Canny Edge Detector: Project Report

Pragnavi Ravuluri Sai Durga
Email: pr2370@nyu.edu

Santosh Srinivas
Email: sr6411@nyu.edu

Introduction

Canny edge detector is a popular image processing technique used to detect edges in digital images. It was developed by John Canny in 1986 and is still widely used in computer vision applications. The algorithm works by finding the edges of an image based on their local intensity gradients, and then applying a series of thresholding and filtering steps to produce a binary image with strong edge pixels. In this project, we explore the Canny edge detector algorithm in detail, implement it in Python, and apply it to various test images to analyze its output in different scenarios. Through this project, we gain a better understanding of the Canny edge detector, its applications and contribution to the field of computer vision research.

Objective

The primary objective of the project is to implement the Canny's Edge Detector algorithm in Python, which involves four major steps, namely, Gaussian smoothing, gradient operation, non-maxima suppression, and thresholding. The program is designed to accept a grayscale image of size $N \times M$ (rows $\times$ columns) as input.

The output of the program at each of the four steps is as follows:

- An image result after Gaussian smoothing
- A normalized magnitude image
- A normalized magnitude image after non-maxima suppression
- Binary edge maps for thresholds chosen at the 25^{th}, 50^{th}, and 75^{th} percentiles
- A histogram of the normalized magnitude image after non-maxima suppression

The project's goals and objectives were meticulously executed ensuring that the algorithm's implementation is accurate and aligned with the proposed objectives.

Instructions: How to Run the Program

The following are the instructions for executing the Canny Edge Detector program:

System Requirements:

It is essential to ensure that Python and the required packages, namely NumPy, Matplotlib, OpenCV, and Pillow, are installed in the local machine before running the program. In case any of the packages are not installed, execute the following commands on the terminal:

- NumPy - pip3 install numpy
- Matplotlib - pip3 install matplotlib
- OpenCV - pip3 install opencv-python
- Pillow - pip3 install pillow

After installing the packages, the program is ready for compilation.

Running the Program:

First, save the source code file canny.py and the three test images, namely barbara.bmp, peppers.bmp, and goldhill.bmp, in the same directory folder. Then, navigate to the directory in the terminal and execute the following command for each of the test images, passing the .py file and the test image as arguments:

- python3 canny.py barabara.bmp
- python3 canny.py goldhill.bmp
- python3 canny.py peppers.bmp

The output images of the input test image are stored in the same directory. For example, for barbara.bmp, the output images of the Gaussian function, gradient smoothing, non-maxima suppression, thresholding, and histogram are named as follows:

- 1_Gaussian_Smoothened_barbara.bmp
- 2_Gradient_Magnitude_barbara.bmp
- 3_NMS_barbara.bmp
- 4_Binary_Edge_Map_T1_barbara.bmp
- 5_Binary_Edge_Map_T2_barbara.bmp
- 6_Binary_Edge_Map_T3_barbara.bmp
- 7_nms_histogram_barbara.png (pyplot can't be saved as .bmp file)

Source Code

```python
'''
The following program is the implementation of Canny's Edge Detector algorithm in Python, which involves four major steps, namely,
Gaussian smoothing, gradient operation, non-maxima suppression, and thresholding. The program accepts
a grayscale image of size N * M (rows * columns) as input.
The output of the program at each of the four steps is as follows:
1) an image result after Gaussian smoothing,
(2) a normalized magnitude image,
3) a normalized magnitude image after non-maxima suppression, and
(4) binary edge maps for thresholds chosen at the 25th, 50th, and 75th percentiles
(5) a histogram of the normalized magnitude image after non-maxima suppression.

Contributors:

Pragnavi RSD(pr2370)
Santosh Srinivas Ravichandran(sr6411)

'''

# Import necessary libraries
import numpy as np  # for numerical computations
from PIL import Image  # for image processing
import math  # for mathematical operations
import sys  # for system-level operations
import cv2  # for computer vision and image processing
import matplotlib.pyplot as plt #for plotting histograms and graphs
import os # using operating system dependent functionality like running system commands, extracting the file format extension from file path etc

# This function applies Gaussian smoothing on an input grayscale image using a pre-defined kernel.
# The function takes the image path as input and returns the smoothed image.

def gaussian_smoothing(img_path):

    # Load input image in grayscale
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    # Get image height and width
    img_h, img_w = img.shape

    # Define Gaussian kernel
    gaussian_mask = np.array([
        [1, 1, 2, 2, 2, 1, 1],
        [1, 2, 2, 4, 2, 2, 1],
        [2, 2, 4, 8, 4, 2, 2],
        [2, 4, 8, 16, 8, 4, 2],
        [2, 2, 4, 8, 4, 2, 2],
        [1, 2, 2, 4, 2, 2, 1],
        [1, 1, 2, 2, 2, 1, 1]
    ])

    # Get size of Gaussian kernel
    mask_size = gaussian_mask.shape[0]

    # Set buffer size for boundary pixels
    buffer = mask_size // 2

    # Initialize a 2D array to hold the smoothed image
    smooth_image = np.empty((img_h,img_w), dtype=np.float32)


    # Apply Gaussian smoothing using the pre-defined kernel
    for i in range(0+buffer,img_h-buffer,1):
        for j in range(0+buffer,img_w-buffer,1):
            # Calculate the weighted sum of the pixels in the kernel
            smooth_image[i][j] = \
                gaussian_mask[0][0]*img[i-buffer][j-buffer] \
                + gaussian_mask[0][1]*img[i-buffer][j-buffer+1] \
                + gaussian_mask[0][2]*img[i-buffer][j-buffer+2] \
                + gaussian_mask[0][3]*img[i-buffer][j-buffer+3] \
                + gaussian_mask[0][4]*img[i-buffer][j-buffer+4] \
                + gaussian_mask[0][5]*img[i-buffer][j-buffer+5] \
                + gaussian_mask[0][6]*img[i-buffer][j-buffer+6] \
                + gaussian_mask[1][0]*img[i-buffer+1][j-buffer] \
                + gaussian_mask[1][1]*img[i-buffer+1][j-buffer+1] \
                + gaussian_mask[1][2]*img[i-buffer+1][j-buffer+2] \
                + gaussian_mask[1][3]*img[i-buffer+1][j-buffer+3] \
                + gaussian_mask[1][4]*img[i-buffer+1][j-buffer+4] \
                + gaussian_mask[1][5]*img[i-buffer+1][j-buffer+5] \
                + gaussian_mask[1][6]*img[i-buffer+1][j-buffer+6] \
                + gaussian_mask[2][0]*img[i-buffer+2][j-buffer] \
                + gaussian_mask[2][1]*img[i-buffer+2][j-buffer+1] \
                + gaussian_mask[2][2]*img[i-buffer+2][j-buffer+2] \
                + gaussian_mask[2][3]*img[i-buffer+2][j-buffer+3] \
                + gaussian_mask[2][4]*img[i-buffer+2][j-buffer+4] \
                + gaussian_mask[2][5]*img[i-buffer+2][j-buffer+5] \
                + gaussian_mask[2][6]*img[i-buffer+2][j-buffer+6] \
                + gaussian_mask[3][0]*img[i-buffer+3][j-buffer] \
                + gaussian_mask[3][1]*img[i-buffer+3][j-buffer+1] \
                + gaussian_mask[3][2]*img[i-buffer+3][j-buffer+2] \
                + gaussian_mask[3][3]*img[i-buffer+3][j-buffer+3] \
                + gaussian_mask[3][4]*img[i-buffer+3][j-buffer+4] \
                + gaussian_mask[3][5]*img[i-buffer+3][j-buffer+5] \
                + gaussian_mask[3][6]*img[i-buffer+3][j-buffer+6] \
                + gaussian_mask[4][0]*img[i-buffer+4][j-buffer] \
                + gaussian_mask[4][1]*img[i-buffer+4][j-buffer+1] \
                + gaussian_mask[4][2]*img[i-buffer+4][j-buffer+2] \
                + gaussian_mask[4][3]*img[i-buffer+4][j-buffer+3] \
                + gaussian_mask[4][4]*img[i-buffer+4][j-buffer+4] \
                + gaussian_mask[4][5]*img[i-buffer+4][j-buffer+5] \
                + gaussian_mask[4][6]*img[i-buffer+4][j-buffer+6] \
                + gaussian_mask[5][0]*img[i-buffer+5][j-buffer] \
                + gaussian_mask[5][1]*img[i-buffer+5][j-buffer+1] \
                + gaussian_mask[5][2]*img[i-buffer+5][j-buffer+2] \
                + gaussian_mask[5][3]*img[i-buffer+5][j-buffer+3] \
```

```python
                + gaussian_mask[5][4]*img[i-buffer+5][j-buffer+4] \
                + gaussian_mask[5][5]*img[i-buffer+5][j-buffer+5] \
                + gaussian_mask[5][6]*img[i-buffer+5][j-buffer+6] \
                + gaussian_mask[6][0]*img[i-buffer+6][j-buffer] \
                + gaussian_mask[6][1]*img[i-buffer+6][j-buffer+1] \
                + gaussian_mask[6][2]*img[i-buffer+6][j-buffer+2] \
                + gaussian_mask[6][3]*img[i-buffer+6][j-buffer+3] \
                + gaussian_mask[6][4]*img[i-buffer+6][j-buffer+4] \
                + gaussian_mask[6][5]*img[i-buffer+6][j-buffer+5] \
                + gaussian_mask[6][6]*img[i-buffer+6][j-buffer+6] \

    # Normalize the smoothed image by dividing with the sum of Gaussian kernel elements
    smooth_image = smooth_image/np.sum(gaussian_mask)

    # Show the smoothed image
    im = Image.fromarray(smooth_image).convert('L')
    im.show()

    # Save the smoothed image in the format 'Gaussian_Smoothened_imagename.bmp'
    im.save('1_Gaussian_Smoothened_'+img_file)

    return smooth_image


# This function computes the Gradient operation on the input gaussian smoothed image using predefined masks from the Robinson compass mask for edge detection.
# The function takes the gaussian smoothed image array as input and returns the normalized edge magnitude array and the gradient angle array.

def gradient_operation(gaussian_smooth_image):

    img = gaussian_smooth_image
    img_h, img_w = img.shape

    #Defining the gradient masks - g0 for 0 degree, g1 for 45 degree, g2 for 90 degree & g3 for 135 degree.
    g0 = np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
    g1 = np.array([[0,1,2],[-1,0,1],[-2,-1,0]])
    g2 = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])
    g3 = np.array([[2,1,0],[1,0,-1],[0,-1,-2]])

    # Get size of Gradient masks
    mask_size = g0.shape[0]

    # Set buffer size for boundary pixels
    buffer = mask_size // 2

    # Initialize 2D array to hold each output image of gradient operation on the input image using each masks

    h0 = np.zeros((img_h,img_w), dtype=np.float32)
    h1 = np.zeros((img_h,img_w), dtype=np.float32)
    h2 = np.zeros((img_h,img_w), dtype=np.float32)
    h3 = np.zeros((img_h,img_w), dtype=np.float32)

    # Apply Gradient operation using the pre-defined gradient masks

    for i in range(0+buffer,img_h-buffer,1):
            for j in range(0+buffer,img_w-buffer,1):

                # Calculate the weighted sum of the pixels in g0 mask
                h0[i][j] = \
                    g0[0][0]*img[i-buffer][j-buffer] \
                    + g0[0][1]*img[i-buffer][j-buffer+1] \
                    + g0[0][2]*img[i-buffer][j-buffer+2] \
                    + g0[1][0]*img[i-buffer+1][j-buffer] \
                    + g0[1][1]*img[i-buffer+1][j-buffer+1] \
                    + g0[1][2]*img[i-buffer+1][j-buffer+2] \
                    + g0[2][0]*img[i-buffer+2][j-buffer] \
                    + g0[2][1]*img[i-buffer+2][j-buffer+1] \
                    + g0[2][2]*img[i-buffer+2][j-buffer+2] \

                # Calculate the weighted sum of the pixels in g1 mask
                h1[i][j] = \
                    g1[0][0]*img[i-buffer][j-buffer] \
                    + g1[0][1]*img[i-buffer][j-buffer+1] \
                    + g1[0][2]*img[i-buffer][j-buffer+2] \
                    + g1[1][0]*img[i-buffer+1][j-buffer] \
                    + g1[1][1]*img[i-buffer+1][j-buffer+1] \
                    + g1[1][2]*img[i-buffer+1][j-buffer+2] \
                    + g1[2][0]*img[i-buffer+2][j-buffer] \
                    + g1[2][1]*img[i-buffer+2][j-buffer+1] \
                    + g1[2][2]*img[i-buffer+2][j-buffer+2] \

                # Calculate the weighted sum of the pixels in g2 mask
                h2[i][j] = \
                    g2[0][0]*img[i-buffer][j-buffer] \
                    + g2[0][1]*img[i-buffer][j-buffer+1] \
                    + g2[0][2]*img[i-buffer][j-buffer+2] \
                    + g2[1][0]*img[i-buffer+1][j-buffer] \
                    + g2[1][1]*img[i-buffer+1][j-buffer+1] \
                    + g2[1][2]*img[i-buffer+1][j-buffer+2] \
                    + g2[2][0]*img[i-buffer+2][j-buffer] \
                    + g2[2][1]*img[i-buffer+2][j-buffer+1] \
                    + g2[2][2]*img[i-buffer+2][j-buffer+2] \

                # Calculate the weighted sum of the pixels in g3 mask
                h3[i][j] = \
                    g3[0][0]*img[i-buffer][j-buffer] \
                    + g3[0][1]*img[i-buffer][j-buffer+1] \
                    + g3[0][2]*img[i-buffer][j-buffer+2] \
                    + g3[1][0]*img[i-buffer+1][j-buffer] \
                    + g3[1][1]*img[i-buffer+1][j-buffer+1] \
                    + g3[1][2]*img[i-buffer+1][j-buffer+2] \
                    + g3[2][0]*img[i-buffer+2][j-buffer] \
                    + g3[2][1]*img[i-buffer+2][j-buffer+1] \
                    + g3[2][2]*img[i-buffer+2][j-buffer+2] \
```

```python
    # Take the absolute value of each response after gradient operation
    h0 = abs(h0)
    h1 = abs(h1)
    h2 = abs(h2)
    h3 = abs(h3)

    # Find the maximum value among the four gradient directions at each location
    max_arr = np.maximum(np.maximum(np.maximum(h0, h1), h2), h3)

    # Find the indices where the maximum value occurs for each gradient direction (0,45,90 or 135 degree)
    h0_indices = np.where(h0 == max_arr)
    h1_indices = np.where(h1 == max_arr)
    h2_indices = np.where(h2 == max_arr)
    h3_indices = np.where(h3 == max_arr)

    # Create an array to store the gradient angle for each pixel
    gradient_angle = np.zeros((img_h,img_w), dtype=np.float32)

    # Merge the indices for each gradient direction into a list
    merged_h0 = [list(row) for row in zip(*h0_indices)]
    merged_h1 = [list(row) for row in zip(*h1_indices)]
    merged_h2 = [list(row) for row in zip(*h2_indices)]
    merged_h3 = [list(row) for row in zip(*h3_indices)]

    # Assign the gradient angle for each pixel based on the maximum gradient direction
    for ind in merged_h0:
        i,j = ind
        gradient_angle[i][j] = 0

    for ind in merged_h1:
        i,j = ind
        gradient_angle[i][j] = 45

    for ind in merged_h2:
        i,j = ind
        gradient_angle[i][j] = 90

    for ind in merged_h3:
        i,j = ind
        gradient_angle[i][j] = 135

    # Normalize the maximum gradient values by dividing by 4
    normalized_array = max_arr/4

    # Set the edge pixels to 0 where part of the mask goes outside of the image border or lies in the undefined region of the image after Gaussian filtering
    normalized_array[:4, :] = 0
    normalized_array[:, :4] = 0
    normalized_array[-4:, :] = 0
    normalized_array[:, -4:] = 0

    gradient_angle[:4, :] = 0
    gradient_angle[:, :4] = 0
    gradient_angle[-4:, :] = 0
    gradient_angle[:, -4:] = 0

    # Show the normalized gradient magnitude image
    im = Image.fromarray(normalized_array).convert('L')
    im.show()

    # Save the normalized gradient magnitude  image in the format '2_Gradient_Magnitude_imagename.bmp'
    im.save('2_Gradient_Magnitude_'+img_file)

    return normalized_array, gradient_angle

# This function performs non-maximum suppression on a gradient magnitude array, based on the gradient angle array.
# The resulting non-maximum suppression (NMS) array and the list of gradient magnitude values after nms excluding 0 are returned.

def non_maxima_suppression(gradient_magnitude_array, gradient_angle_array):

    img = gradient_magnitude_array
    img_h, img_w = img.shape

    # initialize sector matrix
    sector = np.zeros((img_h, img_w))

    # quantize angle into 4 sectors
    for i in range(0,img_h,1):
        for j in range(0,img_w,1):
            if 0 <= gradient_angle_array[i,j] < 22.5 :
                sector[i,j] = 0
            elif 22.5 <= gradient_angle_array[i,j] < 67.5 :
                sector[i,j] = 1
            elif 67.5 <= gradient_angle_array[i,j] < 112.5 :
                sector[i,j] = 2
            elif 112.5 <= gradient_angle_array[i,j] < 157.5 :
                sector[i,j] = 3
            elif 157.5 <= gradient_angle_array[i,j] < 202.5 :
                sector[i,j] = 0
            elif 202.5 <= gradient_angle_array[i,j] < 247.5 :
                sector[i,j] = 1
            elif 247.5 <= gradient_angle_array[i,j] < 292.5 :
                sector[i,j] = 2
            elif 292.5 <= gradient_angle_array[i,j] < 337.5 :
                sector[i,j] = 3
            elif 337.5 <= gradient_angle_array[i,j] <= 360:
                sector[i,j] = 0
            else:
                sector[i,j] = -1

    # define buffer
    buffer = 4+1
```

```
    # initialize nms and magnitude_arr (for percentile calculation later)
    nms = np.zeros((img_h, img_w))
    magnitude_arr = []

    # performing non maxima suppression by comparing gradient magnitudes according to quantized sectors
    for i in range(0+buffer,img_h-buffer,1):
        for j in range(0+buffer,img_w-buffer,1):
            if sector[i,j] == 2:
                # compare to upper and lower magnitudes
                if ( gradient_magnitude_array[i][j] > gradient_magnitude_array[i-1][j] ) \
                    and ( gradient_magnitude_array[i][j] > gradient_magnitude_array[i+1][j] ) :
                    nms[i,j] = gradient_magnitude_array[i][j]
                    magnitude_arr.append(gradient_magnitude_array[i][j])
                else :
                    nms[i,j] = 0
            elif sector[i,j] == 3:
                # compare to upper left and lower right mag
                if ( gradient_magnitude_array[i][j] > gradient_magnitude_array[i-1][j-1] ) \
                    and ( gradient_magnitude_array[i][j] > gradient_magnitude_array[i+1][j+1] ) :
                    nms[i,j] = gradient_magnitude_array[i][j]
                    magnitude_arr.append(gradient_magnitude_array[i][j])
                else :
                    nms[i,j] = 0
            elif sector[i,j] == 0:
                # compare to right and left mag
                if ( gradient_magnitude_array[i][j] > gradient_magnitude_array[i][j-1] ) \
                    and ( gradient_magnitude_array[i][j] > gradient_magnitude_array[i][j+1] ) :
                    nms[i,j] = gradient_magnitude_array[i][j]
                    magnitude_arr.append(gradient_magnitude_array[i][j])
                else :
                    nms[i,j] = 0
            elif sector[i,j] == 1 :
                # compare to upper right and lower left mag
                if ( gradient_magnitude_array[i][j] > gradient_magnitude_array[i-1][j+1] ) \
                    and ( gradient_magnitude_array[i][j] > gradient_magnitude_array[i+1][j-1] ) :
                    nms[i,j] = gradient_magnitude_array[i][j]
                    magnitude_arr.append(gradient_magnitude_array[i][j])
                else :
                    nms[i,j] = 0
            elif sector[i,j] == -1:
                # suppress to zero
                nms[i,j] = 0
    # show nms
    im = Image.fromarray(nms).convert("L")
    im.show()

    # save nms (4)
    im.save('3_NMS_'+img_file)

    return nms, magnitude_arr

#This function applies thresholding to the non-maximum suppression (NMS) array and generates three binary edge maps
#with edge pixels set on three different thresholds from 25th, 50th and 75th percentile of gradient magnitude array after nms excluding 0.

def thresholding(nms_array, magnitude_arr):

    img_h, img_w = nms_array.shape

    # calculate thresholds from percentiles
    T1 = np.percentile(magnitude_arr,25)
    T2 = np.percentile(magnitude_arr,50)
    T3 = np.percentile(magnitude_arr,75)

    # initialize final threshold images
    final_threshold_img_t1 = np.zeros((img_h, img_w))
    final_threshold_img_t2 = np.zeros((img_h, img_w))
    final_threshold_img_t3 = np.zeros((img_h, img_w))

    # apply threshold and generate binary edge map
    for i in range(0,img_h,1):
        for j in range(0,img_w,1):
            if nms_array[i][j] >=T1:
                final_threshold_img_t1[i][j] = 255
            if nms_array[i][j] >=T2:
                final_threshold_img_t2[i][j] = 255
            if nms_array[i][j] >=T3:
                final_threshold_img_t3[i][j] = 255

    # show final image T1
    im = Image.fromarray(final_threshold_img_t1).convert('1')
    im.show()

    # save final image T1 (5)
    im.save('4_Binary_Edge_Map_T1_'+img_file)

    # show final image T2
    im = Image.fromarray(final_threshold_img_t2).convert('1')
    im.show()

    # save final image T2 (5)
    im.save('5_Binary_Edge_Map_T2_'+img_file)

    # show final image T3
    im = Image.fromarray(final_threshold_img_t3).convert('1')
    im.show()

    # save final image T3 (5)
    im.save('6_Binary_Edge_Map_T3_'+img_file)

    return final_threshold_img_t1, final_threshold_img_t2, final_threshold_img_t3

#This function takes in nms array as input and generates a histogram of the gradient magnitudes after nms.
```

```python
def nms_histogram(nms_array):

    max_pixels = 512*512

    gray_image = np.random.choice(nms_array.flatten(), size=max_pixels)

    # add axis labels and title
    plt.xlabel('Gradient Magnitudes after NMS')
    plt.ylabel('No. of Pixels')
    plt.title('Histogram of Gradient values after NMS')

    # create histogram plot with bins as 256
    plt.hist(nms_array.flatten(), bins=256, range=(0, 255), color='gray')

    #save histogram image (8)
    plt.savefig('7_nms_histogram_'+os.path.splitext(img_file)[0] + '.png')

    # display the plot
    plt.show()


# ensure proper arguments given
if (len(sys.argv)) < 2:
    print("Command failure. Usage: $ python3 canny.py [image_file_name].bmp")
    exit()

# show input image from parameter passed
img_file = sys.argv[1]

try:
    img = Image.open(img_file).convert('L')
    img.show()
except:
    print("Error loading image")
    sys.exit(1)

# convert list to numpy array
input_img = np.array(img)

# compute input dimentions
height, width = input_img.shape

# canny edge detection steps

# 1. Gaussian smoothing
smooth_image = gaussian_smoothing(img_file)
# 2. Gradient Operation
gradient_magnitude, gradient_angle  = gradient_operation(smooth_image)
# 3. NMS
nms, magnitude_arr = non_maxima_suppression(gradient_magnitude, gradient_angle)
# 4. Thresholding
final_threshold_img_t1, final_threshold_img_t2, final_threshold_img_t3 = thresholding(nms, magnitude_arr)
#5 . Histogram
nms_histogram(nms)
```

Methodology

1. **Gaussian Smoothing**

   Gaussian smoothing using a predefined 7×7 Gaussian mask was used to reduce noise and to blur images. The mask was applied to each pixel of the image using a nested loop that performed a convolution operation to calculate its smoothed value. The center of the mask was used as the reference center. For parts of the Gaussian mask going outside of the image border, the output image was undefined (undefined values were replaced with 0 in the output image). The resulting value after the convolution operation was stored in the 'smooth_image' array at the corresponding pixel position. Normalization was then performed by dividing the results of the 'smooth_image' array by the sum of the entries (= 140 for the given mask) at each pixel location. The resulting normalized 'smooth_image' array was returned as the output of the function.

2. **Gradient Operation:**

   For gradient operation, predefined masks were used to compute gradients at 0, 45, 90 and 135 degree. The output value was undefined if part of the 3×3 mask goes outside of the image border or lies in the undefined region of the image after Gaussian filtering. The gradient magnitude value responses from all four masks after convolution was compared and the maximum of the absolute values of the responses was stored in the max_arr. Max_arr was then divided by 4 to return the normalized edge magnitude array. The indices of the maximum value for each map are recorded and used to determine the gradient angle of the edges at each pixel position. Thus, the output of the function is the normalized edge magnitude array and the gradient angle array.

3. **Non-Maxima Suppression:**

   Non-maxima suppression is used to suppress non-maximum values in a gradient image and retain only the local maximum values, which correspond to edges or other significant image features. In the function, the gradient angles are quantized into four sectors and stored in the sector array. The gradient magnitudes in the gradient_magnitude_array are compared according to the sector array and non-maximum values are suppressed retaining only gradient maximas. The resulting array of suppressed values is returned as nms, along with an array of all gradient magnitudes after nms excluding 0 values for later percentile calculation.

4. **Thresholding:**

   Thresholding is then used to convert this NMS image into a binary image by setting a threshold value above which a pixel is considered an edge pixel, and below which it is not. In our code, three different threshold values are calculated based on the 25^{th}, 50^{th}, and 75^{th} percentiles of the gradient magnitude image after nms excluding 0 values. The function loops through each pixel of nms_array and checks if the value of the pixel is greater than or equal to each of the three thresholds. If it is, the corresponding pixel in the appropriate threshold image is set to 255 (white), indicating that it is an edge pixel. If it is not, the pixel is set to 0 (black), indicating that it is not an edge pixel. The function returns three thresholded images, each with edge pixels set based on a different percentile threshold.

Barbara.bmp

- Threshold value for 25^{th} percentile: 2.851
- Threshold value for 50^{th} percentile: 6.446
- Threshold value for 75^{th} percentile: 14.023

Peppers.bmp

- Threshold value for 25^{th} percentile: 1.657
- Threshold value for50^{th} percentile: 3.437
- Threshold value for 75^{th} percentile: 9.661

Goldhill.bmp

- Threshold value for 25^{th} percentile: 3.554
- Threshold value for 50^{th} percentile: 7.073
- Threshold value for 75^{th} percentile: 13.905

5. **Histogram:**

The histogram of the normalized edge magnitude image after non-maxima suppression can be used to get an idea of the distribution of the gradient values after NMS and can be useful in determining appropriate values for thresholding. We've explicitly limited x and y ranges for better visualization of the thresholds. Gradient values in x axis are chosen in the range of (1,maximum gradient value after nms). The vertical red, green and blue lines ($x = T1, T2, T3$) correspond to the threshold values at 25^{th}, 50^{th} and 75^{th} percentile in the gradient magnitude array.

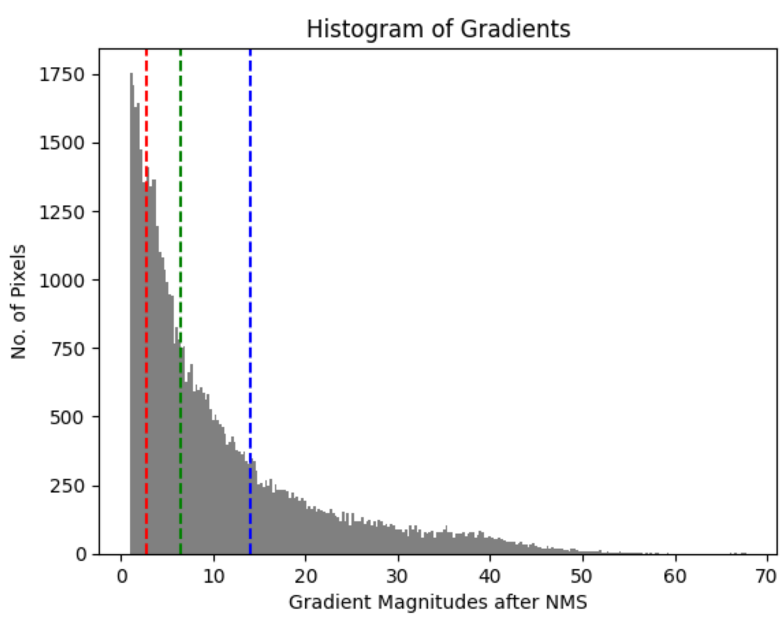

Figure 1: Barabara.bmp: Threshold Visualization Histogram

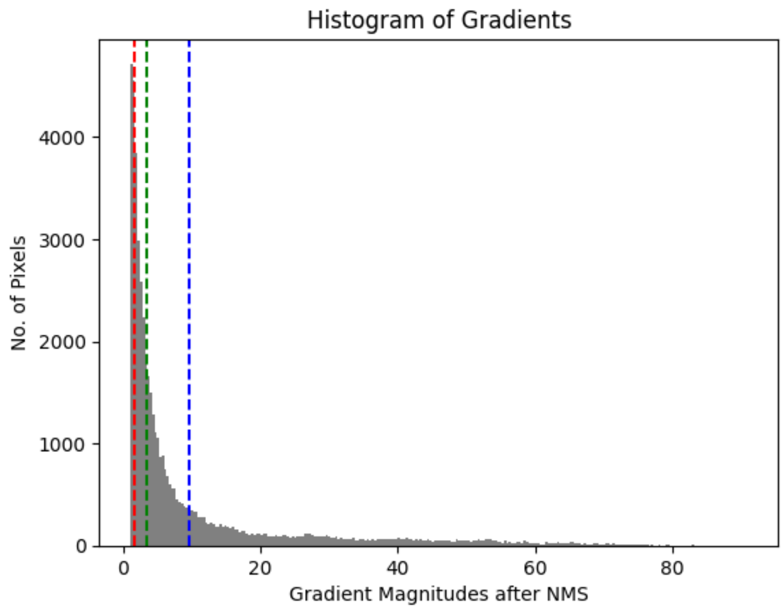

Figure 2: Peppers.bmp: Threshold Visualization Histogram

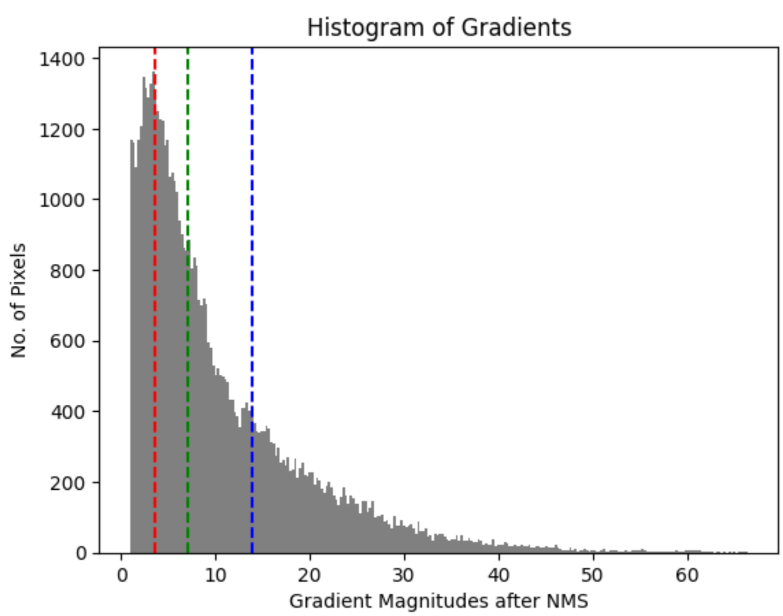

Figure 3: Goldhill.bmp: Threshold Visualization Histogram

Results

Input image: barbara.bmp

Figure 4: Original Image - barbara.bmp

Figure 5: Gaussian Smoothed Image

Figure 6: Gradient Magnitude Image

Figure 7: Non-Maxima Suppressed Image

Figure 8: Binary Edge Map: Thresholded at $25^{th} Percentile (T1 = 2.851)$

Figure 9: Binary Edge Map: Thresholded at $50^{th} Percentile (T2 = 6.446)$

Figure 10: Binary Edge Map: Thresholded at $75^{th} Percentile (T3 = 14.023)$

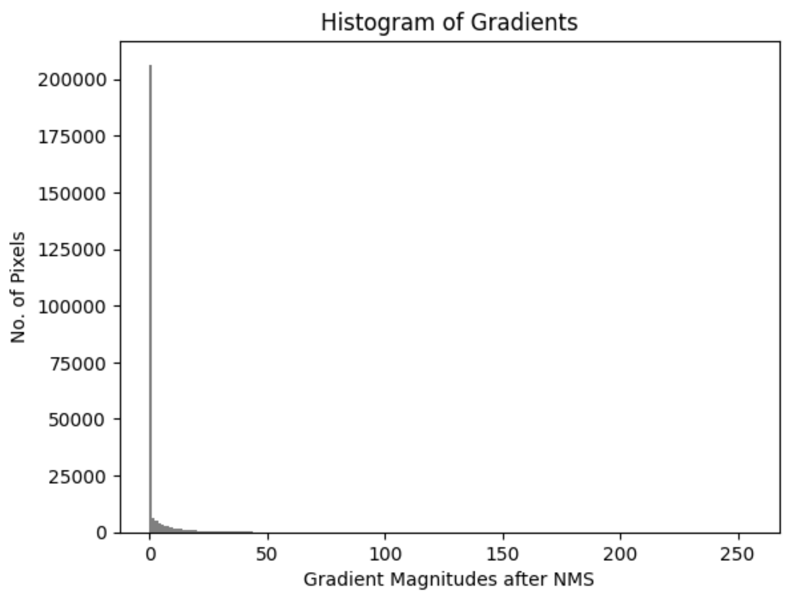

Figure 11: Histogram: Number of Pixels vs Non-Maxima Suppressed Gradient Magnitudes

Input image: Peppers.bmp

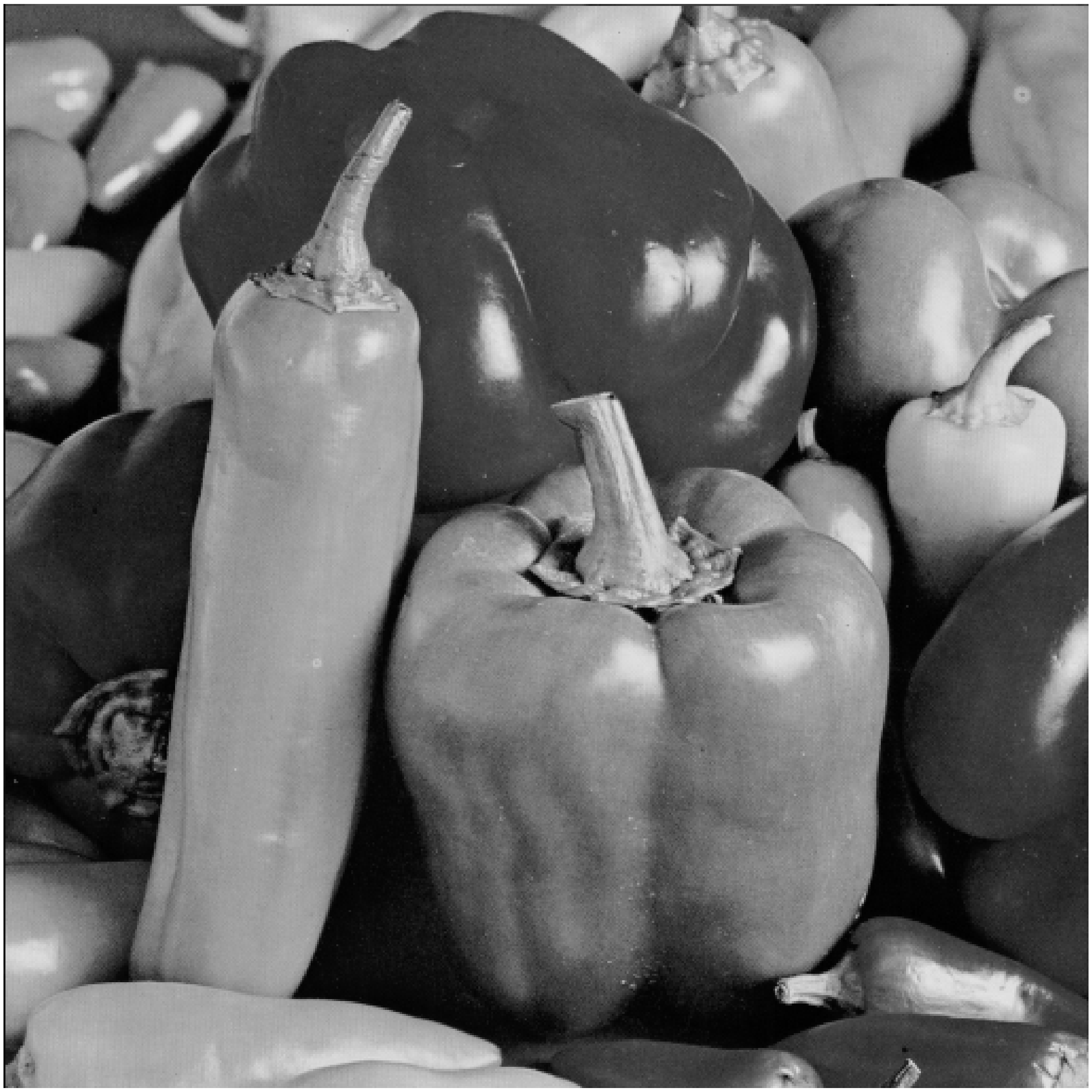

Figure 12: Original Image - Peppers.bmp

Figure 13: Gaussian Smoothed Image

Figure 14: Gradient Magnitude Image

Figure 15: Non-Maxima Suppressed Image

Figure 16: Binary Edge Map: Thresholded at $25^{th} Percentile (T1 = 1.657)$

Figure 17: Binary Edge Map: Thresholded at $50^{th} Percentile (T2 = 3.437)$

Figure 18: Binary Edge Map: Thresholded at $75^{th} Percentile (T3 = 9.661)$

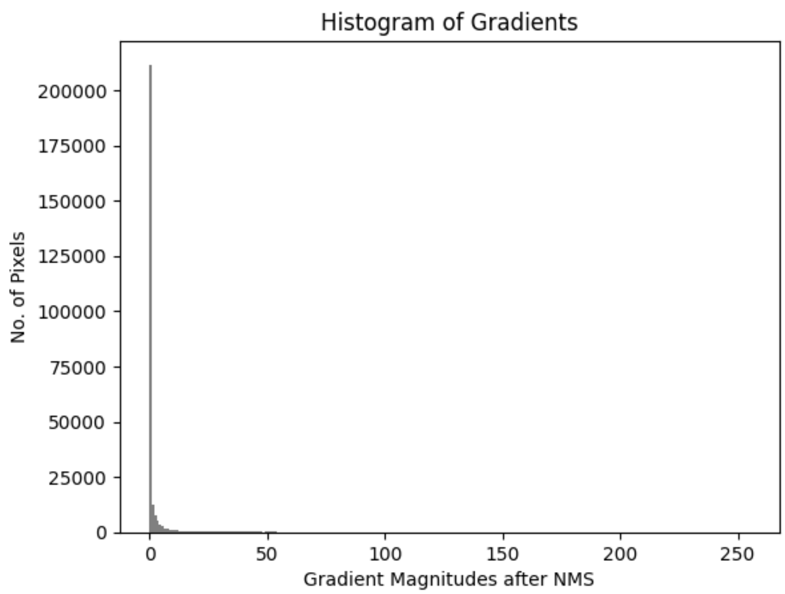

Figure 19: Histogram: Number of Pixels vs Non-Maxima Suppressed Gradient Magnitudes

Input image: Goldhill.bmp

Figure 20: Original Image - Goldhill.bmp

Figure 21: Gaussian Smoothed Image

Figure 22: Gradient Magnitude Image

Figure 23: Non-Maxima Suppressed Image

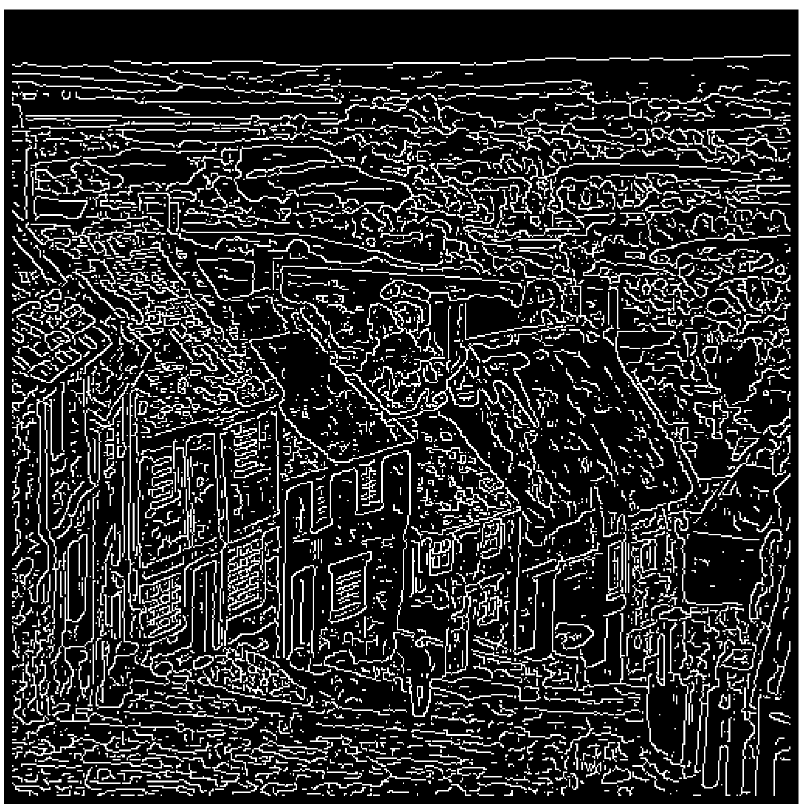

Figure 24: Binary Edge Map: Thresholded at $25^{th} Percentile (T1 = 3.554)$

Figure 25: Binary Edge Map: Thresholded at $50^{th} Percentile (T2 = 7.073)$

Figure 26: Binary Edge Map: Thresholded at $75^{th} Percentile (T3 = 13.905)$

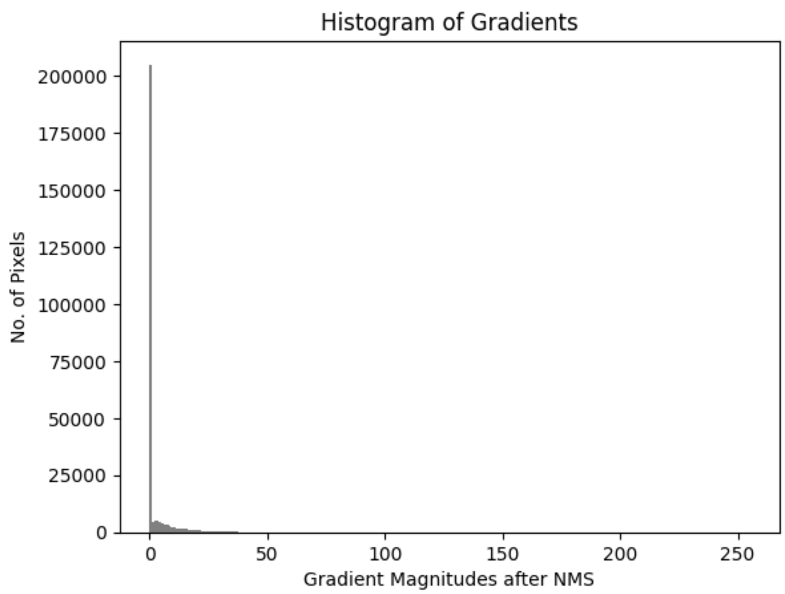

Figure 27: Histogram: Number of Pixels vs Non-Maxima Suppressed Gradient Magnitudes