CSC 482
Assignment 1
Hithesh Shanmugam

A. Write a simple edge detector based on the Canny edge detector:

1. Image Smoothing:

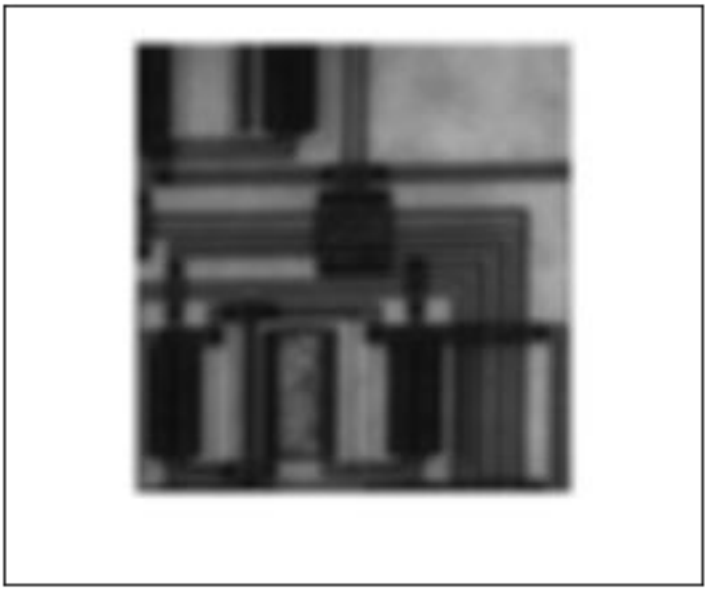
Code:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from scipy import ndimage
from scipy.ndimage import gaussian_filter
from PIL import Image
```

```
def gaussian_smooth(image, sigma):
    # calculate the width of the filter
    width = 2 * sigma + 1
    # create a 2D Gaussian kernel
    kernel = cv2.getGaussianKernel(width, sigma)
    kernel = kernel * kernel.T
    # apply the kernel to the image using a convolution
    return cv2.filter2D(image, -1, kernel)
```

```
image = cv2.imread("C:/Users/sures/OneDrive - DePaul University/Desktop/edge.png")
image=cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
smooth_image = gaussian_smooth(image, 4)
plt.imshow(smooth_image)
plt.xticks([]),plt.yticks([])
plt.show()
```

Output:



2. **Gradient Image**

Code:

```
def image_gradient(image):
    # calculate the x and y gradients using the Sobel operator
    grad_x = cv2.Sobel(image, cv2.CV_32F, 1, 0, ksize=3)
    grad_y = cv2.Sobel(image, cv2.CV_32F, 0, 1, ksize=3)
    return grad_x, grad_y
```

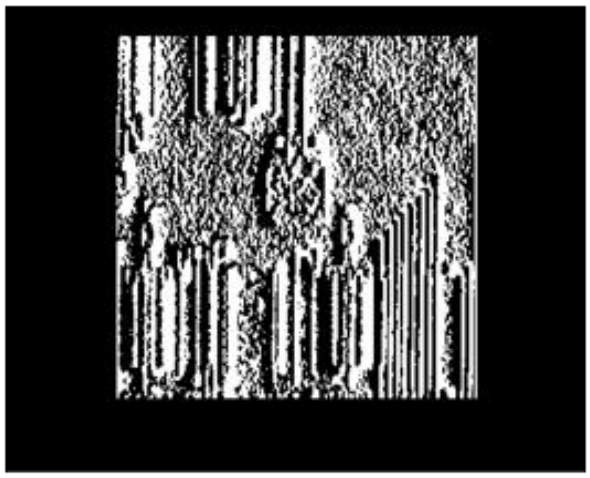
```
grad_x, grad_y = image_gradient(image)
```

```
plt.imshow(grad_x)
plt.xticks([]),plt.yticks([])
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

Gradient X:

Output:

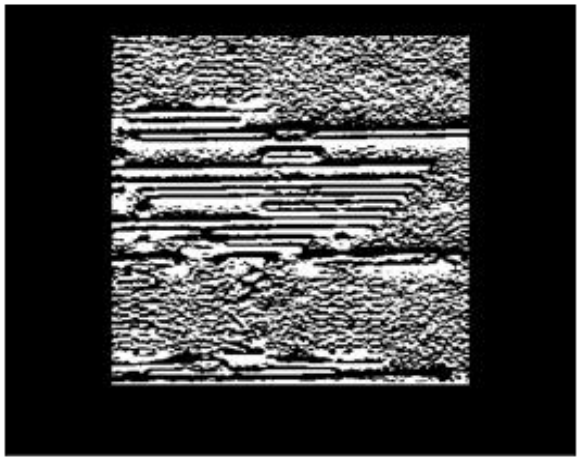


```
plt.imshow(grad_y)
plt.xticks([]),plt.yticks([])
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

Gradient Y:

Output:



3. **Magnitude and direction of the gradient**

Code:

```
def gradient_magnitude_direction(grad_x, grad_y):
    # Compute the magnitude of the gradient
    magnitude = np.sqrt(grad_x**2 + grad_y**2)
    # Normalize the gradient direction
    direction_x = grad_x / (magnitude + 1e-8)
    direction_y = grad_y / (magnitude + 1e-8)
    return magnitude, direction_x, direction_y
```

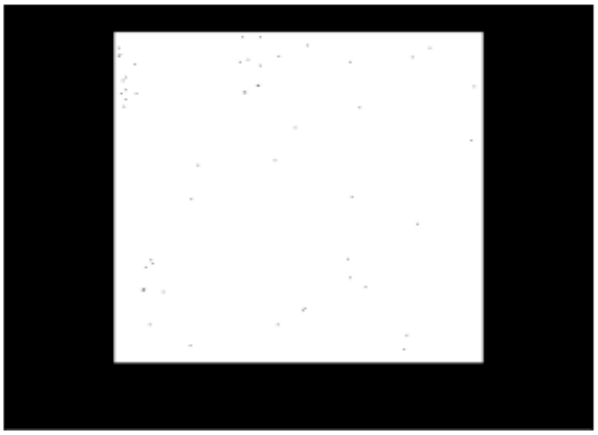
```
magnitude, direction_x, direction_y=gradient_magnitude_direction(grad_x,grad_y)
```

Magnitude:

```
plt.imshow(magnitude)
plt.xticks([]),plt.yticks([])
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

Output:

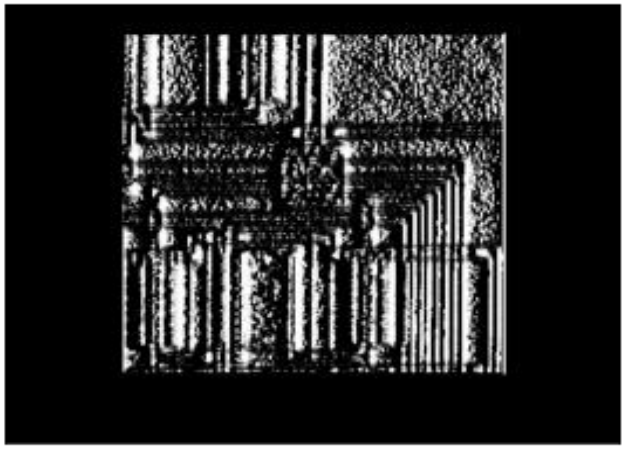


Direction of x:

```
plt.imshow(direction_x)
plt.xticks([]),plt.yticks([])
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

Output:

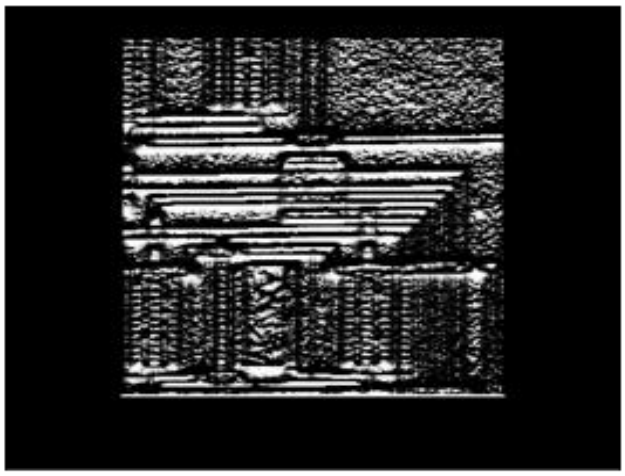


Direction of y:

```
plt.imshow(direction_y)
plt.xticks([]),plt.yticks([])
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

Output:



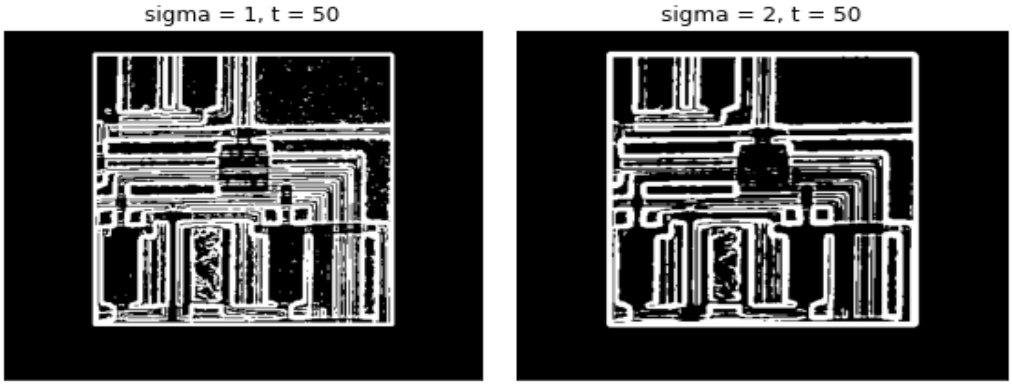
4. **Edge Pixels**

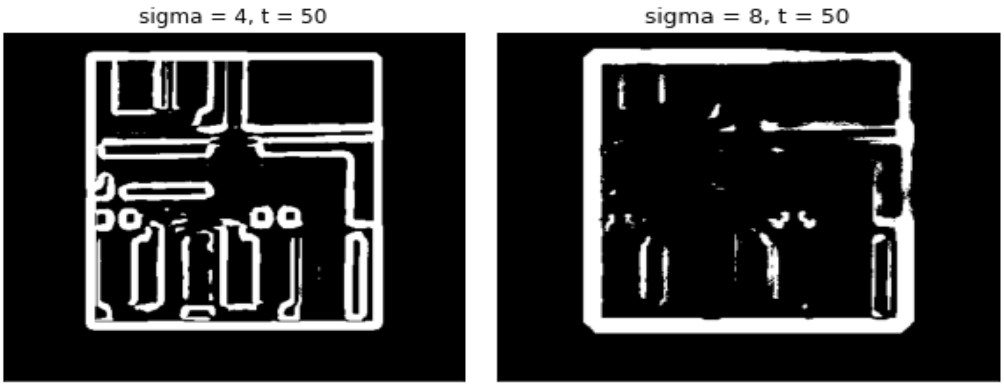
Code:

```
def detect_edges(image, sigma, t):
    # Smooth the image
    smoothed_image = gaussian_smooth(image, sigma)

    # Compute the gradient
    x_gradient, y_gradient = image_gradient(smoothed_image)

    # Compute the magnitude and direction of the gradient
    magnitude, direction_x, direction_y = gradient_magnitude_direction(x_gradient, y_gradient)

    # Create a binary image with edges
    edges = np.zeros_like(magnitude)
    edges[magnitude > t] = 1

    return edges
```

```
edges1 = detect_edges(image, 1, 50)
plt.imshow(edges1, cmap='gray')
plt.title('sigma = 1, t = 50')
plt.xticks([]),plt.yticks([])
plt.show()
```

Output:





Explanation:

As we can see that increasing the sigma value gives thicker edges this is because in my opinion increasing the sigma makes the filter size bigger and this leads to more smoothing of the image that is more gaussian blur so it gets hard to find the edges.

Hysteresis:

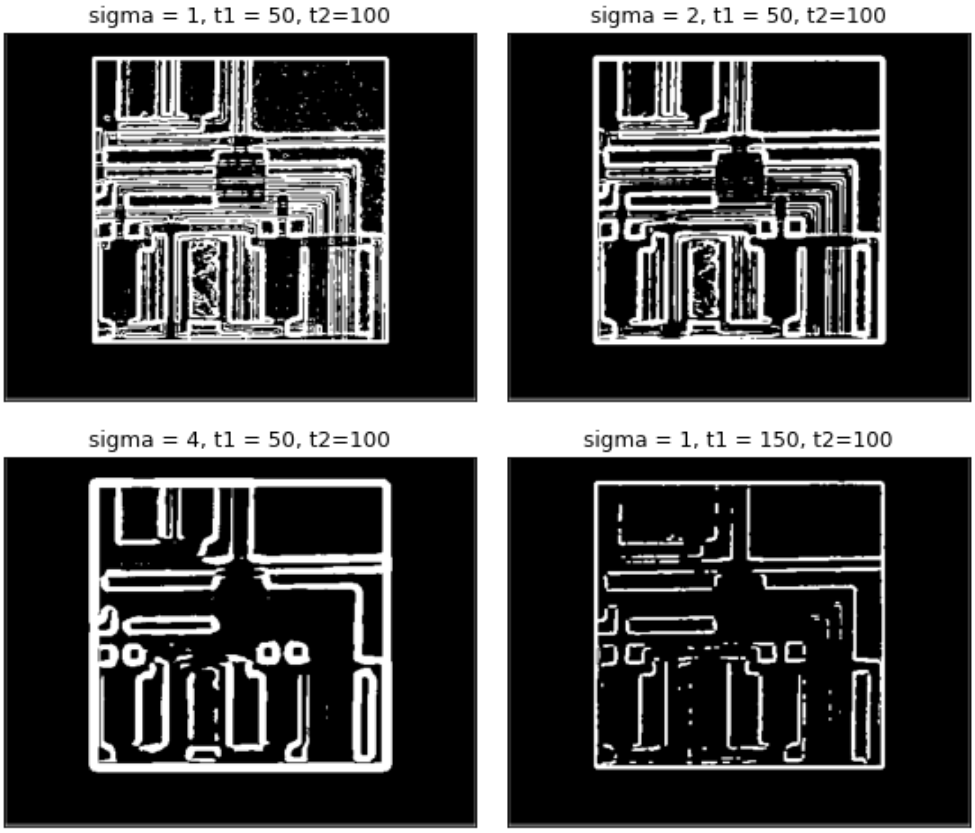
Code:

```
def find_edges(image, sigma, t1, t2):
    # Smooth the image using a Gaussian filter
    smooth_image = gaussian_smooth(image, sigma)
    # Add border to the image
    smooth_image=cv2.copyMakeBorder(smooth_image,1,1,1,1,cv2.BORDER_CONSTANT,value=0)
    # Compute the gradient of the image
    grad_x, grad_y = image_gradient(smooth_image)
    # Compute the magnitude and direction of the gradient
    magnitude, _, _ = gradient_magnitude_direction(grad_x, grad_y)
    # Convert the image to a matrix of floating point numbers
    magnitude = magnitude.astype(np.float32)
    # Create a binary matrix with edges detected using a threshold and hysteresis
    edges = np.zeros(magnitude.shape, dtype=np.uint8)
    edges[magnitude >= t1] = 255
    strong_edges = (magnitude >= t1)
    weak_edges = (magnitude < t1) & (magnitude >= t2)
    for x in range(1,weak_edges.shape[0]-1):
        for y in range(1,weak_edges.shape[1]-1):
            if weak_edges[x, y].any():
                if strong_edges[x-1:x+2, y-1:y+2].any():
                    edges[x, y] = 255
    return edges
```

```
edges=find_edges(image, 1, 50, 100)
```

```
plt.imshow(edges)
plt.title('sigma = 1, t1 = 50, t2=100')
plt.xticks([]),plt.yticks([])
plt.show()
```

Output:



Explanation:

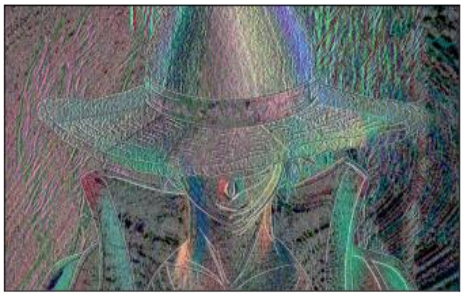
The same thing applies here the more the sigma value the bad the edges are but in here I tried to change the thresholds in the better sigma value that is 1 and if threshold 1 is greater than threshold 2 then the detection of edges is affected by those values.

Color Image:

Image Smoothing:

Gradient Image:

Gradient of X

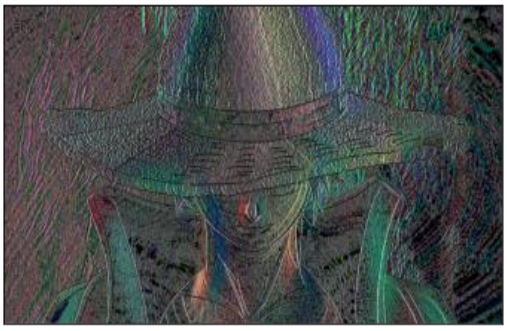


Gradient of Y

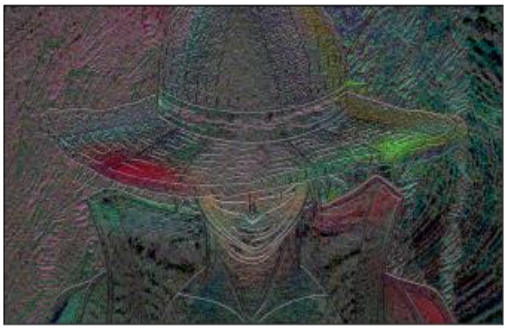
Magnitude and direction of the gradient:

Magnitude

Direction of X



Direction of Y



Edge Pixels:

sigma = 1, t = 50

sigma = 2, t = 50

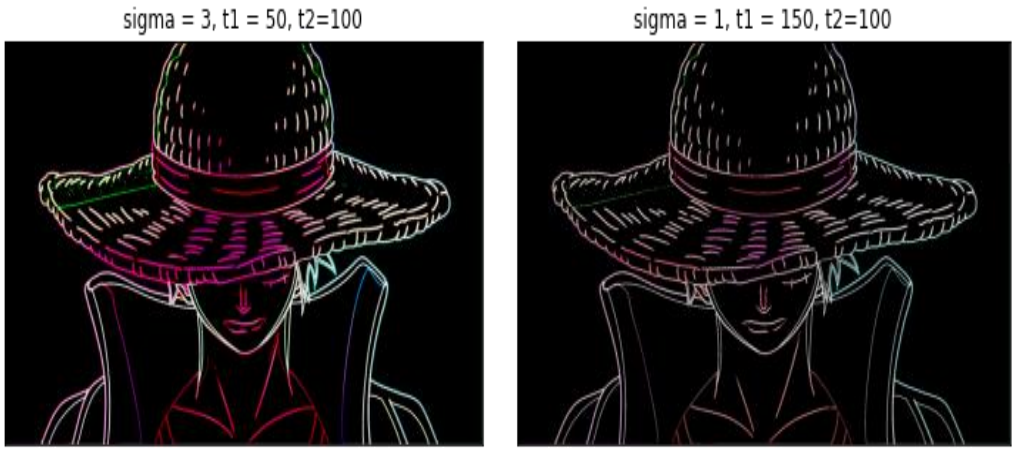
sigma = 3, t = 50

sigma = 4, t = 50

Hysteresis:

sigma = 1, t1 = 50, t2=100

sigma = 2, t1 = 50, t2=100

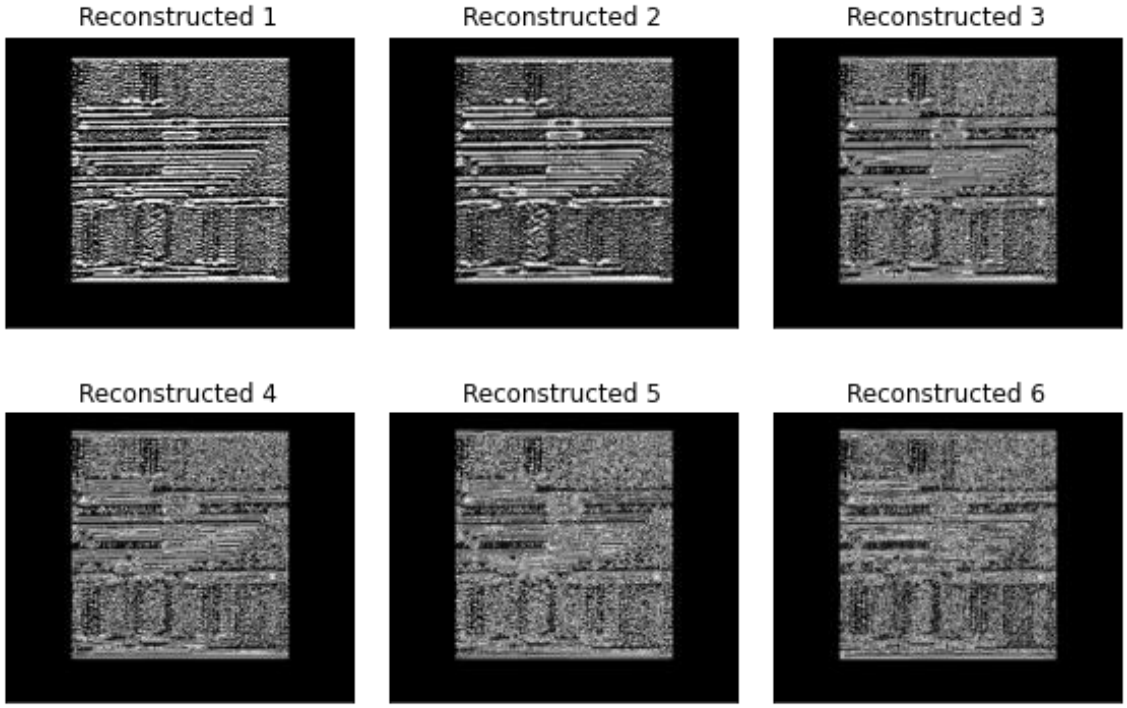


B. (20 points) Perform edge detection at different scales using a Laplacian Pyramid implemented through differences of Gaussians at different scales.

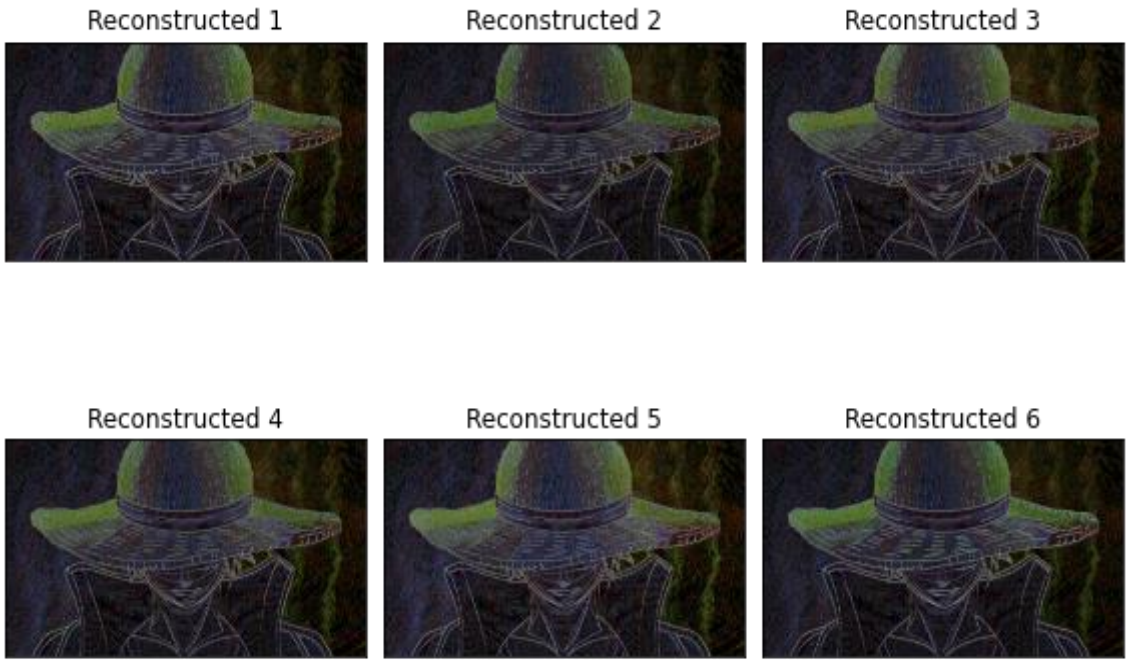
Code:

```
# Load the image
img = cv2.imread("C:/Users/sures/OneDrive - DePaul University/Desktop/edge.png", cv2.IMREAD_GRAYSCALE)
# Create the Gaussian pyramid
gaussian_pyramid = [img]
kernel = cv2.getGaussianKernel(5,1)
for i in range(6):
    gaussian_pyramid.append(cv2.filter2D(gaussian_pyramid[i],-1,kernel))

# Create the Laplacian pyramid
laplacian_pyramid = []
for i in range(6):
    laplacian = gaussian_pyramid[i] - cv2.filter2D(gaussian_pyramid[i+1],-1,kernel)
    laplacian_pyramid.append(laplacian)
fig, axs = plt.subplots(2, 3, figsize=(10, 6),
                        subplot_kw={'xticks': [], 'yticks': []})
fig.subplots_adjust(hspace=0.3, wspace=0.05)

for i in range(6):
    reconstructed_image = laplacian_pyramid[0]
    for j in range(1, i+1):
        reconstructed_image += cv2.filter2D(laplacian_pyramid[j],-1,kernel)
    axs.ravel()[i].imshow(reconstructed_image, cmap='gray')
    axs.ravel()[i].set_title("Reconstructed "+str(i+1))
plt.show()
```

Output:



Color Image Output:



Explanation:

I created a Gaussian pyramid by applying a Gaussian filter to the original image using the cv2.filter2D() function. A kernel of size 5 and sigma 1 is used to smooth the image. The filtered image is then appended to the gaussian_pyramid list. This process is repeated for 6 scales by down sampling the image by a factor of 2 each time.

The Laplacian pyramid is then created by subtracting the expanded version of the next level of the Gaussian pyramid from the current level using the same kernel. The expanded version is obtained by using the cv2.filter2D() function with the same kernel. The result is a DoG pyramid also known as a Laplacian pyramid.

The code then uses the Matplotlib library to display the reconstructed images from the Laplacian pyramid. The images are reconstructed by summing the levels of the Laplacian pyramid starting from the first level. Each reconstructed image is displayed in a subplot with a title indicating the level of reconstruction.

The edge detection in the code is achieved by using the Laplacian pyramid, which highlights the areas of the image where there are rapid changes in intensity. The result is a set of images where the edges are emphasized and the rest of the image is suppressed. The choice of the kernel size and sigma in the cv2.filter2D() function affect the final result, a smaller kernel size and larger sigma will result in more edges being detected, while a larger kernel size and smaller sigma will result in less edges being detected. If you want to compare the results with the results obtained for edge detection by just applying the Laplacian of Gaussian (LoG) transform you can use the cv2.Laplacian() function instead of using the cv2.filter2D() function.