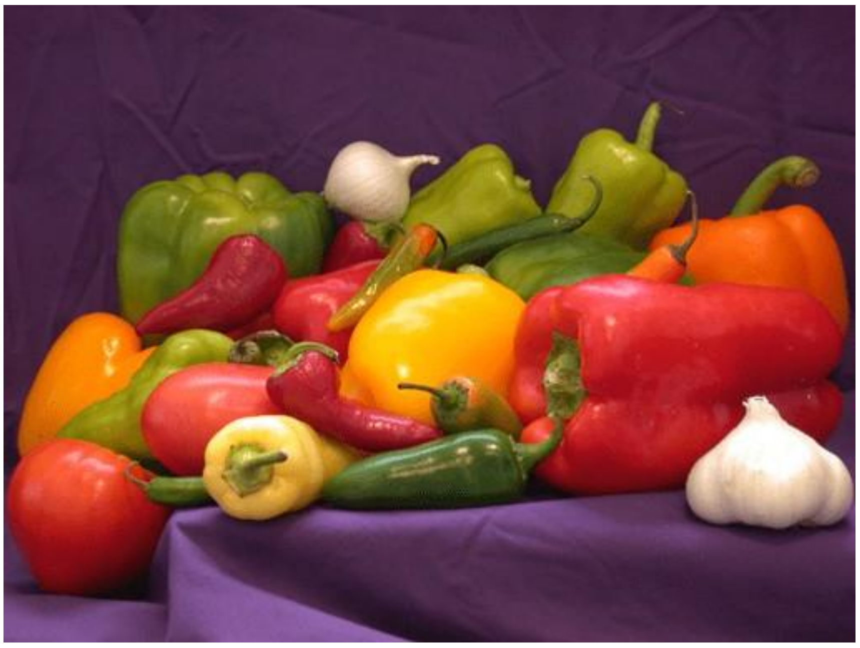
Name: Hitesh Choudhary

PRN: 20200802146

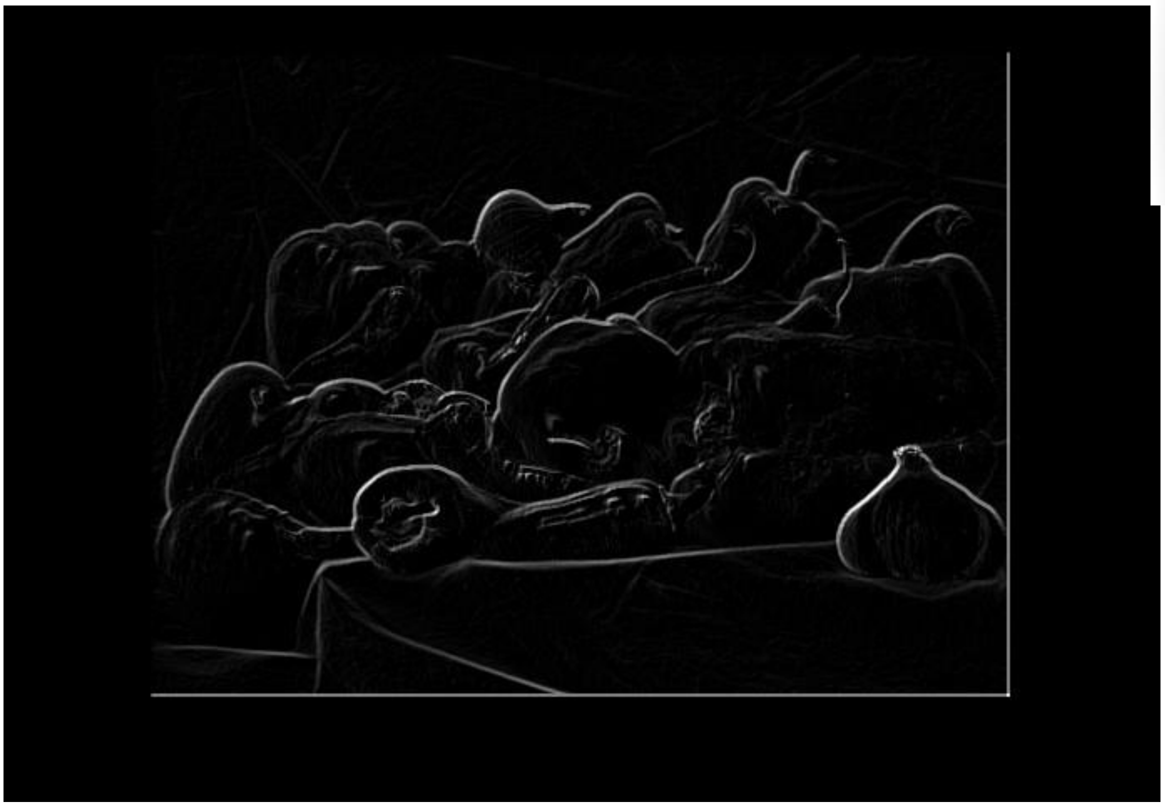
A. Edge Detection using Prewitt filter.

In [ ]:
```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

# Read the image
image_path = '/content/Vegetable Image.png'  # Provide the path to your ima
ge file
original_image = cv2.imread(image_path)

# Convert the image to grayscale
gray_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)

# Perform Prewitt operation for horizontal mask
prewitt_horizontal = cv2.filter2D(gray_image, -1, np.array([[-1, 0, 1], [-
1, 0, 1], [-1, 0, 1]]))

# Perform Prewitt operation for vertical mask
prewitt_vertical = cv2.filter2D(gray_image, -1, np.array([[-1, -1, -1], [0,
0, 0], [1, 1, 1]]))

# Combine the horizontal and vertical edge images
prewitt_combined = cv2.addWeighted(prewitt_horizontal, 0.5, prewitt_vertica
l, 0.5, 0)

# Display the original image and the edge-detected image
print("----------------original image------------------------------------")
cv2_imshow(original_image)
print("---------------------------edge bounding---------------------------")
cv2_imshow(prewitt_combined)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

```
----------------original image------------------------
```



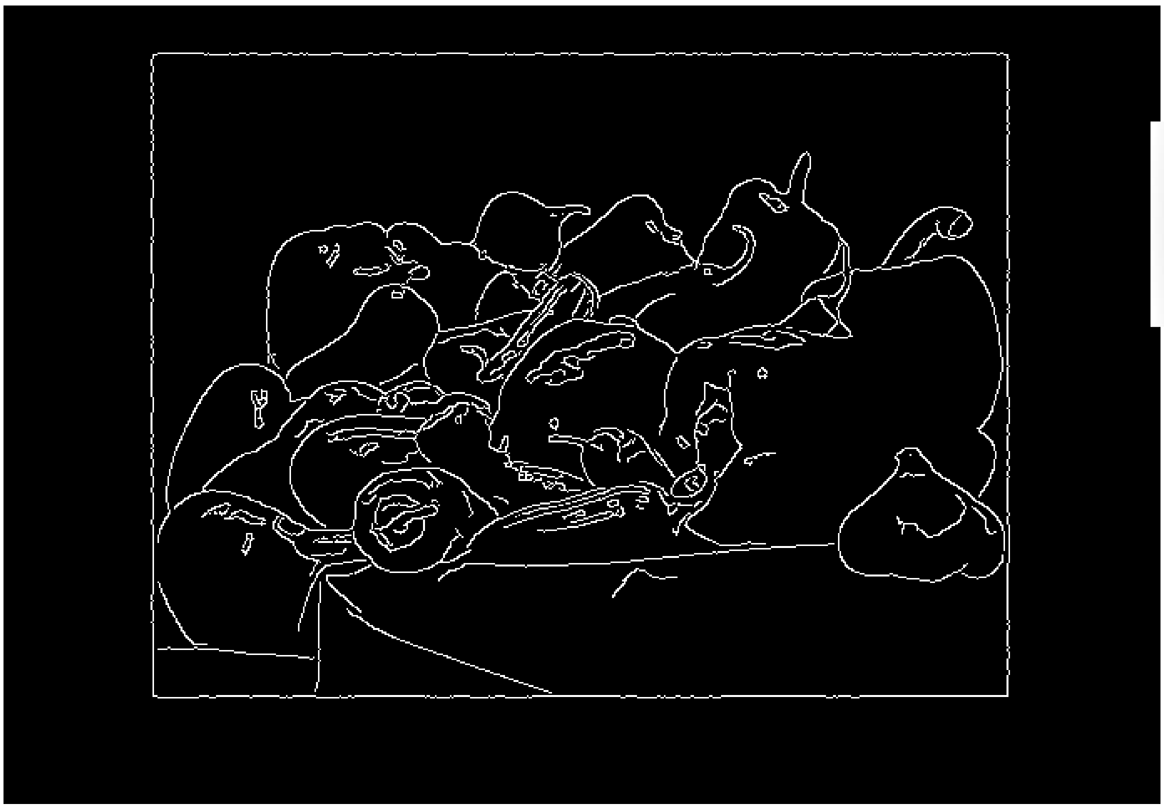
```
---------------------------edge bounding-----------------
```



B. Canny edge detection.

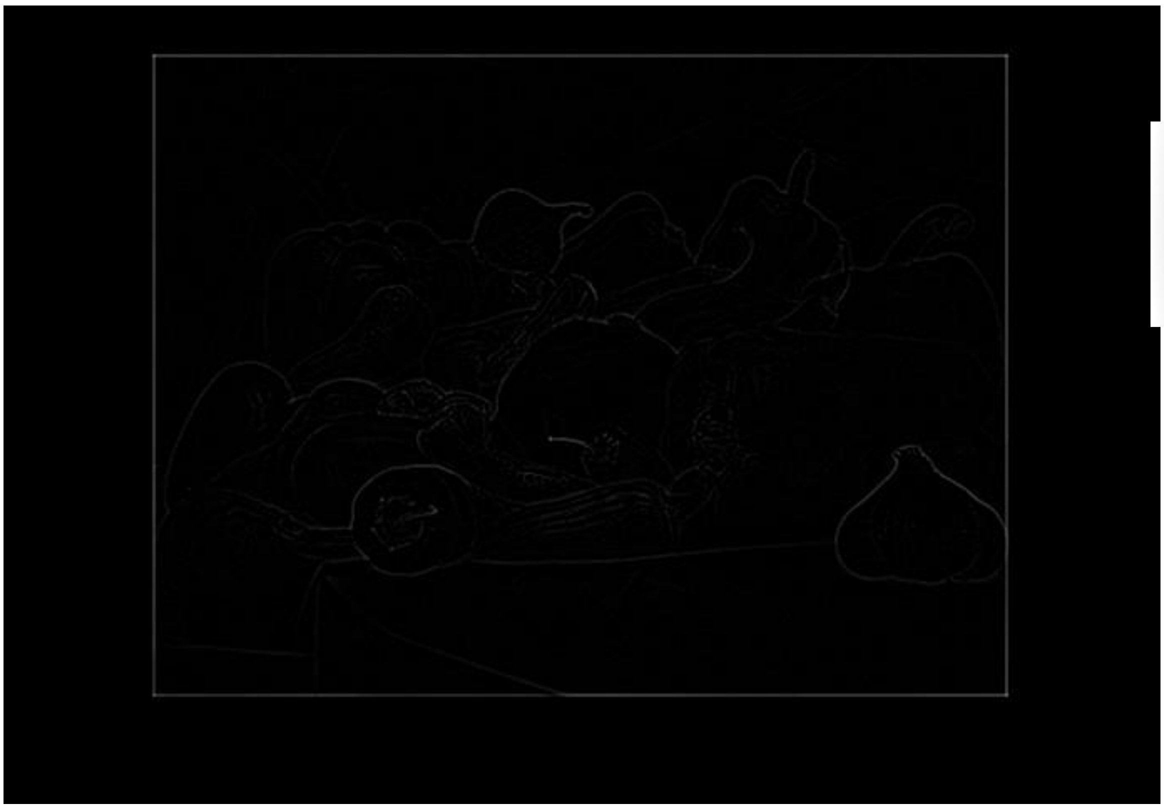
In [ ]:

```
import cv2
import numpy as np


# Step 2: Applying Gaussian blur filter to smooth data
blurred_image = cv2.GaussianBlur(original_image, (5, 5), 0)

# Step 3: Apply the Sobel filter to get the intensity and edge direction ma
trices
gray_image = cv2.cvtColor(blurred_image, cv2.COLOR_BGR2GRAY)
sobel_x = cv2.Sobel(gray_image, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(gray_image, cv2.CV_64F, 0, 1, ksize=3)

# Step 4: Identify the thick and thin edges
gradient_magnitude = np.sqrt(sobel_x ** 2 + sobel_y ** 2)
gradient_magnitude = np.uint8(gradient_magnitude)

# Step 5: Apply non-Max Suppression to thin the edges
sobel_x = np.absolute(sobel_x)
sobel_y = np.absolute(sobel_y)
sobel_combined = cv2.bitwise_or(sobel_x, sobel_y)

# Step 6: Apply Double Threshold to find Strong, Weak, and Non-relevant pix
els
threshold_value_1 = 30
threshold_value_2 = 100
canny_edges = cv2.Canny(blurred_image, threshold_value_1, threshold_value_
2)

# Step 7: Perform edge tracking hysteresis to convert weak pixels into stro
nger pixels
lower_threshold = 50
higher_threshold = 150
canny_edges = cv2.Canny(blurred_image, lower_threshold, higher_threshold)

# Step 8: Provide the output image
cv2_imshow(original_image)
cv2_imshow(canny_edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```



C. Laplacian of Gaussian:

In [ ]:
```
import cv2


# Step 2: Remove noise by applying a Gaussian blur
blurred_image = cv2.GaussianBlur(original_image, (3, 3), 0)

# Step 3: Convert the image to grayscale
gray_image = cv2.cvtColor(blurred_image, cv2.COLOR_BGR2GRAY)

# Step 4: Apply a Laplacian operator to the grayscale image and obtain the
output image
laplacian = cv2.Laplacian(gray_image, cv2.CV_64F)

# Step 5: Display the result in a window
cv2_imshow(original_image)
cv2_imshow(laplacian)
cv2.waitKey(0)
cv2.destroyAllWindows()
```



D. Perform and comparison between Gradient-based and Gaussian-based operators

Gradient-based and Gaussian-based operators are both commonly used for edge detection in image processing. Each method has its own advantages and is suitable for different scenarios. Here's a comparison of the two approaches:

1. Gradient-based operators:

   - Examples include Sobel, Prewitt, and Roberts operators.
   - They calculate the gradient of the image intensity at each pixel, highlighting regions of rapid intensity change.
   - They are sensitive to noise and can produce thick edges, leading to a less precise localization of edges.
   - These operators are computationally less expensive compared to Gaussian-based operators.
   - They are effective for real-time applications where speed is crucial.
2. Gaussian-based operators:

   - Examples include the Laplacian of Gaussian (LoG) and the Difference of Gaussian (DoG) operators.
   - They involve convolving the image with a Gaussian kernel to smooth the image and reduce noise.
   - They detect edges by identifying zero-crossings in the second derivative of the smoothed image.
   - They are less sensitive to noise compared to gradient-based operators, resulting in more accurate edge localization.
   - These operators are computationally more expensive due to the convolution with the Gaussian kernel.

When to use which:

- Use gradient-based operators when real-time processing is required and when the presence of noise is not significant.
- Use Gaussian-based operators when accurate edge localization is crucial, especially in cases where the image contains a considerable amount of noise.

In practice, the choice between the two approaches depends on the specific requirements of the application and the characteristics of the images being processed. It is often beneficial to experiment with both methods and choose the one that yields the best results for a particular task.