
bf = cv2.BFMatcher()

# Match keypoints in the two images
matches = bf.knnMatch(des1, des2, k=2)

# Apply ratio test to filter out poor matches
good_matches = []
for m, n in matches:
    if m.distance < 0.75 * n.distance:
        good_matches.append(m)

# Get the coordinates of the matched keypoints in each image
src_pts = np.float32([kp1[m.queryIdx].pt for m in good_matches]).reshape(-1, 1, 2)
dst_pts = np.float32([kp2[m.trainIdx].pt for m in good_matches]).reshape(-1, 1, 2)

# Find the transformation matrix using RANSAC algorithm
M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

# Warp the source image using the transformation matrix
aligned_img = cv2.warpPerspective(img1, M, (img1.shape[1], img1.shape[0]))

# Show the aligned image
```

```
plt.imshow(aligned_img, 'gray')
plt.title('Aligned Image')
plt.axis('off')
plt.show()
```

The code first loads two grayscale images, img1 and img2. Then, a SIFT detector object is created using cv2.SIFT_create(). The detectAndCompute() method is then used to detect keypoints and compute their descriptors for both images.

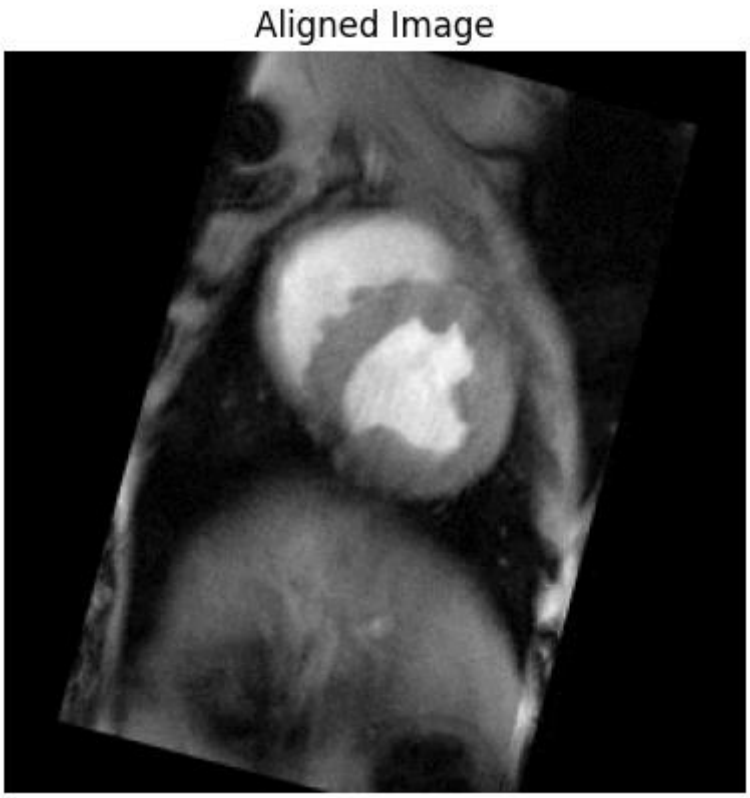
Next, a Brute-Force Matcher object is created using cv2.BFMatcher(), and the knnMatch() method is called to match keypoints in the two images. The ratio test is then applied to filter out poor matches. The matches that pass the ratio test are stored in the good_matches list.

The coordinates of the matched keypoints in each image are then extracted from the good_matches list and stored in the src_pts and dst_pts arrays, respectively.

The findHomography() method is then used to find the transformation matrix M between the two images using RANSAC algorithm. The RANSAC algorithm helps to eliminate outliers that may exist in the matches.

Finally, the warpPerspective() method is used to apply the transformation matrix M to the source image (img1) to align it with the target image (img2). The aligned image is then displayed using the imshow() method.

Here is the output.



5. Heat Equation

Ans: The linear heat equation is a partial differential equation that describes the process of heat diffusion in a medium. In image processing, it can be used as a technique for smoothing an image, which means reducing the amount of noise in the image and making it appear less jagged.

The equation takes the form:

∂u/∂t = D(∂^2u/∂x^2 + ∂^2u/∂y^2)

where u(x,y,t) is the temperature at point (x,y) at time t, and D is the diffusion coefficient. The second derivatives with respect to x and y represent the Laplacian operator, which is a measure of the rate of change of the temperature at a point.

To use the linear heat equation for smoothing an image, we can treat the image as a 2D temperature distribution and apply the equation iteratively to update the temperature distribution over time. The Laplacian operator effectively acts as a filter, smoothing out sharp edges and reducing the high-frequency components of the image. The diffusion coefficient D and the time step size dt can be adjusted to control the amount of smoothing applied and the speed at which it occurs.

Overall, the linear heat equation can be a useful tool for image smoothing, particularly when dealing with noisy or jagged images. Here is the code:

```
# Load the image
img = cv2.imread('/content/drive/MyDrive/Medical Imaging Final/heart.jpg',
0).astype(np.float64)

# diffusion coefficient
D = 0.1

# time step
dt = 0.1

show_iters = [10,100,1000]

x_second_derivative = np.array([[1,-2,1]])
y_second_derivative = x_second_derivative.T

fig, axs = plt.subplots(1,4, figsize=(15, 6))
axs[0].imshow(img, cmap='gray')
axs[0].set_title('Original Image')
subplot = 1

# Apply the heat equation for 1001 iterations
for n in range(1001):

    ## Second derivative of image in the x-direction
    img_xx = cv2.filter2D(src=img, ddepth=-1, kernel=x_second_derivative)

    ## Second derivative of image in the y-direction
    img_yy = cv2.filter2D(src=img, ddepth=-1, kernel=y_second_derivative)

    # Apply the discretized heat equation
    img += D * dt * (img_xx*2 + img_yy*2)
    if n in show_iters:
```

```
    axs[subplot].imshow(img, cmap='gray')
    axs[subplot].set_title('Image after ' + str(n) + ' iterations')
    subplot += 1
```

This code performs image smoothing using the heat equation. The heat equation is a partial differential equation that describes the diffusion of heat or other quantities in a medium. In the context of image processing, it is used to smooth images by removing noise and small details, while preserving large-scale features such as edges and boundaries.

The code loads an image and converts it to a floating-point format for numerical stability. The diffusion coefficient (D) and time step (dt) are set to 0.1. The second derivative of the image is calculated in the x and y directions using the Sobel operator (x_second_derivative and y_second_derivative). The heat equation is then applied for 1001 iterations using the discretized form:

img += D * dt * (img_xx*2 + img_yy*2)

Here, img_xx and img_yy are the second derivatives of the image in the x and y directions, respectively. The result of each iteration is an image that is smoother than the previous iteration. The final result after 1001 iterations is displayed, as well as intermediate results after 10, 100, and 1000 iterations.

Image smoothing using the heat equation is used to improve the quality of images by removing noise and small details, while preserving the larger structures and edges. This technique is particularly useful in medical imaging and computer vision applications where images are often noisy and may contain unwanted artifacts.

Here are the outputs after multiple iterations: