EDGE and BOUNDARY

1. Write a program to perform to perform Sobel edge detection.
2. Write a program to perform to perform Canny edge detection.
3. Perform Laplasian edge detection.
4. Perform Harris corner detection

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
from google.colab.patches import cv2_imshow
```

```
img = cv2.resize(cv2.imread('/content/drive/MyDrive/Cvip_Lab/images/car.jpg', cv2.IMREAD_COLOR), (450,300))
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
cv2_imshow(gray)
```

```
blur_img = cv2.GaussianBlur(gray,(3,3),0)
cv2_imshow(blur_img)
```

Sobel edge detection

Sobel edge detection is a popular technique used to identify edges in images by calculating the gradient magnitude and direction. It involves convolving the image with two 3x3 kernels, one for horizontal changes and the other for vertical changes.

Formulas:

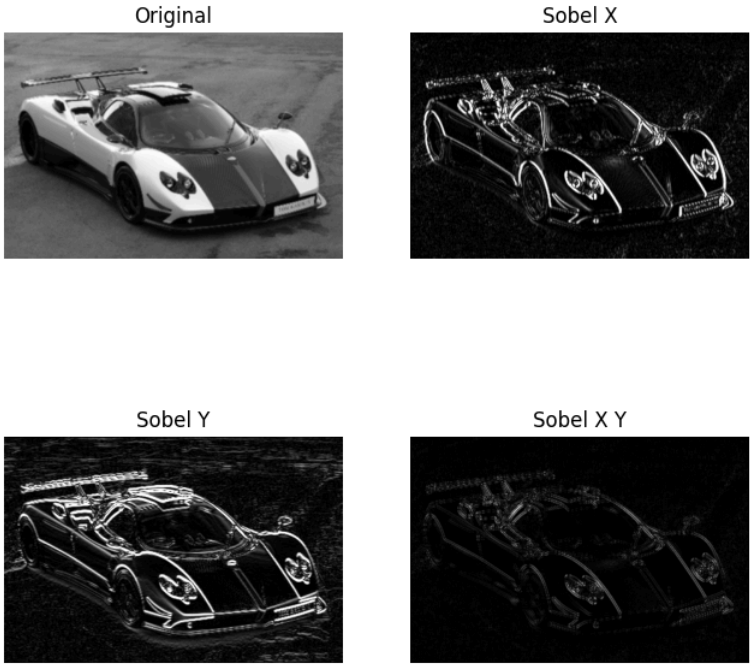
1. Horizontal gradient (G_x):
$$G_x = (P1 + 2 * P2 + P3) - (P7 + 2 * P8 + P9)$$
2. Vertical gradient (G_y):
$$G_y = (P1 + 2 * P4 + P7) - (P3 + 2 * P6 + P9)$$

Edge Magnitude: The magnitude of the edge can be calculated as:
$$G = \sqrt{G_x^2 + G_y^2}$$

Edge Direction: The direction of the edge can be calculated as:
$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Conclusion: Sobel edge detection is a simple yet effective method for detecting edges in images, making it a fundamental technique in computer vision and image processing applications.

```
sobelx = cv2.Sobel(src=blur_img, ddepth=cv2.CV_64F, dx=1, dy=0, ksize=3)
filtered_image_x = cv2.convertScaleAbs(sobelx)

sobely = cv2.Sobel(src=blur_img, ddepth=cv2.CV_64F, dx=0, dy=1, ksize=3)
filtered_image_y = cv2.convertScaleAbs(sobely)

sobelxy = cv2.Sobel(src=blur_img, ddepth=cv2.CV_64F, dx=1, dy=1, ksize=3)
filtered_image_xy = cv2.convertScaleAbs(sobelxy)

plt.figure(figsize=(8,8))
plt.subplot(221)
plt.imshow(blur_img, cmap='gray')
plt.title('Original')
plt.axis("off")

plt.subplot(222)
plt.imshow(filtered_image_x, cmap='gray')
plt.title('Sobel X')
plt.axis("off")

plt.subplot(223)
plt.imshow(filtered_image_y, cmap='gray')
plt.title('Sobel Y')
plt.axis("off")

plt.subplot(224)
plt.imshow(filtered_image_xy, cmap='gray')
plt.title('Sobel X Y')
plt.axis("off")
plt.show()
```



Canny edge detection

Canny edge detection is a multi-step algorithm used to detect a wide range of edges in images with low error rates. It involves several key steps including Gaussian smoothing, gradient calculation, non-maximum suppression, and edge tracking by hysteresis.

Formulas:

1. Gaussian Smoothing:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

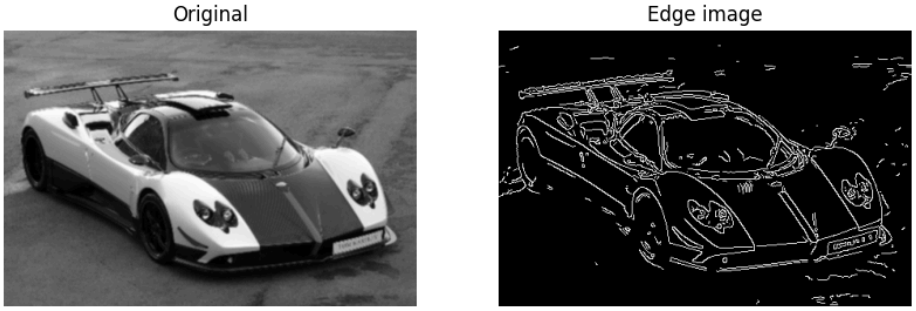
2. Gradient Magnitude (G) and Direction (θ):

$$G = \sqrt{G_x^2 + G_y^2}$$
$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

3. Non-Maximum Suppression: Compare the gradient magnitude of a pixel with its neighbors along the gradient direction to keep only local maxima.
4. Hysteresis Edge Tracking: Use two threshold values, high and low, to determine strong and weak edges. Perform edge tracking by connecting strong edges and weak edges adjacent to strong edges.

Conclusion: Canny edge detection is a robust technique that provides accurate edge detection results by considering gradient information and suppressing non-maximum responses. It is widely used in computer vision and image processing for various applications requiring precise edge detection.

```
edges = cv2.Canny(image=blur_img, threshold1=80, threshold2=110)

plt.figure(figsize=(10,10))
plt.subplot(121)
plt.imshow(blur_img, cmap='gray')
plt.title('Original')
plt.axis("off")

plt.subplot(122)
plt.imshow(edges, cmap='gray')
plt.title('Edge image')
plt.axis("off")
plt.show()
```



Laplacian edge detection

Laplacian edge detection is a technique used to detect edges in images by computing the second derivative of the image intensity. It highlights regions where the intensity changes rapidly, indicating the presence of edges.

Formulas:

1. Laplacian operator:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

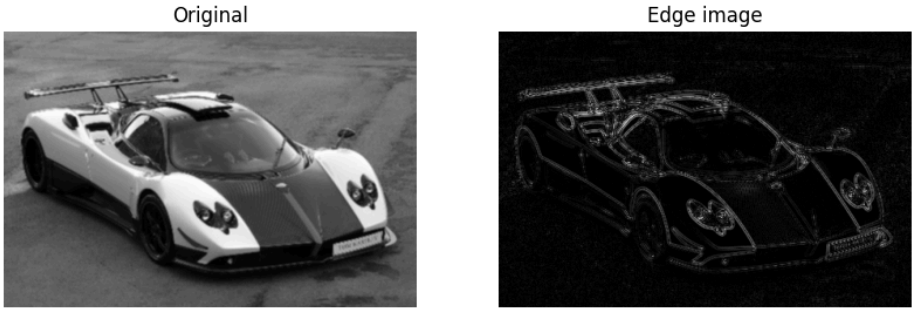
2. Discrete Laplacian operator:

$$\Delta^2 f = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y)$$

Edge Detection: By applying the Laplacian operator to the image, regions with a zero-crossing in the Laplacian response indicate the presence of edges.

Conclusion: Laplacian edge detection is a simple yet effective method for detecting edges in images by highlighting regions of rapid intensity changes. It is commonly used in computer vision and image processing applications for edge detection and feature extraction tasks.

```
laplacian = cv2.Laplacian(blur_img,3,cv2.CV_64F)
filtered_image = cv2.convertScaleAbs(laplacian)
plt.figure(figsize=(10,10))
plt.subplot(121)
plt.imshow(blur_img, cmap='gray')
plt.title('Original')
plt.axis("off")

plt.subplot(122)
plt.imshow(filtered_image, cmap='gray')
plt.title('Edge image')
plt.axis("off")
plt.show()
```



Harris corner detection

Harris corner detection is a popular method used to detect corners in images by analyzing variations in intensity. Corners are important features in images as they provide key information for tasks like image stitching, object recognition, and tracking.

Formulas:

1. Structure Tensor: The structure tensor is computed using the gradients of the image:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

2. Harris Corner Response Function: The Harris corner response function is calculated using the structure tensor:

$$R = \det(M) - k(\text{trace}(M))^2$$

3. Corner Detection: Corners are detected by identifying local maxima in the Harris corner response function.

Conclusion: Harris corner detection is a robust technique for detecting corners in images, making it a valuable tool in computer vision and image processing applications. It provides accurate corner localization and is widely used in various tasks such as feature matching, image registration, and object tracking.

```
gray = np.float32(gray)
dst = cv2.cornerHarris(gray,2,3,0.1)

plt.figure(figsize=(10,10))
plt.imshow(dst)
plt.title("Corner Detection")
plt.show()

dst = cv2.dilate(dst,None)
# Threshold for an optimal value, it may vary depending on the image.
img[dst>0.01*dst.max()]=[0,0,255]
cv2_imshow(img)
```