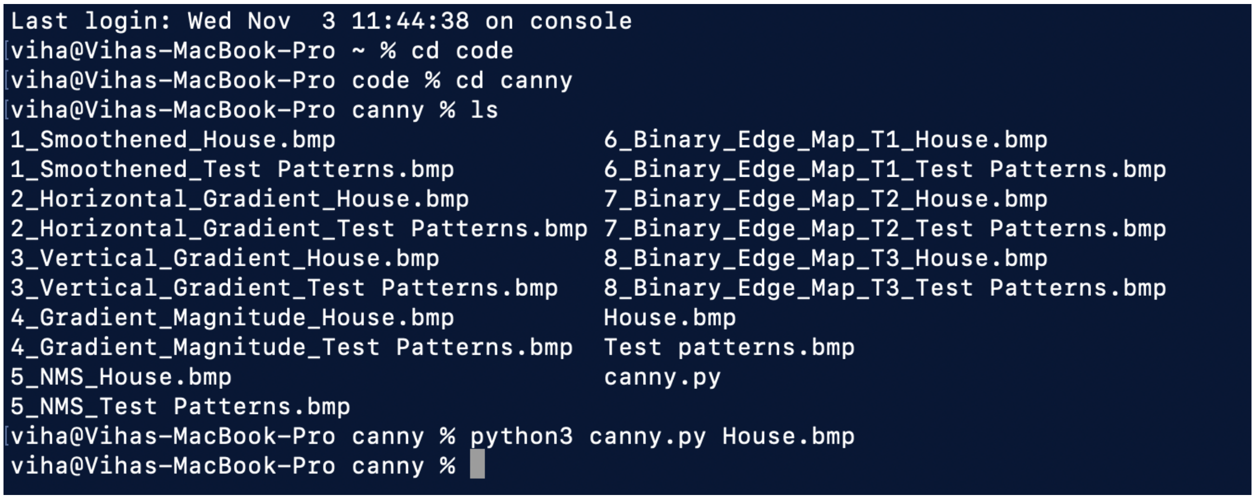
vg2237
sys8910

Project 1: Canny Edge Detector

Submitted by:
Viha Gupta - vg2237
Suyash Soniminde - sys8910

Instructions:

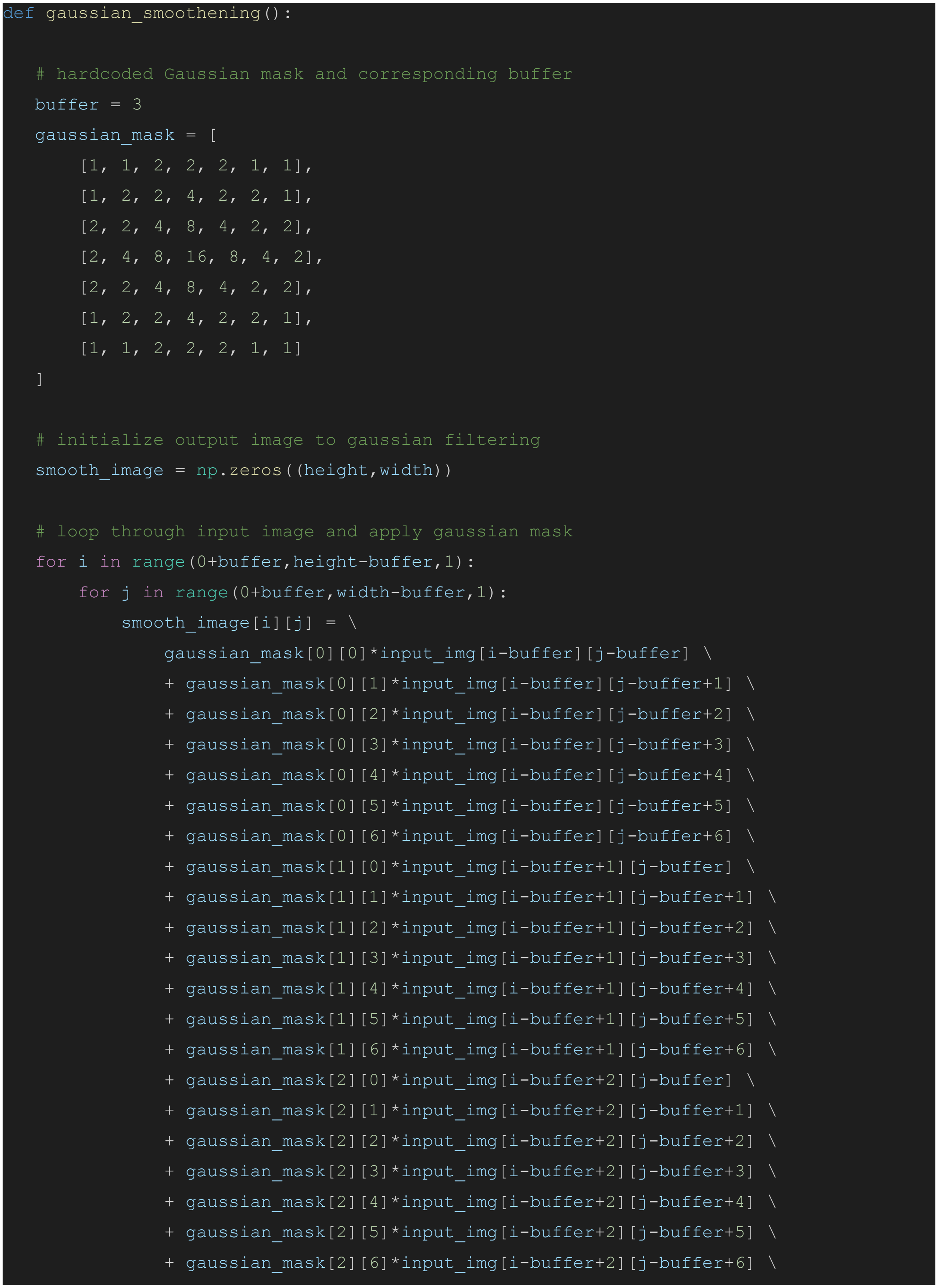
1. Ensure you have python3 installed on your system
2. Save the python script and input images to a directory on your computer
3. Open a terminal window and cd into the above directory
4. Execute the python script by passing 2 parameters: the script name and the input image
   For example:
   $ *python3 canny.py House.bmp*
   $ *python3 canny.py 'Test patterns.bmp'*
5. The 8 output images will be saved in your current directory



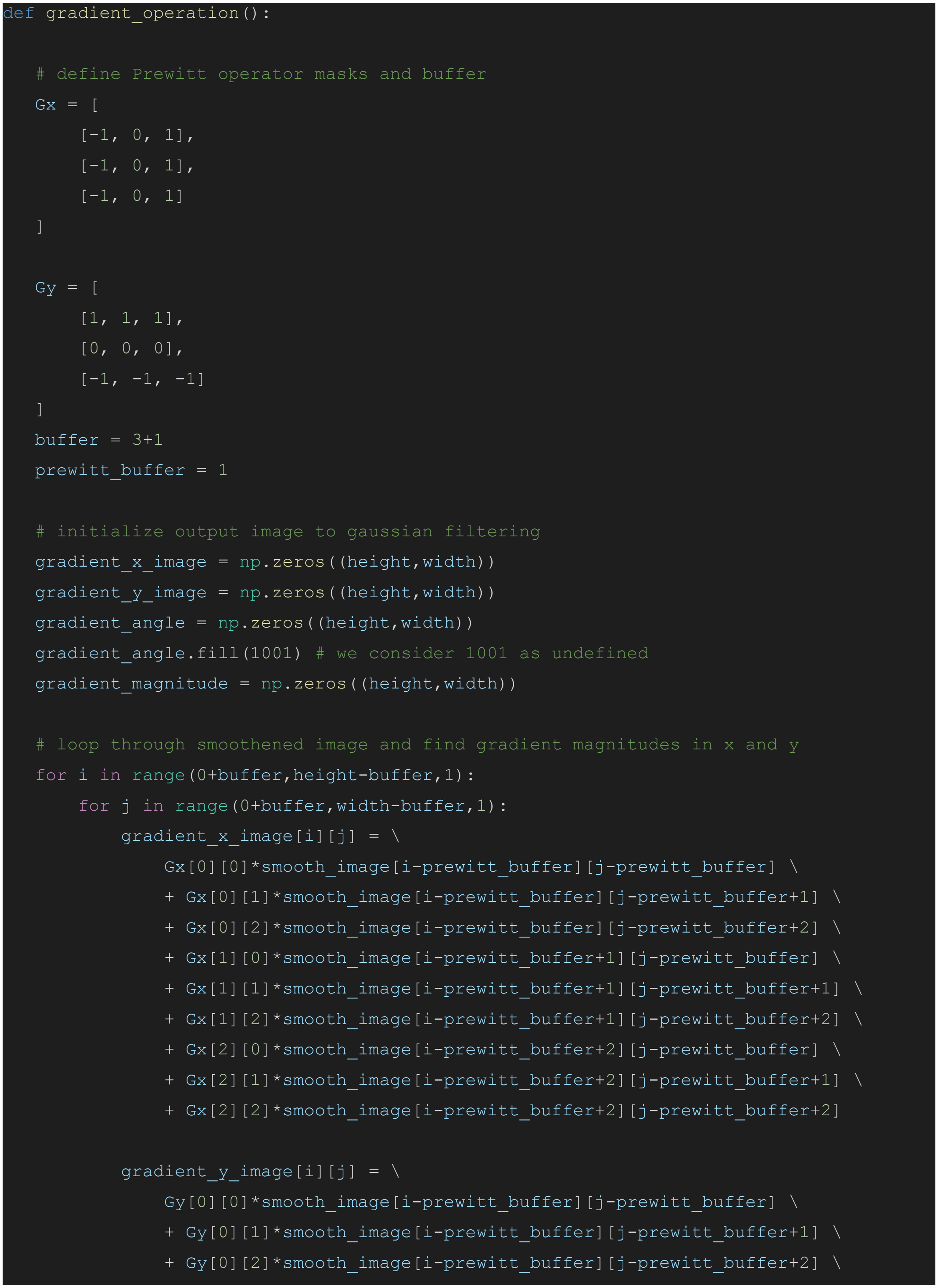
```
Last login: Wed Nov  3 11:44:38 on console
[viha@Vihas-MacBook-Pro ~ % cd code
[viha@Vihas-MacBook-Pro code % cd canny
[viha@Vihas-MacBook-Pro canny % ls
1_Smoothened_House.bmp                  6_Binary_Edge_Map_T1_House.bmp
1_Smoothened_Test Patterns.bmp          6_Binary_Edge_Map_T1_Test Patterns.bmp
2_Horizontal_Gradient_House.bmp         7_Binary_Edge_Map_T2_House.bmp
2_Horizontal_Gradient_Test Patterns.bmp 7_Binary_Edge_Map_T2_Test Patterns.bmp
3_Vertical_Gradient_House.bmp           8_Binary_Edge_Map_T3_House.bmp
3_Vertical_Gradient_Test Patterns.bmp   8_Binary_Edge_Map_T3_Test Patterns.bmp
4_Gradient_Magnitude_House.bmp          House.bmp
4_Gradient_Magnitude_Test Patterns.bmp  Test patterns.bmp
5_NMS_House.bmp                         canny.py
5_NMS_Test Patterns.bmp
[viha@Vihas-MacBook-Pro canny % python3 canny.py House.bmp
viha@Vihas-MacBook-Pro canny %
```

Source Code:

```
import numpy as np
from PIL import Image
import math
import sys

'''
Submitted by:
Viha Gupta vg2237
Suyash Soniminde sys8910
'''
```

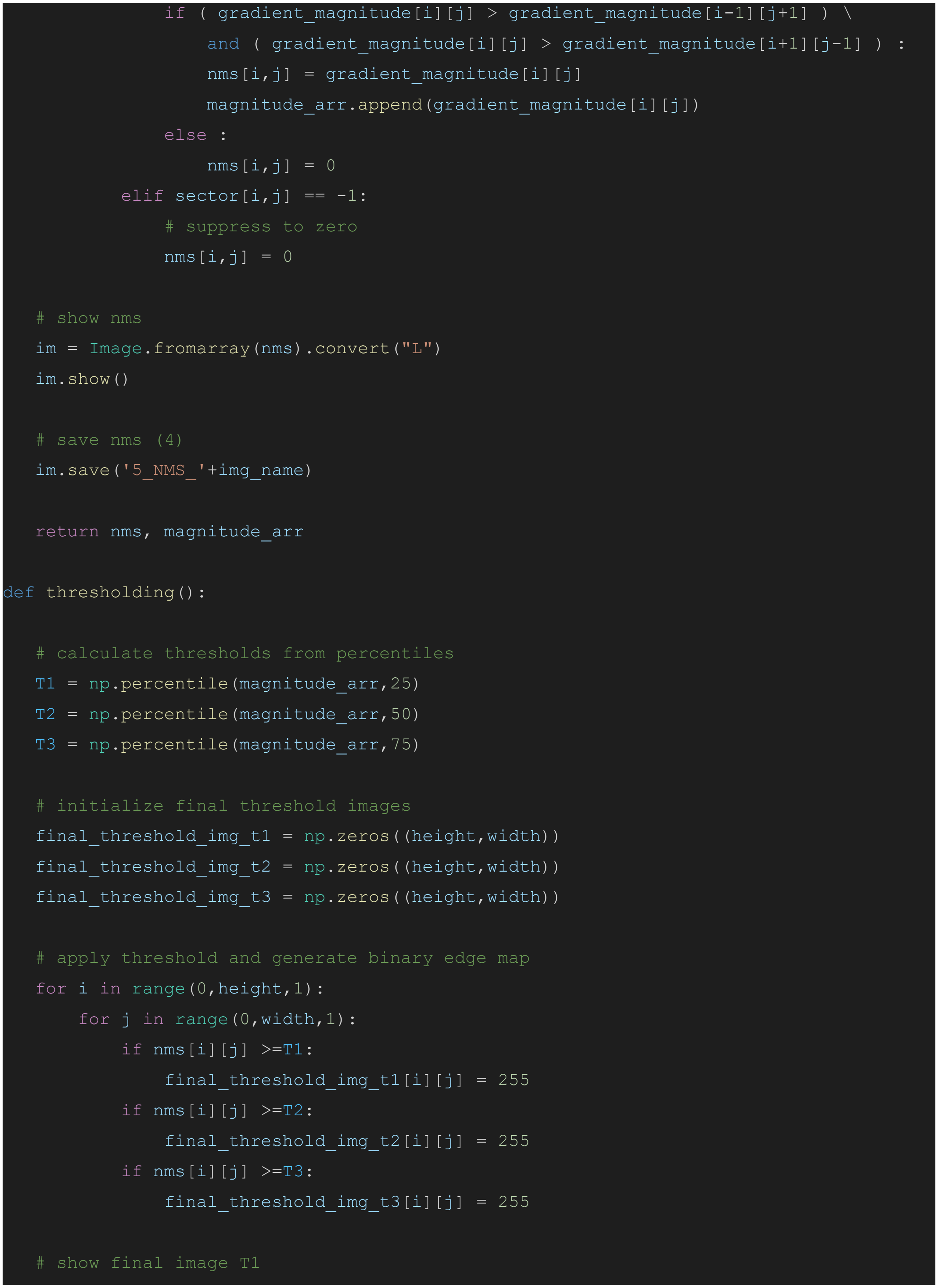


```python
def gaussian_smoothening():

    # hardcoded Gaussian mask and corresponding buffer
    buffer = 3
    gaussian_mask = [
        [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]
    ]

    # initialize output image to gaussian filtering
    smooth_image = np.zeros((height,width))

    # loop through input image and apply gaussian mask
    for i in range(0+buffer,height-buffer,1):
        for j in range(0+buffer,width-buffer,1):
            smooth_image[i][j] = \
                gaussian_mask[0][0]*input_img[i-buffer][j-buffer] \
                + gaussian_mask[0][1]*input_img[i-buffer][j-buffer+1] \
                + gaussian_mask[0][2]*input_img[i-buffer][j-buffer+2] \
                + gaussian_mask[0][3]*input_img[i-buffer][j-buffer+3] \
                + gaussian_mask[0][4]*input_img[i-buffer][j-buffer+4] \
                + gaussian_mask[0][5]*input_img[i-buffer][j-buffer+5] \
                + gaussian_mask[0][6]*input_img[i-buffer][j-buffer+6] \
                + gaussian_mask[1][0]*input_img[i-buffer+1][j-buffer] \
                + gaussian_mask[1][1]*input_img[i-buffer+1][j-buffer+1] \
                + gaussian_mask[1][2]*input_img[i-buffer+1][j-buffer+2] \
                + gaussian_mask[1][3]*input_img[i-buffer+1][j-buffer+3] \
                + gaussian_mask[1][4]*input_img[i-buffer+1][j-buffer+4] \
                + gaussian_mask[1][5]*input_img[i-buffer+1][j-buffer+5] \
                + gaussian_mask[1][6]*input_img[i-buffer+1][j-buffer+6] \
                + gaussian_mask[2][0]*input_img[i-buffer+2][j-buffer] \
                + gaussian_mask[2][1]*input_img[i-buffer+2][j-buffer+1] \
                + gaussian_mask[2][2]*input_img[i-buffer+2][j-buffer+2] \
                + gaussian_mask[2][3]*input_img[i-buffer+2][j-buffer+3] \
                + gaussian_mask[2][4]*input_img[i-buffer+2][j-buffer+4] \
                + gaussian_mask[2][5]*input_img[i-buffer+2][j-buffer+5] \
                + gaussian_mask[2][6]*input_img[i-buffer+2][j-buffer+6] \
                + gaussian_mask[3][0]*input_img[i-buffer+3][j-buffer] \
                + gaussian_mask[3][1]*input_img[i-buffer+3][j-buffer+1] \
                + gaussian_mask[3][2]*input_img[i-buffer+3][j-buffer+2] \
                + gaussian_mask[3][3]*input_img[i-buffer+3][j-buffer+3] \
                + gaussian_mask[3][4]*input_img[i-buffer+3][j-buffer+4] \
                + gaussian_mask[3][5]*input_img[i-buffer+3][j-buffer+5] \
                + gaussian_mask[3][6]*input_img[i-buffer+3][j-buffer+6] \
                + gaussian_mask[4][0]*input_img[i-buffer+4][j-buffer] \
                + gaussian_mask[4][1]*input_img[i-buffer+4][j-buffer+1] \
                + gaussian_mask[4][2]*input_img[i-buffer+4][j-buffer+2] \
                + gaussian_mask[4][3]*input_img[i-buffer+4][j-buffer+3] \
                + gaussian_mask[4][4]*input_img[i-buffer+4][j-buffer+4] \
                + gaussian_mask[4][5]*input_img[i-buffer+4][j-buffer+5] \
                + gaussian_mask[4][6]*input_img[i-buffer+4][j-buffer+6] \
                + gaussian_mask[5][0]*input_img[i-buffer+5][j-buffer] \
                + gaussian_mask[5][1]*input_img[i-buffer+5][j-buffer+1] \
                + gaussian_mask[5][2]*input_img[i-buffer+5][j-buffer+2] \
                + gaussian_mask[5][3]*input_img[i-buffer+5][j-buffer+3] \
                + gaussian_mask[5][4]*input_img[i-buffer+5][j-buffer+4] \
                + gaussian_mask[5][5]*input_img[i-buffer+5][j-buffer+5] \
                + gaussian_mask[5][6]*input_img[i-buffer+5][j-buffer+6] \
                + gaussian_mask[6][0]*input_img[i-buffer+6][j-buffer] \
                + gaussian_mask[6][1]*input_img[i-buffer+6][j-buffer+1] \
                + gaussian_mask[6][2]*input_img[i-buffer+6][j-buffer+2] \
                + gaussian_mask[6][3]*input_img[i-buffer+6][j-buffer+3] \
                + gaussian_mask[6][4]*input_img[i-buffer+6][j-buffer+4] \
                + gaussian_mask[6][5]*input_img[i-buffer+6][j-buffer+5] \
                + gaussian_mask[6][6]*input_img[i-buffer+6][j-buffer+6] \


    # normalize the image by dividing by 140
    smooth_image = smooth_image/140

    # show smooth image
    im = Image.fromarray(smooth_image).convert('L')
    im.show()

    # save smooth_image (1)
    im.save('1_Smoothened_'+img_name)

    return smooth_image
```



```
def gradient_operation():

    # define Prewitt operator masks and buffer
    Gx = [
        [-1, 0, 1],
        [-1, 0, 1],
        [-1, 0, 1]
    ]

    Gy = [
        [1, 1, 1],
        [0, 0, 0],
        [-1, -1, -1]
    ]
    buffer = 3+1
    prewitt_buffer = 1

    # initialize output image to gaussian filtering
    gradient_x_image = np.zeros((height,width))
    gradient_y_image = np.zeros((height,width))
    gradient_angle = np.zeros((height,width))
    gradient_angle.fill(1001) # we consider 1001 as undefined
    gradient_magnitude = np.zeros((height,width))

    # loop through smoothened image and find gradient magnitudes in x and y
    for i in range(0+buffer,height-buffer,1):
        for j in range(0+buffer,width-buffer,1):
            gradient_x_image[i][j] = \
                Gx[0][0]*smooth_image[i-prewitt_buffer][j-prewitt_buffer] \
                + Gx[0][1]*smooth_image[i-prewitt_buffer][j-prewitt_buffer+1] \
                + Gx[0][2]*smooth_image[i-prewitt_buffer][j-prewitt_buffer+2] \
                + Gx[1][0]*smooth_image[i-prewitt_buffer+1][j-prewitt_buffer] \
                + Gx[1][1]*smooth_image[i-prewitt_buffer+1][j-prewitt_buffer+1] \
                + Gx[1][2]*smooth_image[i-prewitt_buffer+1][j-prewitt_buffer+2] \
                + Gx[2][0]*smooth_image[i-prewitt_buffer+2][j-prewitt_buffer] \
                + Gx[2][1]*smooth_image[i-prewitt_buffer+2][j-prewitt_buffer+1] \
                + Gx[2][2]*smooth_image[i-prewitt_buffer+2][j-prewitt_buffer+2]

            gradient_y_image[i][j] = \
                Gy[0][0]*smooth_image[i-prewitt_buffer][j-prewitt_buffer] \
                + Gy[0][1]*smooth_image[i-prewitt_buffer][j-prewitt_buffer+1] \
                + Gy[0][2]*smooth_image[i-prewitt_buffer][j-prewitt_buffer+2] \
            + Gy[1][0]*smooth_image[i-prewitt_buffer+1][j-prewitt_buffer] \
            + Gy[1][1]*smooth_image[i-prewitt_buffer+1][j-prewitt_buffer+1] \
            + Gy[1][2]*smooth_image[i-prewitt_buffer+1][j-prewitt_buffer+2] \
            + Gy[2][0]*smooth_image[i-prewitt_buffer+2][j-prewitt_buffer] \
            + Gy[2][1]*smooth_image[i-prewitt_buffer+2][j-prewitt_buffer+1] \
            + Gy[2][2]*smooth_image[i-prewitt_buffer+2][j-prewitt_buffer+2]

# first take abs value
gradient_x_image = abs(gradient_x_image)
gradient_y_image = abs(gradient_y_image)

# then normalize by dividing by 3
gradient_x_image = gradient_x_image/3
gradient_y_image = gradient_y_image/3

# calculate gradient magnitude and angle
for i in range(0+buffer,height-buffer,1):
    for j in range(0+buffer,width-buffer,1):
        if gradient_x_image[i][j] == 0:
            gradient_angle[i][j] = 1001
        else :
            gradient_angle[i][j] = \
            gradient_y_image[i][j]/gradient_x_image[i][j]
        if gradient_angle[i][j] < 0 :
            gradient_angle[i][j] = gradient_angle[i][j] + 360

        gradient_magnitude[i][j] = \
            math.sqrt(gradient_x_image[i][j] * gradient_x_image[i][j] \
                + gradient_y_image[i][j] * gradient_y_image[i][j])

# normalize gradient magnitude to [0-255]
nmz = np.max(gradient_magnitude)/255
gradient_magnitude = gradient_magnitude/nmz

# show horizontal gradient
im = Image.fromarray(gradient_x_image).convert('L')
im.show()

# save horizontal gradient images (2)
im.save('2_Horizontal_Gradient_'+img_name)

# show vertical gradient
```

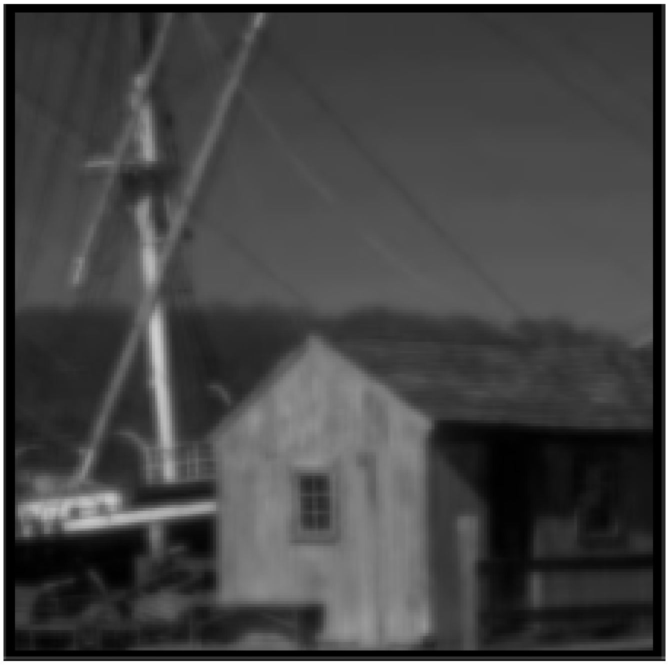
```
    im = Image.fromarray(gradient_y_image).convert('L')
    im.show()

    # save vertical gradient images (2)
    im.save('3_Vertical_Gradient_'+img_name)

    # show gradient magnitude
    im = Image.fromarray(gradient_magnitude).convert('L')
    im.show()

    # save gradient magnitude image (3)
    im.save('4_Gradient_Magnitude_'+img_name)

    return gradient_angle, gradient_magnitude

def non_maxima_suppression():

    # initialize sector matrix
    sector = np.zeros((height,width))

    # quantize angle into 4 sectors
    for i in range(0,height,1):
        for j in range(0,width,1):
            if 0 <= gradient_angle[i,j] < 22.5 :
                sector[i,j] = 0
            elif 22.5 <= gradient_angle[i,j] < 67.5 :
                sector[i,j] = 1
            elif 67.5 <= gradient_angle[i,j] < 112.5 :
                sector[i,j] = 2
            elif 112.5 <= gradient_angle[i,j] < 157.5 :
                sector[i,j] = 3
            elif 157.5 <= gradient_angle[i,j] < 202.5 :
                sector[i,j] = 0
            elif 202.5 <= gradient_angle[i,j] < 247.5 :
                sector[i,j] = 1
            elif 247.5 <= gradient_angle[i,j] < 292.5 :
                sector[i,j] = 2
            elif 292.5 <= gradient_angle[i,j] < 337.5 :
                sector[i,j] = 3
            elif 337.5 <= gradient_angle[i,j] <= 360:
                sector[i,j] = 0
            # special case for when gradient x = 0 and gradient angle = 1001 ie undefined case
            elif gradient_angle[i,j] > 1000:
                sector[i,j] = -1

    # define buffer
    buffer = 4+1

    # initialize nms and magnitude_arr (for percentile calculation later)
    nms = np.zeros((height,width))
    magnitude_arr = []

    # performing non maxima suppression by comparing gradient magnitudes according to quantized sectors
    for i in range(0+buffer,height-buffer,1):
        for j in range(0+buffer,width-buffer,1):
            if sector[i,j] == 0:
                # compare to right and left magnitudes
                if ( gradient_magnitude[i][j] > gradient_magnitude[i-1][j] ) \
                    and ( gradient_magnitude[i][j] > gradient_magnitude[i+1][j] ) :
                    nms[i,j] = gradient_magnitude[i][j]
                    magnitude_arr.append(gradient_magnitude[i][j])
                else :
                    nms[i,j] = 0
            elif sector[i,j] == 1:
                # compare to upper right and lower left mag
                if ( gradient_magnitude[i][j] > gradient_magnitude[i-1][j-1] ) \
                    and ( gradient_magnitude[i][j] > gradient_magnitude[i+1][j+1] ) :
                    nms[i,j] = gradient_magnitude[i][j]
                    magnitude_arr.append(gradient_magnitude[i][j])
                else :
                    nms[i,j] = 0
            elif sector[i,j] == 2:
                # compare to upper and lower mag
                if ( gradient_magnitude[i][j] > gradient_magnitude[i][j-1] ) \
                    and ( gradient_magnitude[i][j] > gradient_magnitude[i][j+1] ) :
                    nms[i,j] = gradient_magnitude[i][j]
                    magnitude_arr.append(gradient_magnitude[i][j])
                else :
                    nms[i,j] = 0
            elif sector[i,j] == 3:
                # compare to upper left and lower right mag
```



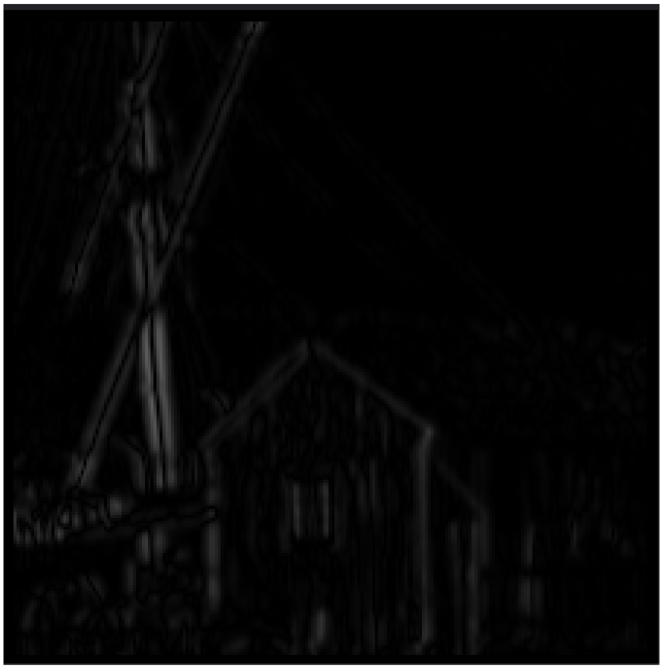
```
                if ( gradient_magnitude[i][j] > gradient_magnitude[i-1][j+1] ) \
                    and ( gradient_magnitude[i][j] > gradient_magnitude[i+1][j-1] ) :
                    nms[i,j] = gradient_magnitude[i][j]
                    magnitude_arr.append(gradient_magnitude[i][j])
                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('5_NMS_'+img_name)

    return nms, magnitude_arr

def thresholding():

    # 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((height,width))
    final_threshold_img_t2 = np.zeros((height,width))
    final_threshold_img_t3 = np.zeros((height,width))

    # apply threshold and generate binary edge map
    for i in range(0,height,1):
        for j in range(0,width,1):
            if nms[i][j] >=T1:
                final_threshold_img_t1[i][j] = 255
            if nms[i][j] >=T2:
                final_threshold_img_t2[i][j] = 255
            if nms[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('6_Binary_Edge_Map_T1_'+img_name)

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

    # save final image T2 (5)
    im.save('7_Binary_Edge_Map_T2_'+img_name)

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

    # save final image T3 (5)
    im.save('8_Binary_Edge_Map_T3_'+img_name)

    return final_threshold_img_t1, final_threshold_img_t2, final_threshold_img_t3



# ensure proper arguments given
if (len(sys.argv)) < 2:
    print("Command failure. Please pass image name as parameter and try
again.\nExample: $ python3 canny.py House.bmp")
    exit()

# show input image from parameter passed
img_name = sys.argv[1]
im = Image.open(img_name).convert('L')
im.show()

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

# compute input dimentions
height, width = input_img.shape

# canny edge detection steps
```

```
smooth_image = gaussian_smoothening()
gradient_angle, gradient_magnitude = gradient_operation()
nms, magnitude_arr = non_maxima_suppression()
final_threshold_img_t1, final_threshold_img_t2, final_threshold_img_t3 =
thresholding()
```

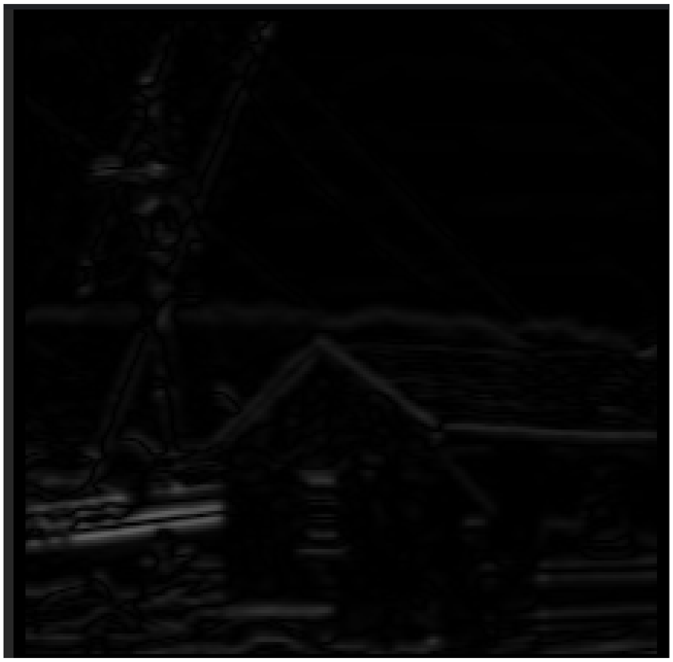
Output Images - House:



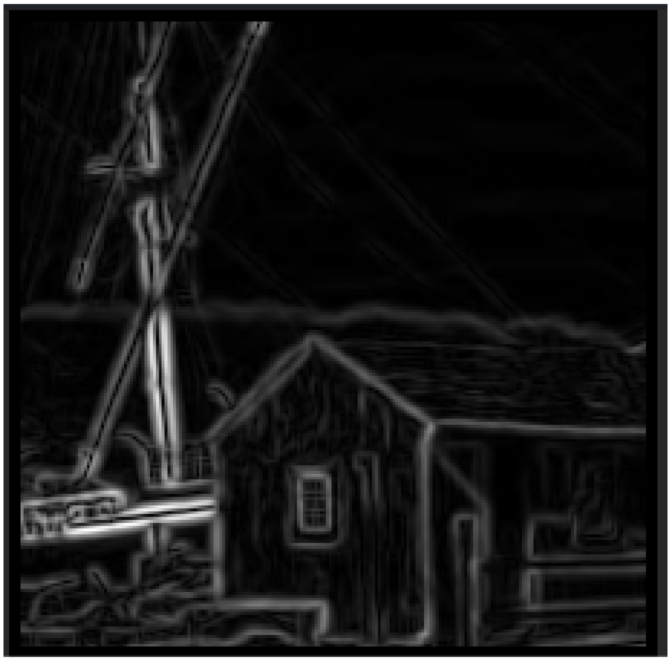
1_Smoothened_House.bmp



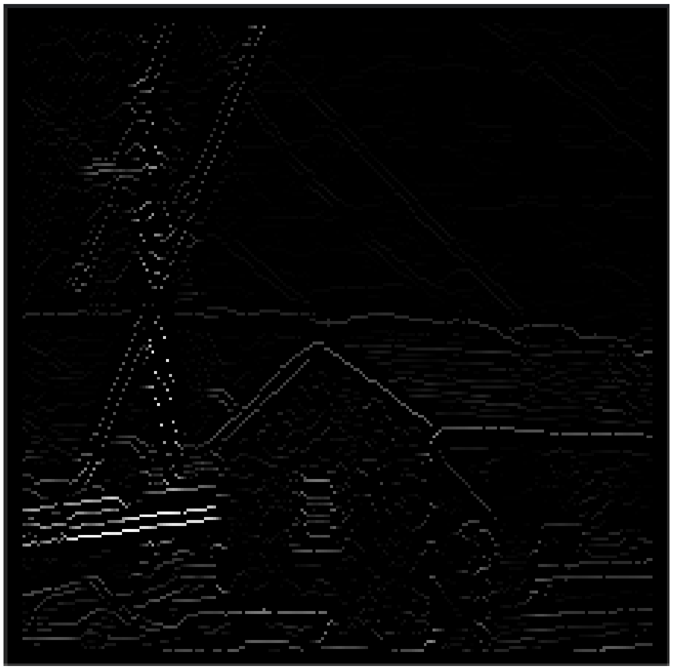
2_Horizontal_Gradient_House.bmp



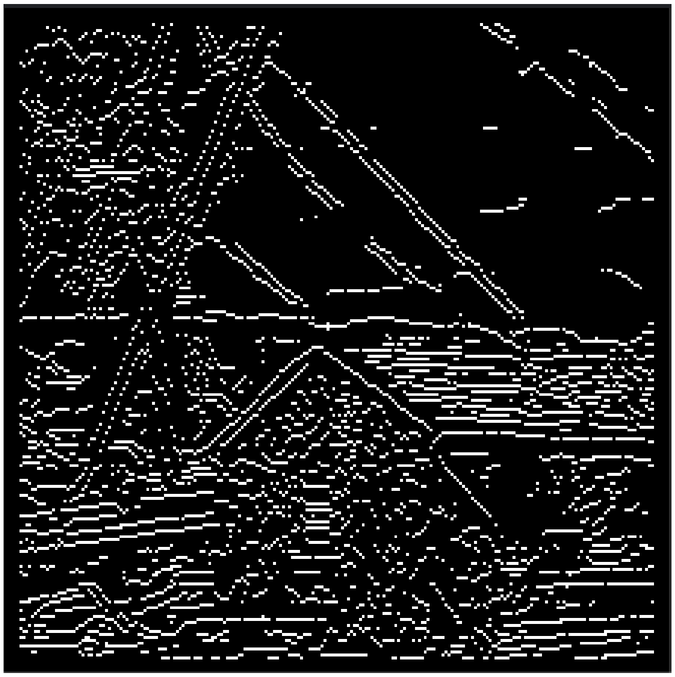
3_Vertical_Gradient_House.bmp



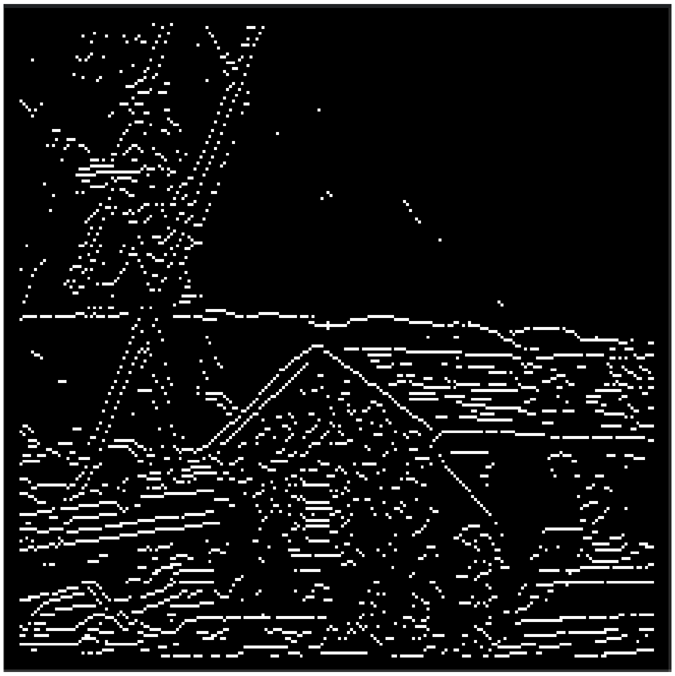
4_Gradient_Magnitude_House.bmp



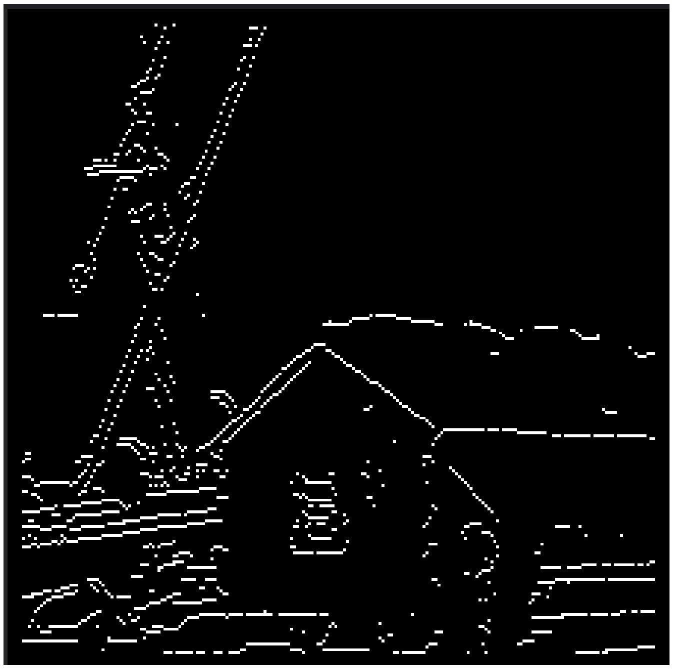
5_NMS_House.bmp



6_Binary_Edge_Map_T1_House.bmp

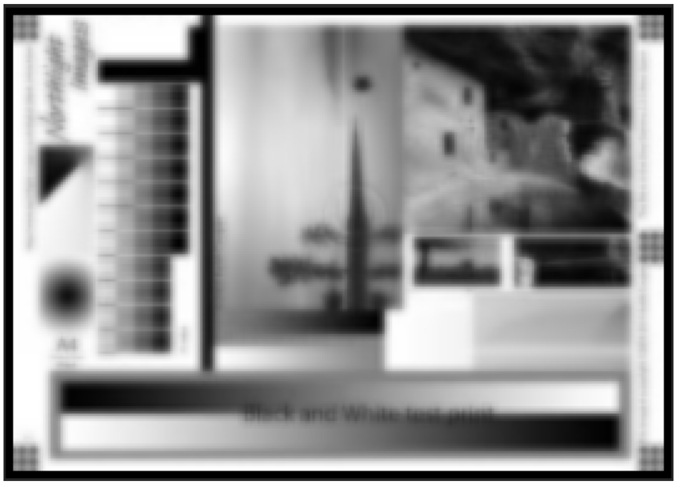


7_Binary_Edge_Map_T2_House.bmp

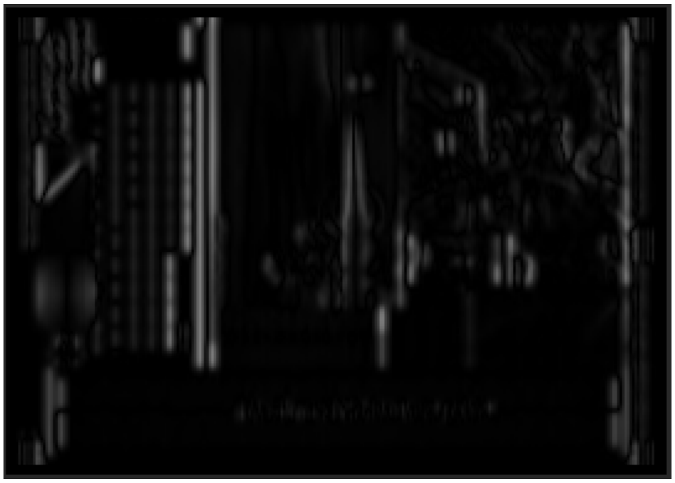


8_Binary_Edge_Map_T3_House.bmp

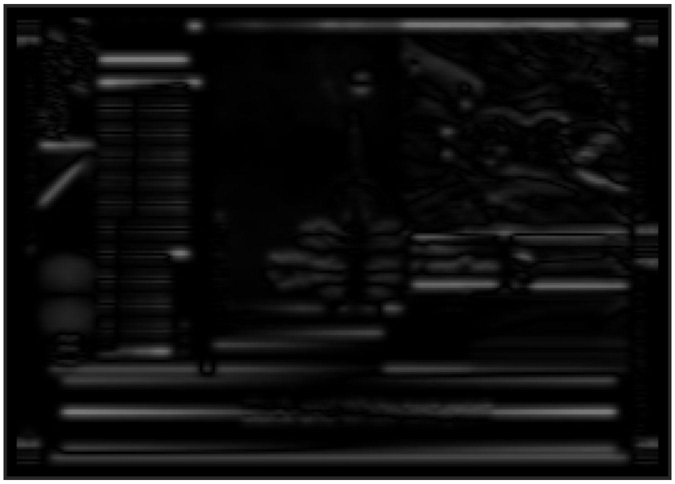
Output Images - Test Patterns:



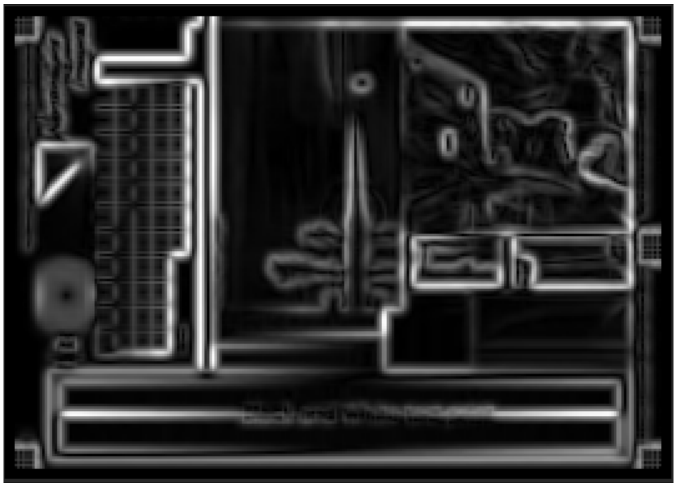
1_Smoothened_Test Patterns.bmp



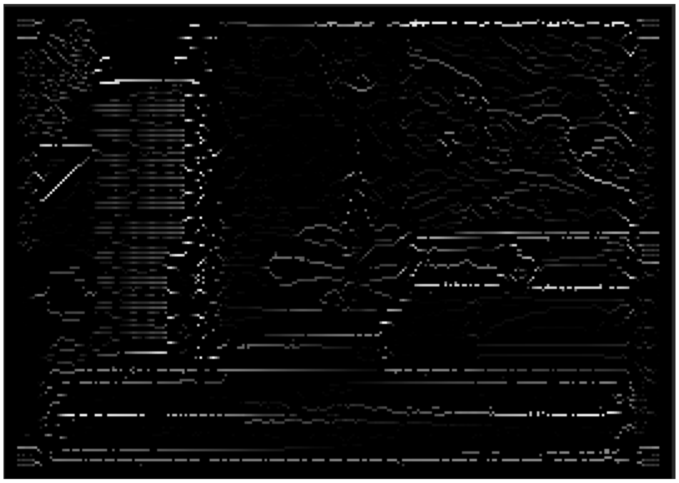
2_Horizontal_Gradient_Test Patterns.bmp



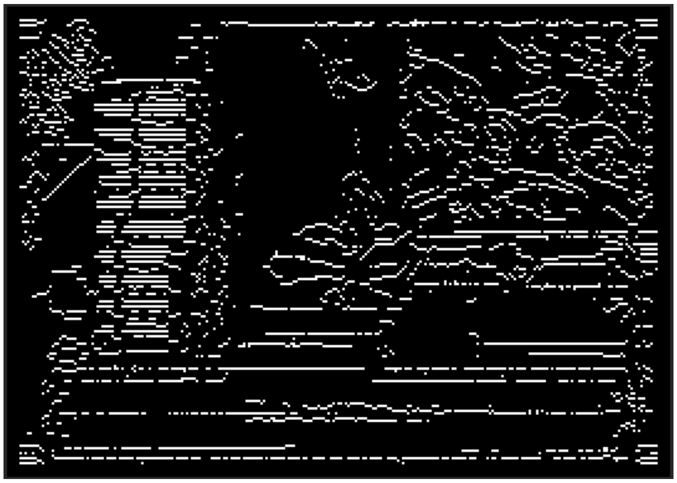
3_Vertical_Gradient_Test Patterns.bmp



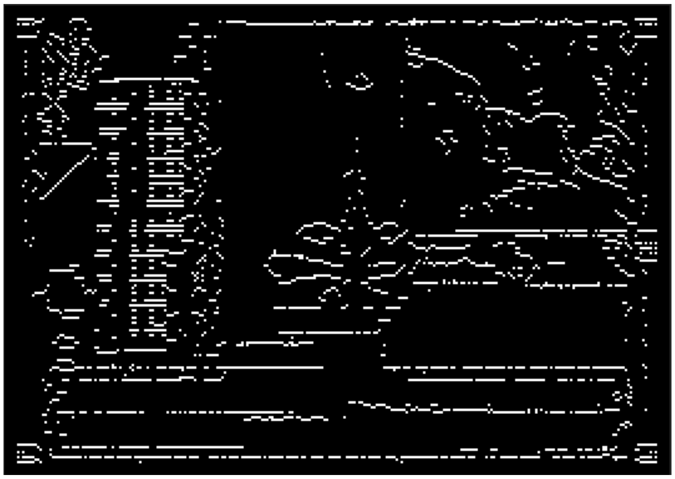
4_Gradient_Magnitude_Test Patterns.bmp



5_NMS_Test Patterns.bmp

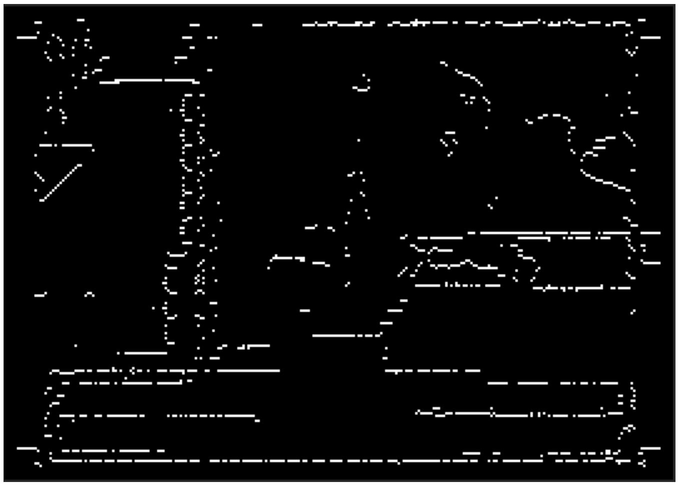


6_Binary_Edge_Map_T1_Test Patterns.bmp



7_Binary_Edge_Map_T2_Test Patterns.bmp

vg2237
sys8910



8_Binary_Edge_Map_T3_Test Patterns.bmp