Computer Vision

Assignment #4 – Canny Edge Detection and Harris Corner Detection

Problem #1: Program the Canny Edge Detection in Python by programming all steps of the algorithm (without using the OpenCV library Canny function) and experiment on different images by adjusting the parameters of the algorithm.

Canny Edge Detection is one of the best edge detection algorithms that produces a black and white image with edges as the white pixels. Canny edge detection follows five steps as:

1. Noise Removal by application of a Gaussian kernel filter to remove high frequencies.
2. Determining magnitude and angle of gradients (first order derivatives).
3. Non-maximum suppression in the direction of the gradient.
4. Double thresholding to transform the image in non-edges, weak edges and strong edges.
5. Hysteresis to determine the final edges by converting weak edges into strong or non-edges.

Step 1 can be programmed by implementing the Gaussian kernel equation in 2-d as:

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

```
def compute_gaussian_kernel(kernel_size, sigma):
    kernel = np.zeros((kernel_size,kernel_size),dtype=float)
    for x in range(-kernel_size//2+1,kernel_size//2+1):
        for y in range(-kernel_size//2+1,kernel_size//2+1):
            kernel[x+kernel_size//2,y+kernel_size//2] =
(1/(2*math.pi*sigma**2))*math.exp(-((x**2+y**2)/(2*sigma**2)))
    kernel = kernel
    return kernel, np.sum(kernel)
```

After the kernel is determined, we can convolve it with the input image (after it has been converted to gray scale) by using the following code:

```
def convolve(img: np.array, kernel: np.array) -> np.array:
    # kernel is assumed to be square
    output_size = (img.shape[0]-kernel.shape[0]+1, img.shape[1]-
kernel.shape[0]+1)

    output_img = np.zeros((output_size[0], output_size[1]),dtype=img.dtype)
    kernel_size = kernel.shape[0]  # kernel size

    for i in range(output_size[0]):
            for j in range(output_size[1]):
                mat = img[i:i+kernel_size, j:j+kernel_size] # values at current
kernel location
                mat = mat.reshape((kernel_size,kernel_size))
                # do element-wise multiplication and add the result
                output_img[i, j] = np.clip(np.sum(np.multiply(mat,
kernel)),0,255)
    return output_img
```

Step 2 of the Canny Edge detection uses the Sobel filter kernels to compute the gradients in the X and Y directions. The 3x3 Sobel kernels are:

$$\frac{dP}{dx} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\frac{dP}{dy} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Note that if we wanted to have a 5x5 kernel, then the two derivative kernels become:

$$\frac{d}{dx} = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{bmatrix}$$

$$\frac{d}{dy} = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 \\ -2 & -2 & -2 & -2 & -2 \end{bmatrix}$$

After applying the individual kernels in the X and Y directions, we can combine the resulting images to produce the gradient magnitude as:

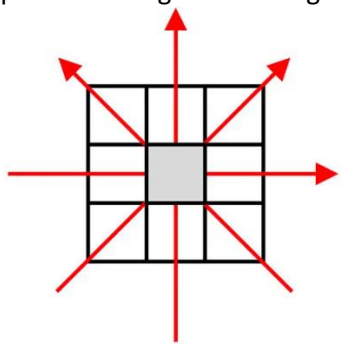
$$\text{Gradient Magnitude} = \sqrt{\left(\frac{dP}{dx}\right)^2 + \left(\frac{dP}{dy}\right)^2}$$

$$\text{Gradient Angles} = tan^{-1}\left(\frac{\frac{dP}{dy}}{\frac{dP}{dx}}\right)$$

The following code implements the above gradient and angle calculations on an image as:

```
def apply_sobel_filters(img):
    sobel_kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=float)
    sobel_ky = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype=float)

    img_x = convolve(img, sobel_kx)
    img_y = convolve(img, sobel_ky)

    G = np.hypot(img_x,img_y) # G = gradient magnitude
    G = G / G.max() * 255 # scale 0-255
    theta = np.arctan2(img_y, img_x)
    return (G, theta)
```

Step 3 of the Canny Edge detection (Non maximal suppression) uses the angle information rounded to the nearest 45 degrees to determine the previous and the next pixel with respect to the current pixel. This is because the 8 neighbors of the pixel form angles in 45 degree increments as shown below.



In non maximal suppression, we compare three magnitude values in the direction of the gradient at the current pixel location. If the magnitude of the gradient is either less than the previous pixel's gradient magnitude (in the direction of the gradient), or the next pixel's magnitude, we suppress the output (make it zero) in the resulting edge detected image. The following code implements the non maximal suppression.

```
def non_maximal_suppression(img, gradient_angle): # gradient_angle matrix is in
radians
    img_out = np.zeros((img.shape[0],img.shape[1])) # image after non max
suppression
    angle_degrees = gradient_angle * 180. / np.pi
    angle_degrees[angle_degrees < 0] += 180

    for i in range(1,img.shape[0]-1):
        for j in range(1,img.shape[1]-1):
            prev_pixel = 0  # previous pixel in the direction of gradient
            next_pixel = 0

            # close to horizontal
            if (0 <= angle_degrees[i,j] < 22.5) or (157.5 <= angle_degrees[i,j]
<= 180):
                prev_pixel = img[i, j-1]  # i is row, j is column
                next_pixel = img[i, j+1]
            # close to 45 degrees
            elif (22.5 <= angle_degrees[i,j] < 67.5):
                prev_pixel = img[i+1, j-1]
                next_pixel = img[i-1, j+1]
            #close to 90 degrees
            elif (67.5 <= angle_degrees[i,j] < 112.5):
                prev_pixel = img[i+1, j]
                next_pixel = img[i-1, j]
            #close to 135 degrees
            elif (112.5 <= angle_degrees[i,j] < 157.5):
                prev_pixel = img[i-1, j-1]
                next_pixel = img[i+1, j+1]

            # compare previous and next pixels in the gradient direction and
            # zero out current pixel if it is smaller than the prev. or next.
            if (img[i,j] >= prev_pixel) and (img[i,j] >= next_pixel):
                img_out[i,j] = img[i,j]
```

```
            else:
                img_out[i,j] = 0
    return img_out
```

Step 4 of the Canny Edge Detection implements the double thresholding to determine if each pixel is a non-edge, a weak edge or a strong edge. For example if the low threshold value is 40 and high threshold value is 80, then we check the gradient magnitude matrix, and all locations where the magnitude is less than 40 will be changed to 0 (non-edge) in the output image, and the pixel locations where the magnitude is greater than the high threshold i.e., 80 in our example, will be changed to 255 (definitely an edge) in the output image. The pixel locations where the magnitude of the gradient is in between the low and the high threshold value will be changed to fixed predetermined pixel value e.g., 50 indicating it is a weak edge. The last step of the algorithm will convert a weak edge to a non-edge (value of 0) or a strong edge (value of 255). The following code implements the double thresholding part of the Canny Edge Detection algorithm.

```
def double_threshold_image(img, lowThreshold=20, highThreshold=80,
weak_value=50):
    M, N = img.shape
    thresholded_img = np.zeros((M,N), dtype=np.int32)
    strong_value = 255
    # determine indices of zero, weak and strong points
    zeros_i, zeros_j = np.where(img < lowThreshold)
    weak_i, weak_j = np.where((img <= highThreshold) & (img >= lowThreshold))
    strong_i, strong_j = np.where(img >= highThreshold)

    thresholded_img[strong_i, strong_j] = strong_value
    thresholded_img[weak_i, weak_j] = weak_value
    thresholded_img[zeros_i,zeros_j] = 0
    return thresholded_img
```

The last step of the Canny Edge Detection algorithm is known as the hysteresis and its job is to decide if a weak edge should be converted to a non-edge or a strong edge. The decision is made by examining the 8 neighbors of the pixel where the weak edge exists. If any of the 8 neighbors is a strong edge, then weak edge value is changed to 25, i.e., a strong edge in the output image. The following code implements the hysteresis step of the Canny Edge Detection algorithm.

```
def hysteresis(img, weak, strong=255):
    for i in range(1, img.shape[0]-1):
        for j in range(1, img.shape[1]-1):
            if (img[i,j] == weak): # if current pixel is weak, check if one of
its 8 neighbors is strong
                if ((img[i-1, j-1] == strong) or (img[i, j-1] == strong) or
(img[i+1, j-1] == strong)
                    or (img[i-1, j] == strong) or (img[i+1, j] == strong)
                    or (img[i-1, j+1] == strong) or (img[i, j+1] == strong) or
(img[i+1, j+1] == strong)):
                    img[i, j] = strong
                else:
                    img[i, j] = 0
    return img
```

Now that we understand the difference steps of the Canny Edge Detection algorithm, lets put these together in a Python program and test it on different images. Create a new Python application called CannyEdgeDetection_yourID. Then add a file called Utils.py to the project with the following code in it.

```
import numpy as np
import math

def compute_gaussian_kernel(kernel_size, sigma):
    kernel = np.zeros((kernel_size,kernel_size),dtype=float)
    for x in range(-kernel_size//2+1,kernel_size//2+1):
        for y in range(-kernel_size//2+1,kernel_size//2+1):
            kernel[x+kernel_size//2,y+kernel_size//2] =
(1/(2*math.pi*sigma**2))*math.exp(-((x**2+y**2)/(2*sigma**2)))
    kernel = kernel
    return kernel, np.sum(kernel)

def apply_sobel_filters(img):
    sobel_kx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=float)
    sobel_ky = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]], dtype=float)

    img_x = convolve(img, sobel_kx)
    img_y = convolve(img, sobel_ky)

    #G = np.sqrt((img_x**2) + (img_y**2)) # gives different result than hypot
    G = np.hypot(img_x,img_y)
    G = G / G.max() * 255 # scale 0-255
    theta = np.arctan2(img_y, img_x)
    return (G, theta)

def convolve(img: np.array, kernel: np.array) -> np.array:
    # kernel is assumed to be square
    output_size = (img.shape[0]-kernel.shape[0]+1, img.shape[1]-
kernel.shape[0]+1)

    output_img = np.zeros((output_size[0], output_size[1]),dtype=img.dtype)
    kernel_size = kernel.shape[0]  # kernel size

    for i in range(output_size[0]):
            for j in range(output_size[1]):
                mat = img[i:i+kernel_size, j:j+kernel_size] # values at current
kernel location
                mat = mat.reshape((kernel_size,kernel_size))
                # do element-wise multiplication and add the result
                output_img[i, j] = np.clip(np.sum(np.multiply(mat,
kernel)),0,255)
    return output_img

def non_maximal_suppression(img, gradient_angle): # gradient_angle is in
radians
    img_out = np.zeros((img.shape[0],img.shape[1])) # image after non max
suppression
    angle_degrees = gradient_angle * 180. / np.pi
    angle_degrees[angle_degrees < 0] += 180

    for i in range(1,img.shape[0]-1):
```

```
        for j in range(1,img.shape[1]-1):
            prev_pixel = 0  # previous pixel in the direction of gradient
            next_pixel = 0

            # close to horizontal
            if (0 <= angle_degrees[i,j] < 22.5) or (157.5 <= angle_degrees[i,j]
<= 180):
                prev_pixel = img[i, j-1]  # i is row, j is column
                next_pixel = img[i, j+1]
            # close to 45 degrees
            elif (22.5 <= angle_degrees[i,j] < 67.5):
                prev_pixel = img[i+1, j-1]
                next_pixel = img[i-1, j+1]
            #close to 90 degrees
            elif (67.5 <= angle_degrees[i,j] < 112.5):
                prev_pixel = img[i+1, j]
                next_pixel = img[i-1, j]
            #close to 135 degrees
            elif (112.5 <= angle_degrees[i,j] < 157.5):
                prev_pixel = img[i-1, j-1]
                next_pixel = img[i+1, j+1]

            # compare previous and next pixels in the gradient direction and
            # zero out current pixel if it is smaller than the prev. or next.
            if (img[i,j] >= prev_pixel) and (img[i,j] >= next_pixel):
                img_out[i,j] = img[i,j]
            else:
                img_out[i,j] = 0
    return img_out

def double_threshold_image(img, lowThreshold=20, highThreshold=80,
weak_value=50):
    M, N = img.shape
    thresholded_img = np.zeros((M,N), dtype=np.int32)
    strong_value = 255
    # determine indices of zero, weak and strong points
    zeros_i, zeros_j = np.where(img < lowThreshold)
    weak_i, weak_j = np.where((img <= highThreshold) & (img >= lowThreshold))
    strong_i, strong_j = np.where(img >= highThreshold)

    thresholded_img[strong_i, strong_j] = strong_value
    thresholded_img[weak_i, weak_j] = weak_value
    thresholded_img[zeros_i,zeros_j] = 0
    return thresholded_img

def hysteresis(img, weak, strong=255):
    for i in range(1, img.shape[0]-1):
        for j in range(1, img.shape[1]-1):
            if (img[i,j] == weak): # if current pixel is weak, check if one of
8 neighbors is strong
                if ((img[i-1, j-1] == strong) or (img[i, j-1] == strong) or
(img[i+1, j-1] == strong)
                    or (img[i-1, j] == strong) or (img[i+1, j] == strong)
                    or (img[i-1, j+1] == strong) or (img[i, j+1] == strong) or
(img[i+1, j+1] == strong)):
```

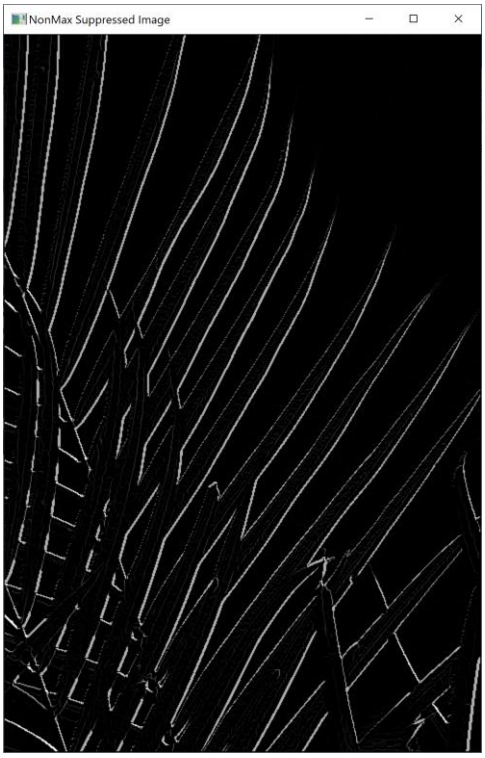
```
                    img[i, j] = strong
                else:
                    img[i, j] = 0
    return img
```

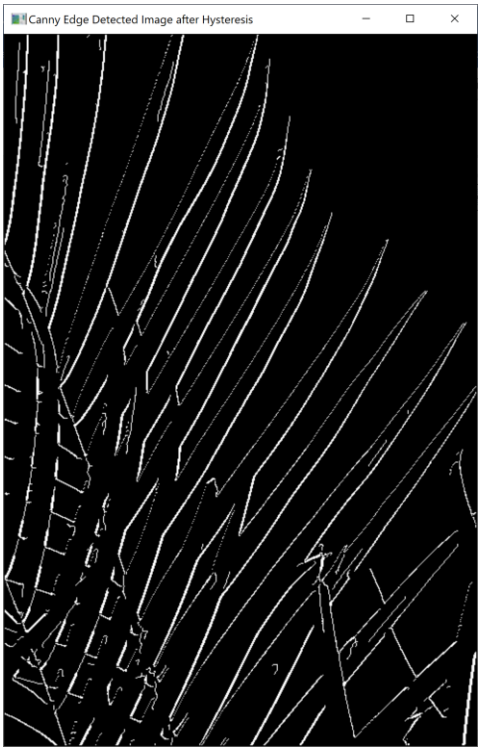
In the CannyEdgeDetection.py, type the following code:

```
import sys
import Utils
import cv2
import numpy as np

def main():
    image_filename = 'd:/temp/canny1.jpg'
    img_color = cv2.imread(image_filename) # original color image
    img = cv2.imread(image_filename,0)  # 0 indicates to read as grayscale

    print(img.shape)
    if img is None:
        print('Could not open or find the image: ', image_filename)
        exit(0)

    # step 1: apply gaussian filter to remove noise
    g,denom = Utils.compute_gaussian_kernel(3,1.0) # kernel size = 3, sigma = 1
    print(g, denom)
    kernel = g/denom

    noise_removed_img = Utils.convolve(img,kernel)
    cv2.imshow('Original Image Color', img_color)
    cv2.imshow('Original Image - Gray scale', img)
    cv2.imshow('Gausssian Filtered Image', noise_removed_img)
    cv2.waitKey()

    # step 2: apply sobel filters, get gradient magnitude and angles
    gradient_magnitude_img, gradient_angles =
Utils.apply_sobel_filters(noise_removed_img)
    # open cv require image data to be in uint8 form before displaying it
    gradient_magnitude_img = gradient_magnitude_img.astype(np.uint8)
    cv2.imshow('Gradient Magnitude Image', gradient_magnitude_img)
    cv2.waitKey()

    # step 3: apply non maximal suppression
    nonmax_suppressed_img =
Utils.non_maximal_suppression(gradient_magnitude_img,gradient_angles)
    nonmax_suppressed_img = nonmax_suppressed_img.astype(np.uint8)
    cv2.imshow('NonMax Suppressed Image', nonmax_suppressed_img)
    cv2.waitKey()

    # step 4: apply thresholding and then hysteresis
    # first determine weak and strong pixels
    low_threshold = 10
    high_threshold = 60
    weak_value = 50  # pixel value for weak edges
```

```
    img_thresholded =
Utils.double_threshold_image(nonmax_suppressed_img,low_threshold,
high_threshold, weak_value)
    hysteresis_img = Utils.hysteresis(img_thresholded, weak_value)
    hysteresis_img = hysteresis_img.astype(np.uint8)
    cv2.imshow('Canny Edge Detected Image after Hysteresis', hysteresis_img)
    cv2.waitKey()

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

The output images from different steps appear as:





Problem #2:

Experiment the effect of using 5x5 Gaussian kernels, 5x5 Sobel kernels and different double threshold values. Compare your results by using the Canny function from the OpenCV library.

Problem #3: Program the Harris Corner Detection Algorithm in Python by programming all steps of the algorithm (without using the OpenCV library cornerHarris function) and experiment on different images by adjusting the parameters of the algorithm.

Harris Corner detection is one of the best corner detection algorithms that yields x,y coordinates of the corner points It operates on the gray scale image. The algorithm is based on finding the difference in intensity for a displacement of a small window (e.g., 5x5) in all directions. This is described mathematically as:

$$E(u,v) = \sum_{x,y} \underbrace{w(x,y)}_{\text{window function}} \underbrace{[I(x+u,y+v)}_{\text{shifted intensity}} - \underbrace{I(x,y)}_{\text{intensity}}]^2$$

Using first order Taylor series expansion, the shift of the image can be expressed as:

$$\sum [I(x+u,y+v) - I(x,y)]^2$$

$$\approx \sum [I(x,y) + uI_x + vI_y - I(x,y)]^2 \quad \text{First order approx}$$

$$= \sum u^2 I_x^2 + 2uvI_xI_y + v^2 I_y^2$$

$$= \sum \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} \quad \text{Rewrite as matrix equation}$$

$$= \begin{bmatrix} u & v \end{bmatrix} \left(\sum \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \right) \begin{bmatrix} u \\ v \end{bmatrix}$$

Which can be expressed as:

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

Where:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_xI_x & I_xI_y \\ I_xI_y & I_yI_y \end{bmatrix}$$

M is basically a 2x2 matrix obtained by summing the products of first derivatives in the x and y directions in the small window at the location of each pixel (e.g., 5x5 window). By looking at the distribution of the first derivatives in x and y directions, it is intuitively clear that both derivatives in x and y directions i.e., Ix and Iy will be high at the corner locations. Thus, the Harris corner algorithm uses this intuition to come up with a decision for a corner in terms of *M* as:

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

This is because at a corner point, both Ix and Iy will be high which means that the product of Eigen values (given by the determinant of the matrix) will be high at the corner point. If we subtract the sum of Eigen values (given by trace of the matrix), then *R* produces a high value at the corner points and

reduces the possibility where one of the Eigen values may be high (e.g., at an edge) but not the other. K is a weighting factor e.g., 0.04 that is empirically determined.

Create a new Python application called HarrisCornerDetection_yourID. Then add a file called Utils.py to the project with the following code in it.

```
import numpy as np
import math

def compute_gaussian_kernel(kernel_size, sigma):
    kernel = np.zeros((kernel_size,kernel_size),dtype=float)
    for x in range(-kernel_size//2+1,kernel_size//2+1):
        for y in range(-kernel_size//2+1,kernel_size//2+1):
            kernel[x+kernel_size//2,y+kernel_size//2] =
(1/(2*math.pi*sigma**2))*math.exp(-((x**2+y**2)/(2*sigma**2)))
    kernel = kernel
    return kernel, np.sum(kernel)

def convolve(img: np.array, kernel: np.array) -> np.array:
    # kernel is assumed to be square
    output_size = (img.shape[0]-kernel.shape[0]+1, img.shape[1]-
kernel.shape[0]+1)

    output_img = np.zeros((output_size[0], output_size[1]),dtype=img.dtype)
    kernel_size = kernel.shape[0]  # kernel size

    for i in range(output_size[0]):
            for j in range(output_size[1]):
                mat = img[i:i+kernel_size, j:j+kernel_size] # values at current
kernel location
                mat = mat.reshape((kernel_size,kernel_size))
                # do element-wise multiplication and add the result
                output_img[i, j] = np.clip(np.sum(np.multiply(mat,
kernel)),0,255)
    return output_img
```

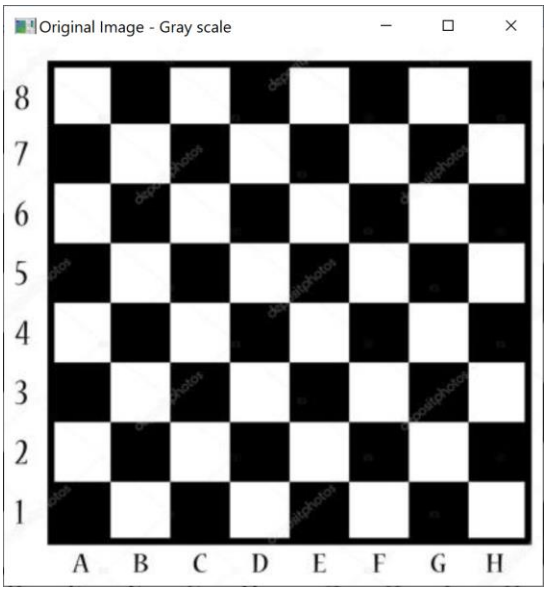
The code in the main file (HarrisCornerDetection.py) appears as:

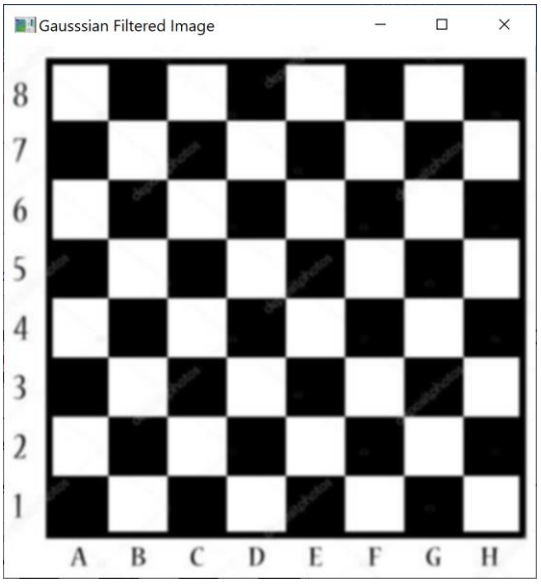
```
import sys
import cv2
import Utils
import numpy as np
import matplotlib.pyplot as plt

def main():
    image_filename = 'd:/temp/chess2.jpg'
    img_color = cv2.imread(image_filename) # original color image
    img = cv2.imread(image_filename,0)  # 0 indicates to read as grayscale

    print(img.shape)
    if img is None:
        print('Could not open or find the image: ', image_filename)
        exit(0)
```

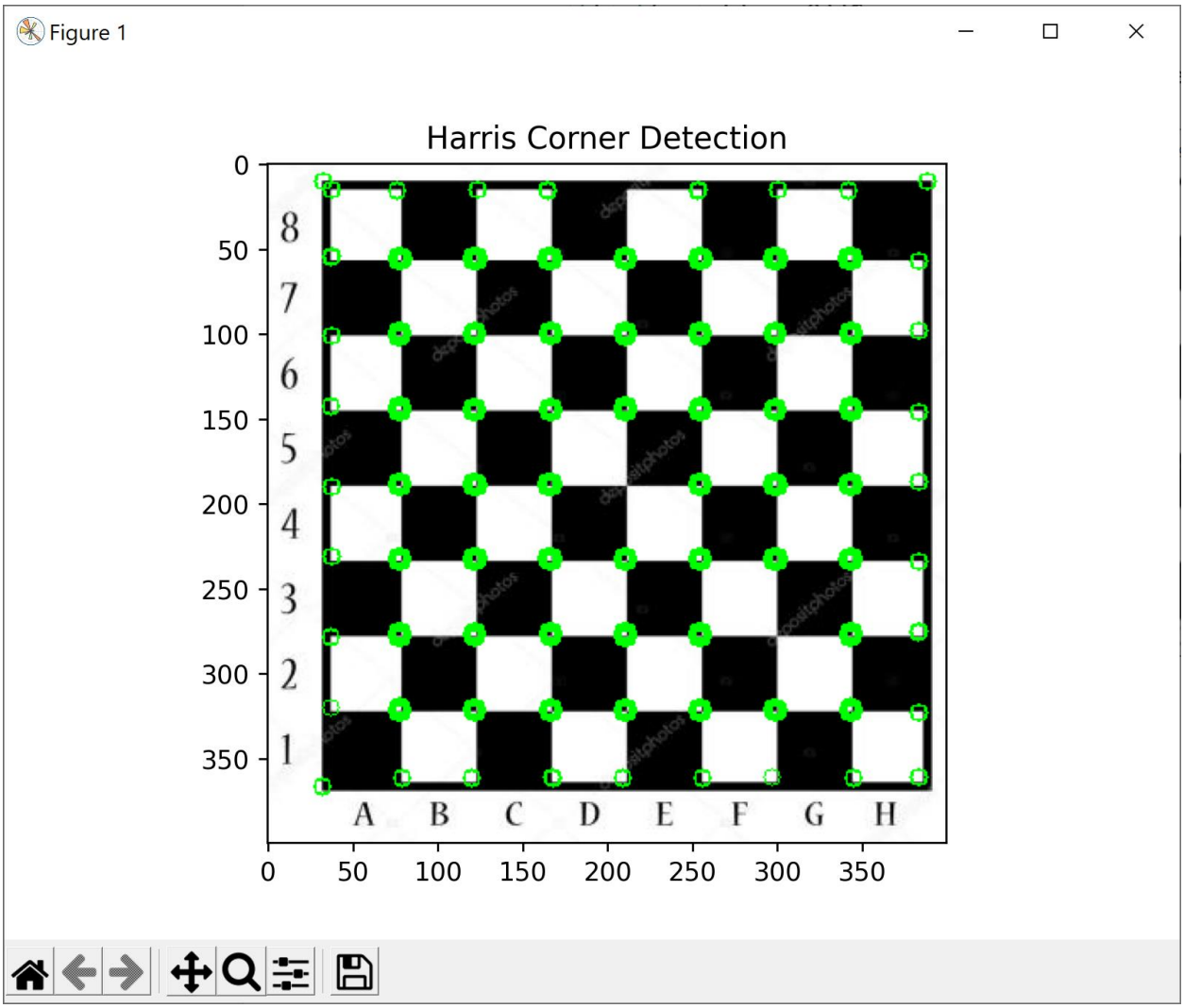
```
    g,denom = Utils.compute_gaussian_kernel(3,1.0) # kernel size = 5, sigma = 1
    print(g, denom)
    kernel = g/denom # gaussian kernel

    # Harris corner parameters
    k = 0.04
    threshold = 0.7
    window_size = 5

    # step 1: apply gaussian filter to remove noise
    noise_removed_img = Utils.convolve(img,kernel)
    cv2.imshow('Original Image Color', img_color)
    cv2.imshow('Original Image - Gray scale', img)
    cv2.imshow('Gausssian Filtered Image', noise_removed_img)
    cv2.waitKey()

    img = noise_removed_img
    width = img.shape[0]
    height = img.shape[1]

    #  step 2: obtain first derivatives in x and y directions
    dy, dx = np.gradient(img) # numpy implements a simple differencing
derivative
    Ixx = dx**2
    Ixy = dx*dy
    Iyy = dy**2

    corner_list = []
    offset = int(window_size/2)
    matrix_resp = np.zeros((width,height))
    # step 3: compute Harris corner response at each pixel location
    # by sliding the window
    for y in range(offset, (img.shape[0] - offset)):
        for x in range(offset, (img.shape[1] - offset)):
            # sliding window
            start_y = y - offset
            end_y = y + offset + 1
            start_x = x - offset
            end_x = x + offset + 1
            windowIxx = Ixx[start_y : end_y, start_x : end_x]
            windowIxy = Ixy[start_y : end_y, start_x : end_x]
            windowIyy = Iyy[start_y : end_y, start_x : end_x]

            # sum derivatives in the window
            Sxx = windowIxx.sum()
            Sxy = windowIxy.sum()
            Syy = windowIyy.sum()

            # determinant and trace of the matrix
            det = (Sxx * Syy) - (Sxy**2)
            trace = Sxx + Syy

            resp = det - k*(trace**2)  # Harris corner response
            matrix_resp[x,y] = resp
```

```
    # step 4: threshold the Harris response to determine corners
    corner_list = []
    cv2.normalize(matrix_resp, matrix_resp, 0, 1, cv2.NORM_MINMAX)
    for y in range(offset, height-offset):
        for x in range(offset, width-offset):
            value=matrix_resp[x, y]
            if value>threshold:
                corner_list.append([x, y, value])
                cv2.circle(img,(x,y),3,(255,0,0))

    for v in corner_list:
        cv2.circle(img_color,(v[0],v[1]),5,(0,255,0))

    plt.imshow(cv2.cvtColor(img_color, cv2.COLOR_BGR2RGB)), plt.title("Harris
Corner Detection")
    plt.show()

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

If we use a chess board image, the results of applying the Harris corner detection appear as:







Problem #4: Experiment with different parameters of the Harris Corner Detection Algorithm and compare you results with the cornerHarris function from the OpenCV library.