# Digital Image Processing

## Practical Project

Dr. Mohamed Berbar

Subscribe in it :-

Eng \Abderrahman Mohamed Abu Al-Hassan

ID \ 15210180

Eng \Abdullah Mohamed Khater

ID \15210193

## Firstly: -

## An explanation for the content:

1. **Low Pass Filter (LPF): The (apply_lpf) function applies a low pass filter to the input image. It creates a kernel of ones, which is then divided by the square of the kernel size to normalize it. This kernel is convolved with the input image using the (filter2D) function from OpenCV. The result is a smoothed version of the input image, where high-frequency components are suppressed.**

```
# Low Pass Filter (LPF)
def apply_lpf(image, kernel_size):
    kernel = np.ones((kernel_size, kernel_size), np.float32) / (kernel_size
** 2)
    return cv.filter2D(src=image, ddepth=-1,kernel= kernel)
```

2. **High Pass Filter (HPF): The (apply_hpf) function applies a high pass filter to the input image. It uses a kernel that emphasizes the center pixel and subtracts the surrounding pixels. This kernel is convolved with the input image using the (filter2D) function from OpenCV. The result is an image that highlights the edges and other high-frequency components.**

```
# High Pass Filter (HPF)
def apply_hpf(image):
    kernel = np.array([[-1, -1, -1], [-1, 9, -1], [-1, -1, -1]])
    return cv.filter2D(src=image, ddepth=-1, kernel=kernel)
```

3. **Mean Filter: The (apply_mean) function applies a mean filter (also known as a box filter) to the input image. It uses the (blur) function from OpenCV, which convolves the image with a normalized box filter. The result is a smoothed version of the input image.**

```
# Mean Filter
def apply_mean(image, kernel_size):
    return cv.blur(image, (kernel_size, kernel_size))
```

## 4. Median Filter:

The median filter replaces each pixel value with the median of the pixel values in a 5x5 neighborhood. It is particularly effective at removing salt-and-pepper noise while preserving edges.

```python
# Median Filter
def apply_median(image, kernel_size):
    # Ensure kernel size is odd
    if kernel_size % 2 == 0:
        kernel_size += 1  # Make it odd by adding 1

    # Apply median blur
    return cv.medianBlur(image, kernel_size)
```

## Edge Detection: -

### 1.Robert Edge Detector :-

The Robert edge detector is an early edge detection operator that uses a pair of 2x2 convolution kernels to detect edges by computing the gradient magnitude.

```python
# Roberts Edge Detection
def apply_roberts(image):
    image_gray = cv.cvtColor(np.array(image), cv.COLOR_RGB2GRAY)
    kernel_x = np.array([[2, 0], [0, -2]])
    kernel_y = np.array([[0, 2], [-2, 0]])
    image_x = cv.filter2D(image_gray, -1, kernel_x)
    image_y = cv.filter2D(image_gray, -1, kernel_y)
    return cv.addWeighted(image_x, 0.5, image_y, 0.5, 0)
```

### 2.Prewitt Edge Detector :-

The Prewitt edge detector uses convolution kernels to approximate the gradient of the image intensity function. This kernel detects vertical edges; a similar kernel can be used to detect horizontal edges.

```
# Prewitt Edge Detection
def apply_prewitt(image):
    image_gray = cv.cvtColor(np.array(image), cv.COLOR_RGB2GRAY)
    kernel_x = np.array([[-2, 0, 2], [-2, 0, 2], [-2, 0, 2]])
    kernel_y = np.array([[-2, -2, -2], [0, 0, 0], [2, 2, 2]])
    image_x = cv.filter2D(image_gray, -1, kernel_x)
    image_y = cv.filter2D(image_gray, -1, kernel_y)
    return cv.addWeighted(image_x, 0.5, image_y, 0.5, 0)
```

## 3.Sobel Edge Detector: -

**The Sobel edge detector computes the gradient of the image intensity at each pixel using convolution with Sobel kernels. The parameters 1, 0 indicate the use of the kernel for detecting horizontal gradients with a kernel size of 5.**

```
# Sobel Edge Detection
def apply_sobel(image):
    image_gray = cv.cvtColor(np.array(image), cv.COLOR_RGB2GRAY)
    image_x = cv.Sobel(image_gray, cv.CV_64F, 1, 0, ksize=3)
    image_y = cv.Sobel(image_gray, cv.CV_64F, 0, 1, ksize=3)
    sobel_image = np.sqrt(image_x ** 2 + image_y ** 2)
        # Normalize the gradient magnitude image
    sobel_image = cv.normalize(sobel_image, None, 0, 255, cv.NORM_MINMAX,
dtype=cv.CV_8U)
    sobel_image= cv.addWeighted(image_x,0.5,image_y,0.5,0)

    return sobel_image
```

## 1. Morphological Transformation :

### 1.1. Erosion:

Erosion is the morphological dual to dilation, and shrinks an object by removing a layer of pixels around its edges. In mathematical terms, the Erosion **A Θ B** results from combining two sets using the vector subtraction of set elements.

**Code Explanation:**

```python
def get_erosion(image,kernal_size=5):
    image_gray=cv.cvtColor(np.array(image),cv.COLOR_RGB2GRAY)
    kernel = np.ones((kernal_size, kernal_size), np.uint8)
    img_erosion = cv.erode(image_gray, kernel, iterations=1)
    return img_erosion
```

- **Converts the input image (PIL image) into a NumPy array , and Converts the RGB image to a grayscale image.**
- **Defining a (kernel size square) filled with ones.**
- **Applies the erosion operation to the grayscale image using the specified kernel. The iteration =1 argument means that the erosion will be applied only once.**

### 1.2. Dilation:

Dilation is a morphological transformation which essentially expands an object by adding a layer of pixels around its edges, and as a consequence of the process shrinks any holes within the object.

Code Explanation:

```python
def get_dilation(image,kernel_size):
    image_gray=cv.cvtColor(np.array(image),cv.COLOR_RGB2GRAY)
    kernel = np.ones((kernel_size, kernel_size), np.uint8)
    img_dilation = cv.dilate(image_gray, kernel, iterations=1)
    return img_dilation
```

- **Converts the input image (PIL image) into a NumPy array , and Converts the RGB image to a grayscale image.**
- **Defining a (kernel size square) filled with ones.**
- **Applies the dilation operation to the grayscale image using the specified kernel.**

### 1.3. Opening:

**Open = erosion then dilation**

**Opening** involves erosion followed by dilation in the outer surface (the foreground) of the image. All the above-said constraints for erosion and dilation applies here. It is a blend of the two prime methods. It is generally used to remove the noise in the image.

**Code Explanation:**

```
def get_open(image,kernel_size):

    image_gray=cv.cvtColor(np.array(image),cv.COLOR_RGB2GRAY)
    kernel = np.ones((kernel_size, kernel_size), np.uint8)
    open_image = cv.morphologyEx(image_gray, cv.MORPH_OPEN, kernel)
    return open_image
```

- **Converts the input image (PIL image) into a NumPy array , and Converts the RGB image to a grayscale image.**
- **Defining a (kernel size square) filled with ones.**
- **Applies the open operation to the grayscale image using the specified kernel.**

### 1.4. Closing:

Close = dilation then erosion

**Closing** involves dilation followed by erosion in the outer surface (the foreground) of the image. All the above-said constraints for erosion and dilation applies here. It is a blend of the two prime methods. It is generally used to remove the noise in the image.

**Code Explanation:**

```
def get_close(image,kernel_size):
    image_gray=cv.cvtColor(np.array(image),cv.COLOR_RGB2GRAY)
    kernel = np.ones((kernel_size, kernel_size), np.uint8)
    close_image = cv.morphologyEx(image_gray, cv.MORPH_CLOSE, kernel)
    return close_image
```

- **Converts the input image (PIL image) into a NumPy array , and Converts the RGB image to a grayscale image.**
- **Defining a (kernel size square) filled with ones.**
- **Applies the dilation operation to the grayscale image using the specified kernel.**

## 2. Segmentation :

### 2.1. Segmentation using region split and merge:

Split and merge segmentation is an image processing technique used to segment an image**. The image is successively split into quadrants based on a homogeneity criterion and similar regions are merged to create the segmented result.**

- In the region merging technique, we try to combine the regions that contain the single object and separate it from the background.. There are many regions merging techniques such as Watershed algorithm, Split and merge algorithm, etc**.**

## Code Explanation:

```
  def get_seg_split_and_merge(image):

# # Convert numpy array to PIL Image
# pil_image = Image.fromarray(image)
# Convert image to grayscale
gray = image.convert('L')

width, height = image.size
segmented_image = np.zeros((height, width), dtype=np.uint8)

# Define recursive function for region split and merge
def split_merge(x, y, w, h):
    # Split region into four quadrants
    if w * h > 100:
        half_w = w // 2
        half_h = h // 2
        split_merge(x, y, half_w, half_h)
        split_merge(x + half_w, y, half_w, half_h)
        split_merge(x, y + half_h, half_w, half_h)
        split_merge(x + half_w, y + half_h, half_w, half_h)
    else:
        # Merge region if homogeneous
        region_mean = np.mean(np.array(gray.crop((x, y, x+w, y+h))))
        segmented_image[y:y+h, x:x+w] = 255 if region_mean > 128 else 0

# Start region split and merge
split_merge(0, 0, width, height)

return segmented_image
```

- **Creates an empty (black) image of the same size as the input image**
- **In the recursive function :**
  - **Splitting**

- Checks if the area of the region is greater than 100 pixels. If so, it proceeds to split the region.
- **half_w = w // 2 and half_h = h // 2** - Calculates the half-width and half-height of the region.
- **split_merge(x, y, half_w, half_h)** - Recursively calls split_merge for the top-left quadrant.
- **split_merge(x + half_w, y, half_w, half_h)** - Recursively calls split_merge for the top-right quadrant.
- **split_merge(x, y + half_h, half_w, half_h**) - Recursively calls split_merge for the bottom-left quadrant.
- **split_merge(x + half_w, y + half_h, half_w, half_h)** - Recursively calls split_merge for the bottom-right quadrant.

- **Merging**
  - If the region is small enough (area <= 100 pixels), it proceeds to the merging step.
  - **region_mean = np.mean(np.array(gray.crop((x, y, x+w, y+h**)))) - Calculates the mean intensity of the grayscale values in the region. **gray.crop((x, y, x+w, y+h))** crops the grayscale image to the region, and np.mean computes the mean of the pixel values.
  - **segmented_image[y:y+h, x:x+w]** = 255 if region_mean > 128 else 0 - If the mean intensity of the region is greater than 128, it sets the region in the segmented_image to white (255), otherwise to black (0).

## 2.2. Thresholding:

Thresholding is the process of converting grey-level images to binary-images.

**Code Explanation:**

```
    def get_seg_threshold(image, value):

  image_gray=cv.cvtColor(np.array(image),cv.COLOR_RGB2GRAY)
  _, threshold_image = cv.threshold(image_gray, value, 255, cv.THRESH_BINARY)
  return cv.cvtColor(threshold_image, cv.COLOR_GRAY2RGB)
```

- **Converts the input image (PIL image) into a NumPy array , and Converts the RGB image to a grayscale image.**
- **Defining a (kernel size square) filled with ones.**
- **Applies a binary threshold to the grayscale image using threshold function. The value parameter is the threshold value. Pixels with intensity greater than this value will be set to 255 (white), and pixels with intensity less than or equal to this value will be set to 0 (black).**

## 3. Hough Transform for circle:

The Hough Transform is used in image analysis to detect simple shapes such as lines, circles, and ellipses.

The Hough Circle Transform works by mapping points in the image space to curves in the parameter space. For circles, the parameter space is defined by the circle's center coordinates (x, y) and radius (r).

**Code Explanation:**

```
def get_hough_transform(image):
    image_gray = cv.cvtColor(np.array(image), cv.COLOR_RGB2GRAY)  # Convert to
numpy array and grayscale
    image_gray = cv.medianBlur(image_gray,5)

    cimg = cv.cvtColor(image_gray,cv.COLOR_GRAY2RGB)
    circles =
cv.HoughCircles(image_gray,cv.HOUGH_GRADIENT,1,20,param1=50,param2=30,minRadius=5
,maxRadius=100)

    if circles is not None:
        circles = np.uint16(np.around(circles))
        for i in circles[0,:]:
         # draw the outer circle
         cv.circle(cimg,(i[0],i[1]),i[2],(0,255,0),2)
         # draw the center of the circle
         cv.circle(cimg,(i[0],i[1]),2,(0,0,255),3)
    return cimg
```

- **Convert Image to Grayscale and Apply Median Blur**
- **Converts the input image to a NumPy array and then to a grayscale image.**
- **Applies a median blur with a kernel size of 5 to reduce noise.**
- **Converts the grayscale image back to an RGB image for drawing colored circles.**
- **Apply Hough Circle Transform**
- **Applies the Hough Circle Transform to detect circles.**
    - **Parameters:**
        - cv.HOUGH_GRADIENT: The method to detect circles.
        - 1: Inverse ratio of the accumulator resolution to the image resolution.
        - 20: Minimum distance between the centers of the detected circles.
        - param1=50: Higher threshold for the Canny edge detector.
        - param2=30: Accumulator threshold for the circle centers.
        - minRadius=5: Minimum circle radius to be detected.
        - maxRadius=100: Maximum circle radius to be detected.
    - **Draw Detected Circles**
        - Checks if any circles were detected.

- Rounds the circle parameters to the nearest integer and converts them to unsigned 16-bit integers.
- Iterates through each detected circle.