Import

]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal
import skimage.io
from PIL import Image
from sklearn.metrics import roc_curve, auc, confusion_matrix
from skimage.filters import threshold_otsu
```

Task 1: Laplacian of Gaussian

]:
```
def create_log_mask(size, sigma):# Function to create a Laplacian of Gaussian (LoG) mask
    # size: mask size, sigma: standard deviation for Gaussian
    n = size // 2
    y, x = np.ogrid[-n:n+1, -n:n+1]
    gaussian_exp = (x**2 + y**2) / (2 * sigma**2)
    log = -1 / (np.pi * sigma**4) * (1 - gaussian_exp) * np.exp(-gaussian_exp)
    log -= log.mean()# normalize the mask
    return log
def create_gaussian_kernel0(size, sigma):# Function to create a Gaussian kernel
    # size: kernel size, sigma: standard deviation for Gaussian
    n = size // 2
    y, x = np.ogrid[-n:n+1, -n:n+1]
    gaussian_kernel = np.exp(-(x**2 + y**2) / (2 * sigma**2))
    gaussian_kernel /= gaussian_kernel.sum() # normalize the kernel
    return gaussian_kernel
def apply_log_filter_and_display(image_path, mask_size=7, sigma=1.0, gaussian_blur=False, gaussian_sigma=3,␣
↪save_path='filtered_image.png'):# Function to apply LoG filter and display the results
    log_mask = create_log_mask(mask_size, sigma)# Create an LoG mask
    image = skimage.io.imread(image_path)
    if len(image.shape) == 3 and image.shape[2] == 3:# Convert to grayscale if the image is colored
        gray_image = skimage.color.rgb2gray(image)
    else:
        gray_image = np.squeeze(image)
    if gaussian_blur:    # Apply Gaussian blur if enabled
        gaussian_kernel = create_gaussian_kernel0(mask_size, gaussian_sigma)
        processed_image = scipy.signal.convolve2d(gray_image, gaussian_kernel, mode='same', boundary='symm')
    else:
        processed_image = gray_image
    log_image = scipy.signal.convolve2d(processed_image, log_mask, mode='same', boundary='symm')# Apply the␣
↪LoG mask to the processed image using convolution
    log_image_normalized = (log_image - log_image.min()) / (log_image.max() - log_image.min())# Normalize the␣
↪LoG filtered image for display
    log_image_mapped = (log_image_normalized * 255).astype(np.uint8)
    thresh = threshold_otsu(log_image_normalized)# Calculate Otsu's threshold and create binary image
    log_bool_image = log_image_normalized > thresh# Convert the LoG filtered image to boolean based on the␣
↪threshold
    fig, axes = plt.subplots(1, 3, figsize=(18, 6))
    ax = axes.ravel()
    ax[0].imshow(gray_image, cmap='gray')
    ax[0].set_title('Original Image')
    ax[0].axis('off')
    ax[1].imshow(log_image_mapped, cmap='gray')
    ax[1].set_title('LoG Filtered Image')
    ax[1].axis('off')
    ax[2].imshow(log_bool_image, cmap='gray')
    ax[2].set_title('LoG Filtered (Bool) Image')
    ax[2].axis('off')
    plt.tight_layout()
    plt.show()
apply_log_filter_and_display('shakey.150.gif', gaussian_blur=True, save_path='Task1_image/
↪shakey_log_with_gaussian_bool.png')
```

```
apply_log_filter_and_display('shakey.150.gif', gaussian_blur=False, save_path='Task1_image/
↪shakey_log_without_gaussian_bool.png')
```

Task 2: Edge Detection

```
]: def load_image(path):
       img = Image.open(path).convert('L')
       img = np.array(img, dtype=np.float32) / 255.0
       return img
   image1, grund_truth1 = load_image('cells/9343 AM.bmp'), load_image('cells/9343 AM Edges.bmp')
   image2, grund_truth2 = load_image('cells/10905 JL.bmp'), load_image('cells/10905 JL Edges.bmp')
   image3, grund_truth3 = load_image('cells/43590 AM.bmp'), load_image('cells/43590 AM Edges.bmp')
   # Roberts
   def roberts_edge_detection(image):
       # Define Roberts Cross kernels for vertical and horizontal edges
       roberts_cross_v = np.array([[1, 0], [0, -1]])
       roberts_cross_h = np.array([[0, -1], [1, 0]])
       vertical_edges = scipy.signal.convolve2d(image, roberts_cross_v, mode='same')
       horizontal_edges = scipy.signal.convolve2d(image, roberts_cross_h, mode='same')
       return np.sqrt(np.square(horizontal_edges) + np.square(vertical_edges))    # Combine the edges
   #Sobel
   def sobel_edge_detection(image):
       # Define Sobel kernels for vertical and horizontal edges
       sobel_v = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
       sobel_h = np.array([[1, 2, 1], [0, 0, 0], [-1,-2,-1]])
       vertical_edges = scipy.signal.convolve2d(image, sobel_v, mode='same')
       horizontal_edges = scipy.signal.convolve2d(image, sobel_h, mode='same')
       return np.sqrt(np.square(horizontal_edges) + np.square(vertical_edges))
   #First order Gaussian
   def compute_gaussian(sigma, mean, vec):
       return 1 / np.sqrt(2 * np.pi * sigma ** 2) * np.exp(- (vec - mean) ** 2 / (2 * sigma ** 2))# Gaussian␣
   ↪function: used to create a Gaussian kernel:
   def calculate_first_order_gaussian(sigma, mean, vec):
       return - ((vec - mean) / sigma ** 2) * compute_gaussian(sigma, mean, vec) # Derivative of Gaussian: used␣
   ↪for finding gradient magnitude in the image
   def generate_gaussian_kernel(sigma, mean, size):
       vec = np.arange(-size // 2, size // 2 + 1, dtype=np.float32) # Generate a one-dimensional Gaussian kernel
       return calculate_first_order_gaussian(sigma, mean, vec)
   def apply_gaussian_filter_to_image(image, sigma=1, mean=0, size=7):
       gaussian_kernel = generate_gaussian_kernel(sigma, mean, size)# Create Gaussian kernels and apply them to␣
   ↪the image
       # Convolve image with Gaussian kernels along x and y directions
       edge_x = scipy.signal.convolve2d(image, gaussian_kernel[None, :], mode='same')
       edge_y = scipy.signal.convolve2d(image, gaussian_kernel[:, None], mode='same')
       gradient_magnitude = np.sqrt(edge_x**2 + edge_y**2)# Calculate gradient magnitude (Euclidean norm)
       return gradient_magnitude
   #Laplacian
   def laplacian_edge_detection(image):
       laplacian_kernel = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])# Laplacian kernel: highlights regions␣
   ↪where the gradient of image has large changes
       return scipy.signal.convolve2d(image, laplacian_kernel, mode='same')
   #LoG
   def log_edge_detection(image, sigma):
       log_kernel = create_log_mask(7, sigma) # Create LoG (Laplacian of Gaussian) kernel
       return scipy.signal.convolve2d(image, log_kernel, mode='same')
   #Gaussian smoothing
   def create_gaussian_kernel(size, sigma):
       n = size // 2
       y, x = np.ogrid[-n:n+1, -n:n+1]
       gaussian_kernel = np.exp(-(x**2 + y**2) / (2 * sigma**2))
       gaussian_kernel /= gaussian_kernel.sum()
       return gaussian_kernel
   def apply_gaussian_blur(image, gaussian_kernel):
       return scipy.signal.convolve2d(image, gaussian_kernel, mode='same')# Apply Gaussian blur to an image
   #Gaussian + Sobel
   def sobel_edge_detection_gaussian(image, kernel_size, sigma):
       gaussian_kernel = create_gaussian_kernel(kernel_size, sigma)
       blurred_image = apply_gaussian_blur(image, gaussian_kernel)
       return sobel_edge_detection(blurred_image)
   #Gaussian + Roberts
   def roberts_edge_detection_gaussian(image, kernel_size, sigma):
       gaussian_kernel = create_gaussian_kernel(kernel_size, sigma)
       blurred_image = apply_gaussian_blur(image, gaussian_kernel)
       return roberts_edge_detection(blurred_image)
   #Gaussian + First order Gaussian
   def gaussian_filter_first_order_Gaussian(image, blur_sigma=1, blur_size=7, edge_sigma=1, edge_mean=0,␣
    ↪edge_size=7):
       gaussian_blur_kernel = create_gaussian_kernel(blur_size, blur_sigma)
       blurred_image = apply_gaussian_blur(image, gaussian_blur_kernel)
       edges = apply_gaussian_filter_to_image(blurred_image, edge_sigma, edge_mean, edge_size)
       return edges
   #Gaussian + Laplacian
   def laplacian_edge_detection_gaussian(image, kernel_size, sigma):
       gaussian_kernel = create_gaussian_kernel(kernel_size, sigma)
       blurred_image = apply_gaussian_blur(image, gaussian_kernel)
       return laplacian_edge_detection(blurred_image)
   #Gaussian + LOG
   def LOG_Gaussian(image, sigma):
       gaussian_kernel = create_gaussian_kernel(9, sigma)
       blurred_image = apply_gaussian_blur(image, gaussian_kernel)
       log_kernel = create_log_mask(9, sigma)
       return scipy.signal.convolve2d(blurred_image, log_kernel, mode='same')
   # Apply otsu's thresholding to the image
   def apply_otsu_threshold(image):
       thresh = threshold_otsu(image)
       binary_image = image > thresh
       return binary_image
   image1_otsu = apply_otsu_threshold(image1)
   image2_otsu = apply_otsu_threshold(image2)
   image3_otsu = apply_otsu_threshold(image3)
   # Functions to normalize and invert images
   def normalize(image):
       min_val, max_val = np.min(image), np.max(image)
       return (image - min_val) / (max_val - min_val)
   def invert_image(image):
       return 1 - normalize(image)
   # Zero-crossing method
   def zero_cross(image):
       z_c_image = np.zeros(image.shape)
       thresh = np.absolute(image).mean() * 0.75
       h,w = image.shape
       for y in range(1, h - 1):
           for x in range(1, w - 1):
               patch = image[y-1:y+2, x-1:x+2]
               p = image[y, x]
               maxP = patch.max()
               minP = patch.min()
               if (p > 0):
                   zeroCross = True if minP < 0 else False
               else:
                   zeroCross = True if maxP > 0 else False
               if ((maxP - minP) > thresh) and zeroCross:
                   z_c_image[y, x] = 1
       return z_c_image
   def process_and_display_all(images, titles, save_figure=True, fi='Task2_image/edge_detection_all.png'):
       num_methods = 5
       plt.figure(figsize=(20, 9))
       for j, image in enumerate(images):
           edge_gaussian_roberts = roberts_edge_detection_gaussian(image, kernel_size=3, sigma=1)
           edge_gaussian_sobel = sobel_edge_detection_gaussian(image, kernel_size=3, sigma=1)
           edge_gaussian_first_order_gaussian = gaussian_filter_first_order_Gaussian(image, blur_sigma=1,␣
    ↪blur_size=7, edge_sigma=1, edge_mean=0, edge_size=7)
           edge_gaussian_laplacian = (laplacian_edge_detection_gaussian(image, kernel_size=9, sigma=3))
           edge_gaussian_log = (LOG_Gaussian(image, sigma=1))
           edges =␣
    ↪[invert_image(edge_gaussian_roberts),invert_image(edge_gaussian_sobel),invert_image(edge_gaussian_first_order_ga
    ↪invert_image(edge_gaussian_log)]
           for i, img in enumerate(edges, 1):
               plt.subplot(3, num_methods, j * num_methods + i)
               plt.imshow(img, cmap='gray')
               if j == 0:
                   plt.title(titles[i - 1])
               plt.axis('off')
       plt.tight_layout()
       if save_figure:
           plt.savefig(fi, dpi=600)
       plt.show()
   image_list = [image1, image2, image3]
   titles = ['Roberts', 'Sobel', 'First Order Gaussian', 'Laplacian', 'LoG']
   process_and_display_all(image_list, titles)
```

Task 3: Advanced Edge Detection

]:
```
def canny_edge_detection(image, sigma = 1,  kernel_size= 9, high=0.90, low=0.50):
    # Create a Gaussian kernel
    n =  kernel_size // 2
    y, x = np.ogrid[-n:n+1, -n:n+1]
    G = 1 / (2 * np.pi * sigma**2) * np.exp(-(x**2 + y**2) / (2 * sigma**2))
    # Compute gradients using the Gaussian kernel
    Gx = (-x / sigma**2) * G # Gradient in X direction
    Gy = (-y / sigma**2) * G # Gradient in Y direction
    fx = scipy.signal.convolve(image, Gx, mode='same')# Convolve with Gx
    fy = scipy.signal.convolve(image, Gy, mode='same')# Convolve with Gy
    magnitude = np.abs(fx) + np.abs(fy)# Magnitude of the gradient
    direction = np.arctan2(fx, fy)# Direction of the gradient
    M, N = magnitude.shape
    canny_image = np.zeros((M,N), dtype=np.float32)
    # Normalize direction angles
    angle = direction * (180. / np.pi)
    angle[angle < 0] += 180
    # Non-maximum suppression: compares pixel to neighbors along the gradient direction
    for i in range(1, M-1):
        for j in range(1, N-1):
            try:
                q = 255
                r = 255
                if (0 <= angle[i,j] < 22.5) or (157.5 <= angle[i,j] <= 180):
                    q = magnitude[i, j+1]
                    r = magnitude[i, j-1]
                elif (22.5 <= angle[i,j] < 67.5):
                    q = magnitude[i+1, j-1]
                    r = magnitude[i-1, j+1]
                elif (67.5 <= angle[i,j] < 112.5):
                    q = magnitude[i+1, j]
                    r = magnitude[i-1, j]
                elif (112.5 <= angle[i,j] < 157.5):
                    q = magnitude[i-1, j-1]
                    r = magnitude[i+1, j+1]
                if (magnitude[i,j] >= q) and (magnitude[i,j] >= r):
                    canny_image[i,j] = magnitude[i,j]
                else:
                    canny_image[i,j] = 255
            except IndexError as e:
                pass
    # Double threshold: determine strong and weak edges
    high_threshold = np.percentile(magnitude, high * 100)
    low_threshold = high_threshold * low
    strong_i, strong_j = np.where(magnitude >= high_threshold)
    zeros_i, zeros_j = np.where(magnitude < low_threshold)
    weak_i, weak_j = np.where((magnitude <= high_threshold) & (magnitude >= low_threshold))
    # Set strong edges to white and non-edges to black
    canny_image[strong_i, strong_j] = 255
    canny_image[zeros_i, zeros_j] = 0
    # Linking weak edges to strong edges (hysteresis)
    for i, j in zip(weak_i, weak_j):
        if 0 in (canny_image[i-1:i+2, j-1:j+2]):
            canny_image[i, j] = 255
        else:
            canny_image[i, j] = 0
    return canny_image
canny_edges = canny_edge_detection(image1_otsu, sigma=1, kernel_size=9, high=0.90, low=0.50)
canny_edges2 = canny_edge_detection(image2_otsu, sigma=1, kernel_size=9, high=0.90, low=0.50)
canny_edges3 = canny_edge_detection(image3_otsu, sigma=1, kernel_size=9, high=0.90, low=0.50)
fig, axes = plt.subplots(1, 3, figsize=(12, 6))
ax = axes.ravel()
ax[0].imshow(invert_image(canny_edges), cmap='gray')
ax[0].set_title('9343 Canny Edge Detection')
ax[0].axis('off')
ax[1].imshow(invert_image(canny_edges2), cmap='gray')
ax[1].set_title('10905 Canny Edge Detection')
ax[1].axis('off')
ax[2].imshow(invert_image(canny_edges3), cmap='gray')
ax[2].set_title('43590 Canny Edge Detection')
ax[2].axis('off')
plt.tight_layout()
plt.show()
```

Task 4: Result evaluation

```
]: # Function to calculate sensitivity and specificity
def calculate_sensitivity_specificity(test_image, ground_truth):
    tn, fp, fn, tp = confusion_matrix(ground_truth.flatten(), test_image.flatten(), labels=[0, 1]).ravel() #␣
 ↪Confusion matrix elements: true negative, false positive, false negative, true positive
    sensitivity = tp / (tp + fn) if tp + fn != 0 else 0
    specificity = tn / (tn + fp) if tn + fp != 0 else 0
    return sensitivity, specificity
# Function to calculate F1 score
def calculate_f1_score(sensitivity, specificity, tp, fp):
    precision = tp / (tp + fp) if (tp + fp) != 0 else 0
    recall = sensitivity  # recall is the same as sensitivity
    f1_score = 2 * (precision * recall) / (precision + recall) if (precision + recall) != 0 else 0
    return f1_score
def invert_image_otsu(image):
    return 1 - image
# Function to plot ROC curves and calculate statistics for various edge detection methods
def plot_all_roc_curves(edges, ground_truth, titles):
    plt.figure(figsize=(10, 10))
    binarized_images = []
    for edge, title in zip(edges, titles):
        # Binarize images based on method
        if title == 'Canny':# Canny's output ranges between 0 and 255; binarize it
            binarized_edge = (edge > 0).astype(int)
        elif title in ['LoG', 'Laplacian', 'Gaussian Laplacian', 'Gaussian LoG', 'Otsu']: # Outputs of these␣
↪methods are already binarized
            binarized_edge = edge
        else:# For other methods, find optimal threshold using ROC curve
            fpr, tpr, thresholds = roc_curve(ground_truth.flatten(), edge.flatten())
            roc_auc = auc(fpr, tpr)
            plt.plot(fpr, tpr, lw=2, label=f'{title} (AUC = {roc_auc:.2f})')
            optimal_idx = np.argmax(tpr - fpr)
            optimal_threshold = thresholds[optimal_idx]
            binarized_edge = (edge > optimal_threshold).astype(int)
        sensitivity, specificity = calculate_sensitivity_specificity(binarized_edge, ground_truth.flatten())#␣
↪Calculate sensitivity and specificity
        tn, fp, fn, tp = confusion_matrix(ground_truth.flatten(), binarized_edge.flatten()).ravel()
        f1 = calculate_f1_score(sensitivity, specificity, tp, fp)
        print(f'{title}- Threshold: {optimal_threshold:.2f}, Sensitivity: {sensitivity:.2f}, Specificity:␣
↪{specificity:.2f}, F1 Score: {f1:.2f}')
        if title in ['LoG', 'Laplacian', 'Gaussian Laplacian', 'Gaussian LoG', 'Canny']:# For these methods,␣
↪plot simplified ROC curve
            fpr = 1 - specificity
            tpr = sensitivity
            auc_value = 1 - (fpr + (1 - tpr)) / 2
            plt.plot([0, fpr, 1], [0, tpr, 1], lw=2, label=f'{title} (AUC = {auc_value:.2f})')
        binarized_images.append(binarized_edge)
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic')
    plt.legend(loc="lower right")
    plt.savefig('roc_curve_image{}.png'.format(idx), dpi=600)
    plt.show()
    plt.figure(figsize=(20, 20))
    for i, img in enumerate(binarized_images, 1):
        plt.subplot(5, 5, i)
        plt.imshow(img, cmap='gray')
        plt.title(titles[i-1])
        plt.axis('off')
    plt.show()
images = [image1, image2, image3]
ground_truths = [grund_truth1, grund_truth2, grund_truth3]
titles = ['Roberts', 'Sobel', 'Gaussian', 'Laplacian', 'LoG', 'Canny','Gaussian Sobel', 'Gaussian Roberts',␣
↪'Gaussian First Order Gaussian', 'Gaussian Laplacian', 'Gaussian LoG']
for idx, image in enumerate(images):
    image_otsu = apply_otsu_threshold(image)
    edge_roberts = (roberts_edge_detection(image))
    edge_sobel = (sobel_edge_detection(image))
    edge_gaussian = (apply_gaussian_filter_to_image(image, sigma=1, mean=0, size=7))
    edge_zerocrossing_lap = zero_cross(laplacian_edge_detection(image_otsu))
    edge_zerocrossing_log = zero_cross(log_edge_detection(image_otsu, sigma=1))
    edge_canny = (canny_edge_detection(image_otsu, sigma=1, kernel_size=9, high=0.90, low=0.50))
    edge_gaussian_sobel = (sobel_edge_detection_gaussian(image, kernel_size=3, sigma=1))
    edge_gaussian_roberts = (roberts_edge_detection_gaussian(image, kernel_size=3, sigma=1))
    edge_gaussian_first_order_gaussian = (gaussian_filter_first_order_Gaussian(image, blur_sigma=1,␣
↪blur_size=7, edge_sigma=1, edge_mean=0, edge_size=7))
    edge_gaussian_laplacian = zero_cross(laplacian_edge_detection_gaussian(image_otsu, kernel_size=9,␣
↪sigma=3))
    edge_gaussian_log = zero_cross(LOG_Gaussian(image_otsu, sigma=1))
    edges = [invert_image(edge_roberts), invert_image(edge_sobel), invert_image(edge_gaussian),␣
↪invert_image_otsu(edge_zerocrossing_lap), invert_image_otsu(edge_zerocrossing_log),␣
↪invert_image(edge_canny), invert_image(edge_gaussian_sobel), invert_image(edge_gaussian_roberts),␣
↪invert_image(edge_gaussian_first_order_gaussian), invert_image_otsu(edge_gaussian_laplacian),␣
↪invert_image_otsu(edge_gaussian_log)]
    plot_all_roc_curves(edges, ground_truths[idx].flatten(), titles)
```