ELE510 Image Processing with robot vision: LAB, Exercise 6, Image features detection.

Purpose: *To learn about the edges and corners features detection, and their descriptors.*

The theory for this exercise can be found in chapter 7 of the text book [1] and in appendix C in the compendium [2]. See also the following documentations for help:

- OpenCV
- numpy
- matplotlib
- scipy

IMPORTANT: Read the text carefully before starting the work. In many cases it is necessary to do some preparations before you start the work on the computer. Read necessary theory and answer the theoretical part first. The theoretical and experimental part should be solved individually. The notebook must be approved by the lecturer or his assistant.

Approval:

The current notebook should be submitted on CANVAS as a single pdf file.

To export the notebook in a pdf format, goes to File -> Download as -> PDF via LaTeX (.pdf).

Note regarding the notebook: The theoretical questions can be answered directly on the notebook using a *Markdown* cell and LaTex commands (if relevant). In alternative, you can attach a scan (or an image) of the answer directly in the cell.

Possible ways to insert an image in the markdown cell:

`![image name]("image_path")`

`<img src="image_path" alt="Alt text" title="Title text" />`

Under you will find parts of the solution that is already programmed.

You have to fill out code everywhere it is indicated with `...`

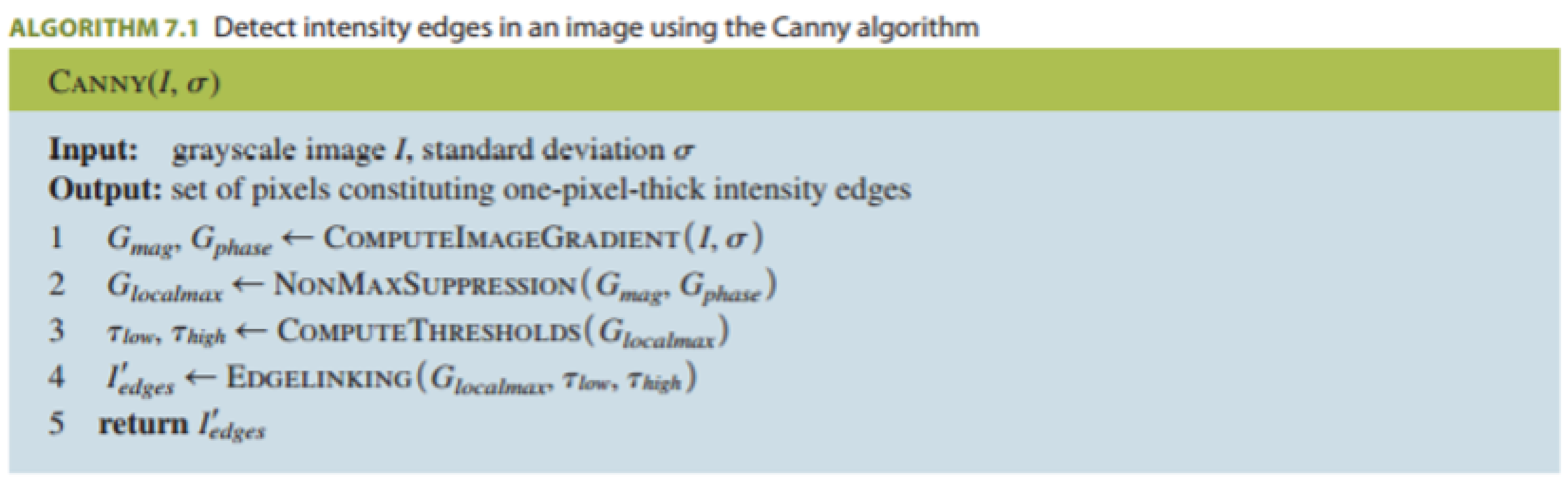
The code section under `######## a)` is answering subproblem a) etc.

Problem 1

Intensity edges are pixels in the image where the intensity (or graylevel) function changes rapidly.

The **Canny edge detector** is a classic algorithm for detecting intensity edges in a grayscale image that relies on the gradient magnitude. The algorithm was developed by John F. Canny in 1986. It is a multi-stage algorithm that provides good and reliable detection.

a) Create the **Canny algorithm**, described at pag. 336 (alg. 7.1). For the last step (`EDGELINKING`) you can either use the algorithm 7.3 at page 338 or the `HYSTERESIS THRESHOLD` algorithm 10.3 described at page 451. All the following images are taken from the text book [1].



ALGORITHM 7.1 Detect intensity edges in an image using the Canny algorithm

```
CANNY($I, \sigma$)

Input:  grayscale image $I$, standard deviation $\sigma$
Output: set of pixels constituting one-pixel-thick intensity edges
1  $G_{mag}, G_{phase} \leftarrow$ COMPUTEIMAGEGRADIENT($I, \sigma$)
2  $G_{localmax} \leftarrow$ NONMAXSUPPRESSION($G_{mag}, G_{phase}$)
3  $\tau_{low}, \tau_{high} \leftarrow$ COMPUTETHRESHOLDS($G_{localmax}$)
4  $I'_{edges} \leftarrow$ EDGELINKING($G_{localmax}, \tau_{low}, \tau_{high}$)
5  return $I'_{edges}$
```

ALGORITHM 7.2 Perform non-maximal suppression

```
NonMaxSuppression($G_{mag}$, $G_{phase}$)

Input:  gradient magnitude and phase
Output: gradient magnitude with all nonlocal maxima set to zero
1  for $(x, y) \in G_{mag}$ do                                              ▷ For each pixel,
2      $\theta \leftarrow G_{phase}(x, y)$                                     adjust the phase
3      if $\theta \geq \frac{7\pi}{8}$ then $\theta \leftarrow \theta - \pi$                                   to ensure that
4      if $\theta < -\frac{\pi}{8}$ then $\theta \leftarrow \theta + \pi$                                   $-\frac{\pi}{8} \leq \theta < \frac{7\pi}{8}$.
5      if $-\frac{\pi}{8} \leq \theta < \frac{\pi}{8}$ then $neigh_1 \leftarrow G_{mag}(x-1, y)$, $neigh_2 \leftarrow G_{mag}(x+1, y)$
6      elseif $\frac{\pi}{8} \leq \theta < \frac{3\pi}{8}$ then $neigh_1 \leftarrow G_{mag}(x-1, y-1)$, $neigh_2 \leftarrow G_{mag}(x+1, y+1)$
7      elseif $\frac{3\pi}{8} \leq \theta < \frac{5\pi}{8}$ then $neigh_1 \leftarrow G_{mag}(x, y-1)$, $neigh_2 \leftarrow G_{mag}(x, y+1)$
8      elseif $\frac{5\pi}{8} \leq \theta < \frac{7\pi}{8}$ then $neigh_1 \leftarrow G_{mag}(x-1, y+1)$, $neigh_2 \leftarrow G_{mag}(x+1, y-1)$
9      if $v \geq neigh_1$ AND $v \geq neigh_2$ then                       ▷ If the pixel is a local maximum
10         $G_{localmax}(x, y) \leftarrow G_{mag}(x, y)$                       in the direction of the gradient,
11     else                                                                  then retain the value;
12         $G_{localmax}(x, y) \leftarrow 0$                                       otherwise set it to zero.
13 return $G_{localmax}$
```

ALGORITHM 7.3 Perform edge linking

```
EdgeLinking($G_{localmax}$, $\tau_{low}$, $\tau_{high}$)

Input:  local gradient magnitude maxima $G_{localmax}$ along with low and high thresholds
Output: binary image $I'_{edges}$ indicating which pixels are along linked edges
1  for $(x, y) \in G_{localmax}$ do
2      if $G_{localmax}(x, y) > \tau_{high}$ then
3          frontier.Push(x, y)
4          $I'_{edges}(q) \leftarrow$ ON
5  while frontier.Size > 0 do
6      $p \leftarrow$ frontier.Pop()
7      for $q \in \mathcal{N}(p)$ do
8          if $G_{localmax}(q) > \tau_{low}$ then
9              frontier.Push(q)
10             $I'_{edges}(q) \leftarrow$ ON
11 return $I'_{edges}$
```

Remember:

- Sigma (second parameter in the Canny algorithm) is not necessary for the calculation since the Sobel operator (in opencv) combines the Gaussian smoothing and differentiation, so the results is nore or less resistant to the noise.
- We are defining the low and high thresholds manually in order to have a better comparison with the predefined opencv function. It is possible to extract the low and high thresholds automatically from the image but it is not required in this problem.

b) Test your algorithm with a image of your choice and compare your results with the predefined function in opencv:

```
cv2.Canny(img, t_low, t_high, L2gradient=True)
```
Documentation.

P.S. :

The goal of this problem it is not to create a **perfect** replication of the algorithm in opencv, but to understand the various steps involved and to be able to extract the edges from an ima ge using these steps.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import cv2
```

In [ ]:
```
# Sobel operator to find the first derivate in the horizontal and vertical directions
def computeImageGradient(Im, ksize=3):
    # Sobel operator  to find the first derivate in the horizontal and vertical directions

    ## TODO: The default ksize is 3, try different values and comment the result
    g_x = cv2.Sobel(Im, ddepth=cv2.CV_32F, dx=1, dy=0, ksize=ksize)
```

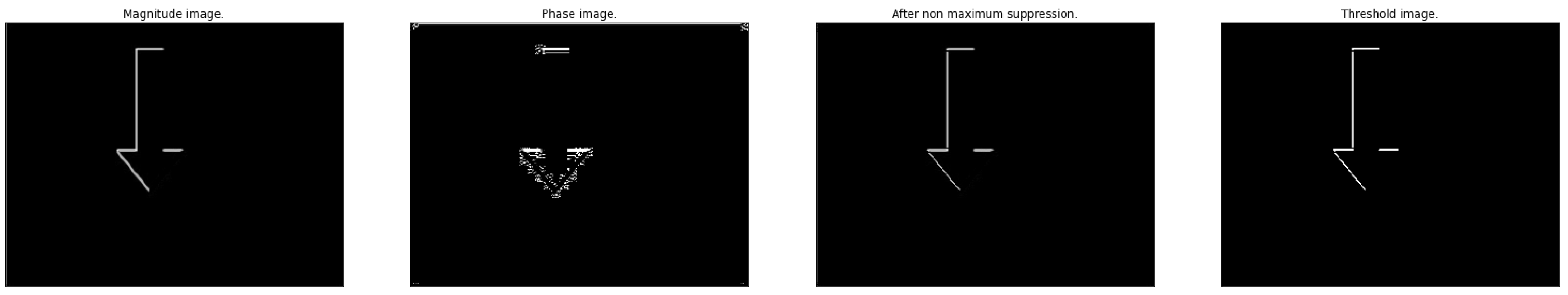
```python
    g_y = cv2.Sobel(Im, ddepth=cv2.CV_32F, dx=0, dy=1, ksize=ksize)

    ############################
    # Calculate the magnitude and the gradient direction like it is performed during the assignment 4 (problem 2a)
    kernelx, kernely = np.array([[1,0,-1],[1,0,-1],[1,0,-1]]), np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
    I_x, I_y = cv2.filter2D(Im, -1, kernelx), cv2.filter2D(Im, -1, kernely)

    G_mag = np.around(np.hypot(I_x, I_y), decimals=2).astype(int)
    G_phase = (((180/np.pi) * np.arctan2(I_y, I_x) ) % 180).reshape(G_mag.shape).astype(int)

    return G_mag, G_phase
```

In [ ]:
```python
# NonMaxSuppression algorithm
def nonMaxSuppression(G_mag, G_phase):
    G_localmax = np.zeros((G_mag.shape))

    # For each pixel, adjust the phase to ensure that -pi/8 <= theta < 7*pi/8
    pi_tresh = np.pi/8
    for x in range(G_localmax.shape[0]):
        x_sub, x_sup = max(0, x-1), x if x+1 >= G_localmax.shape[0] else x + 1
        for y in range(G_localmax.shape[1]):
            y_sub, y_sup = max(0, y-1), y if y+1 >= G_localmax.shape[1] else y + 1
            thetha, V = G_phase[x,y], G_mag[x,y]
            neigh1, neigh2 = -1, -1

            thetha = thetha - np.pi if (thetha >= pi_tresh * 7) else (thetha + np.pi if thetha < -pi_tresh else thetha)
            if thetha >= -pi_tresh and thetha < pi_tresh:
                neigh1, neigh2 = G_mag[x_sub, y], G_mag[x_sup, y]
            elif thetha >= pi_tresh and thetha < pi_tresh * 3:
                neigh1, neigh2 = G_mag[x_sub, y_sup], G_mag[x_sup, y_sub]
            elif thetha >= pi_tresh * 3 and thetha < pi_tresh * 5:
                neigh1, neigh2 = G_mag[x, y_sub], G_mag[x, y_sup]
            elif thetha >= pi_tresh * 5 and thetha < pi_tresh * 7:
                neigh1, neigh2 = G_mag[x_sub, y_sub], G_mag[x_sup, y_sup]

            G_localmax[x,y] = V if (V >= neigh1 and V >= neigh2) else 0
    return G_localmax
```

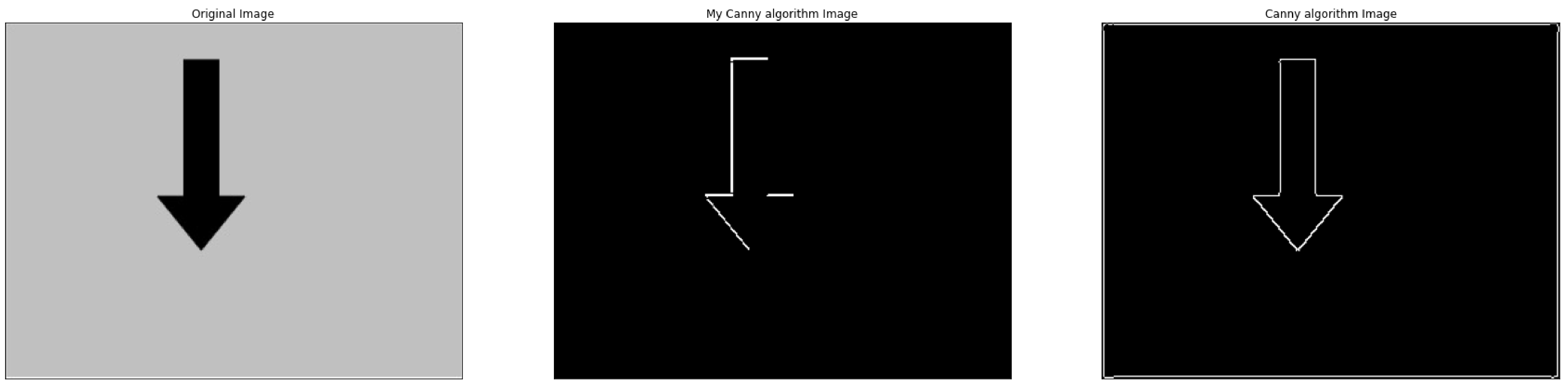
In [ ]:
```python
def edgeLinking(G_localmax, t_low, t_high):
    I_edges = np.zeros((G_localmax.shape))

    # Set the threshold image and perform edge linking (or hysteresis thresholding)
    frontier = []
    ON = 255
    for x in range(G_localmax.shape[0]):
        for y in range(G_localmax.shape[1]):
            if G_localmax[x, y] > t_high:
                frontier.insert(0,(x,y))
                I_edges[x,y] = ON
    while len(frontier) > 0:
        p = frontier.pop()
        for x in range(p[0]-1, p[0] + 1):
            if x < 0 or x >= I_edges.shape[0]:
                continue
            for y in range(p[1] - 1, p[1] + 1):
                if y < 0 or y >= I_edges.shape[1]:
                    continue
                if G_localmax[x, y] > t_low:
                    if I_edges[x,y] == 0:
                        frontier.insert(0,(x,y))
                        I_edges[x,y]= ON
    return I_edges
```

In [ ]:
```python
"""
Function that performs the Canny algorithm.

The entire cell is locked, thus you can only test the function and NOT change it!

Input:
    - Im: image in grayscale
    - t_low: first threshold for the hysteresis procedure (edge linking)
    - t_high: second threshold for the hysteresis procedure (edge linking)
"""
def my_cannyAlgorithm(Im, t_low, t_high):
    ## Compute the image gradient
    G_mag, G_phase = computeImageGradient(Im)

    ## NonMaxSuppression algorithm
    G_localmax = nonMaxSuppression(G_mag, G_phase)

    ## Edge linking
    if t_low>t_high: t_low, t_high = t_high, t_low
    I_edges = edgeLinking(G_localmax, t_low, t_high)

    plt.figure(figsize=(30,30))
    plt.subplot(141), plt.imshow(G_mag, cmap='gray')
    plt.title('Magnitude image.'), plt.xticks([]), plt.yticks([])
    plt.subplot(142), plt.imshow(G_phase, cmap='gray')
    plt.title('Phase image.'), plt.xticks([]), plt.yticks([])
    plt.subplot(143), plt.imshow(G_localmax, cmap='gray')
    plt.title('After non maximum suppression.'), plt.xticks([]), plt.yticks([])
    plt.subplot(144), plt.imshow(I_edges, cmap='gray')
    plt.title('Threshold image.'), plt.xticks([]), plt.yticks([])
    plt.show()

    return I_edges
```

In [ ]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

Im = cv2.imread("./images/arrow_1.jpg", cv2.IMREAD_GRAYSCALE)

t_low = 100
t_high = 250
I_edges = my_cannyAlgorithm(Im, t_low, t_high)
```



In [ ]:
```
# LOCKED cell: useful to check and visualize the results.

plt.figure(figsize=(30,30))
plt.subplot(131), plt.imshow(Im, cmap='gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(132), plt.imshow(I_edges, cmap='gray')
plt.title('My Canny algorithm Image'), plt.xticks([]), plt.yticks([])
plt.subplot(133), plt.imshow(cv2.Canny(Im,t_low, t_high, L2gradient=False), cmap='gray')
plt.title('Canny algorithm Image'), plt.xticks([]), plt.yticks([])
plt.show()
```



Problem 2

One of the most popular approaches to feature detection is the **Harris corner detector**, after a work of Chris Harris and Mike Stephens from 1988.

a) Use the function in opencv `cv2.cornerHarris(...)` (Documentation) with `blockSize=3, ksize=3, k=0.04` with the **./images/chessboard.png** image to detect the corners (you can find the image on CANVAS).

b) Plot the image with the detected corners found.

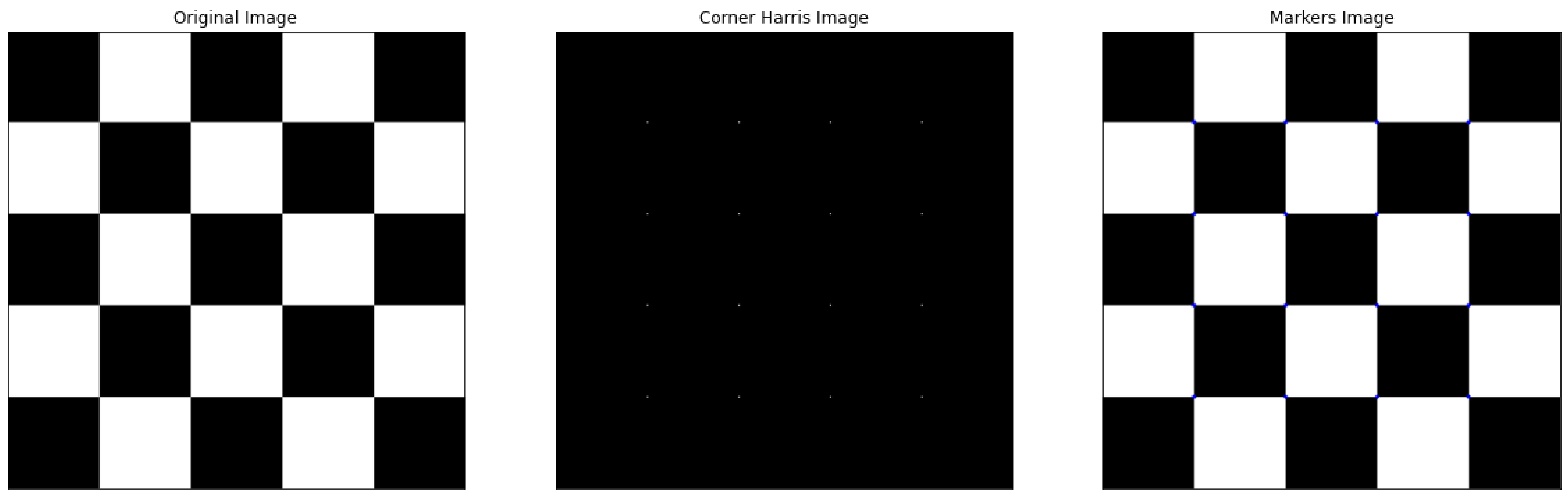
Hint: Use the function `cv2.drawMarker(...)` (Documentation) to show the corners in the image.

c) Detect the corners using the images **./images/arrow_1.jpg**, **./images/arrow_2.jpg** and **./images/arrow_3.jpg**; describe and compare the results in the three images.

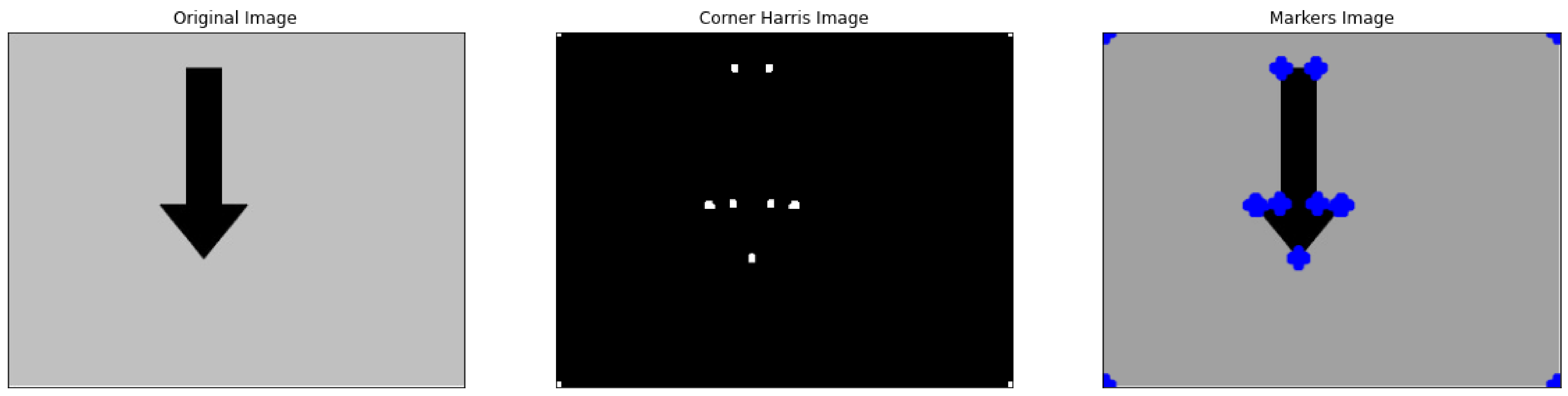
d) What happen if you change (increase/decrease) the `k` constant for the "corner points"?

```
In [ ]: # Answers go here
        def Detect_And_Process_Markers(img_path, k=0.04):
            # gray scale
            img_gray, img_color = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE), cv2.imread(img_path, cv2.IMREAD_COLOR)
            img_ch = cv2.dilate(cv2.cornerHarris(img_gray, blockSize=3, ksize=3, k=k), None)
            img_ch[img_ch > 0.001 * img_ch.max()] = 255

            # draw markers
            for i in range(img_ch.shape[0]):
                for j in range(img_ch.shape[1]):
                    if img_ch[i, j] == 255:
                        cv2.drawMarker(img_color, (j, i), (0, 0, 255), markerType=cv2.MARKER_CROSS, markerSize=10, thickness=2,

            plt.figure(figsize=(20,20))
            plt.subplot(131), plt.imshow(img_gray, cmap='gray')
            plt.title('Original Image'), plt.xticks([]), plt.yticks([])
            plt.subplot(132), plt.imshow(img_ch, cmap='gray')
            plt.title('Corner Harris Image'), plt.xticks([]), plt.yticks([])
            plt.subplot(133), plt.imshow(img_color, cmap='gray')
            plt.title('Markers Image'), plt.xticks([]), plt.yticks([])
            plt.show()
```

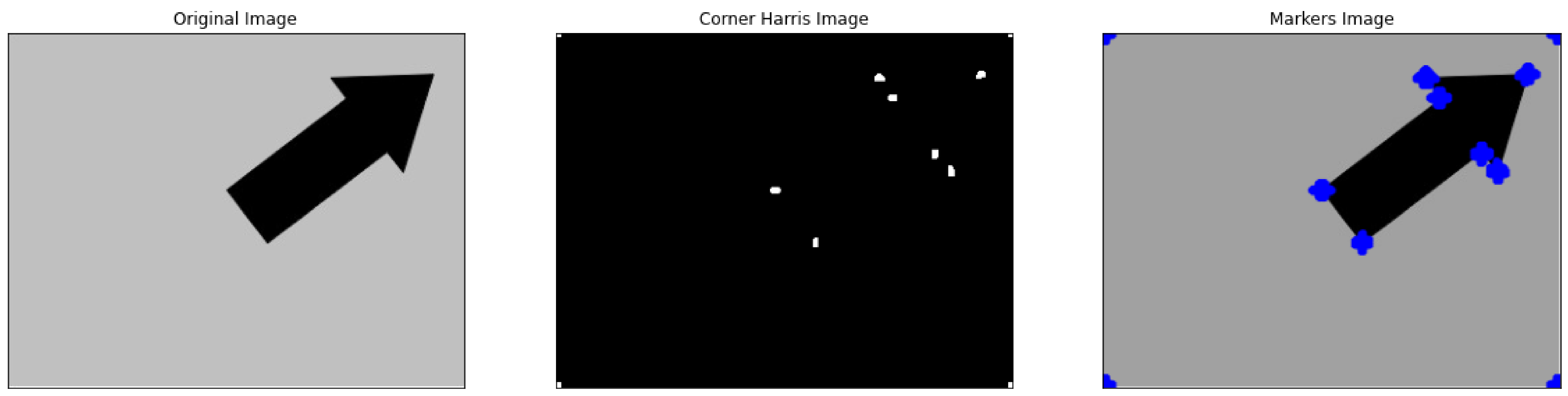
```
In [ ]: Detect_And_Process_Markers('images/chessboard.png')
```



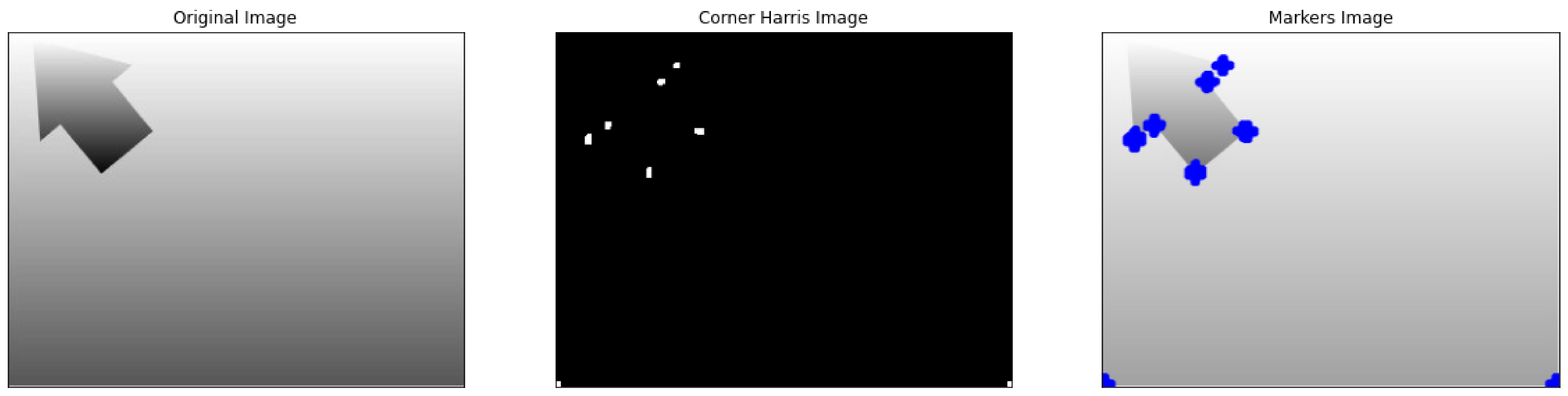
```
In [ ]: Detect_And_Process_Markers('images/arrow_1.jpg')
```



```
In [ ]: Detect_And_Process_Markers('images/arrow_2.jpg')
```



```
In [ ]: Detect_And_Process_Markers('images/arrow_3.jpg')
```

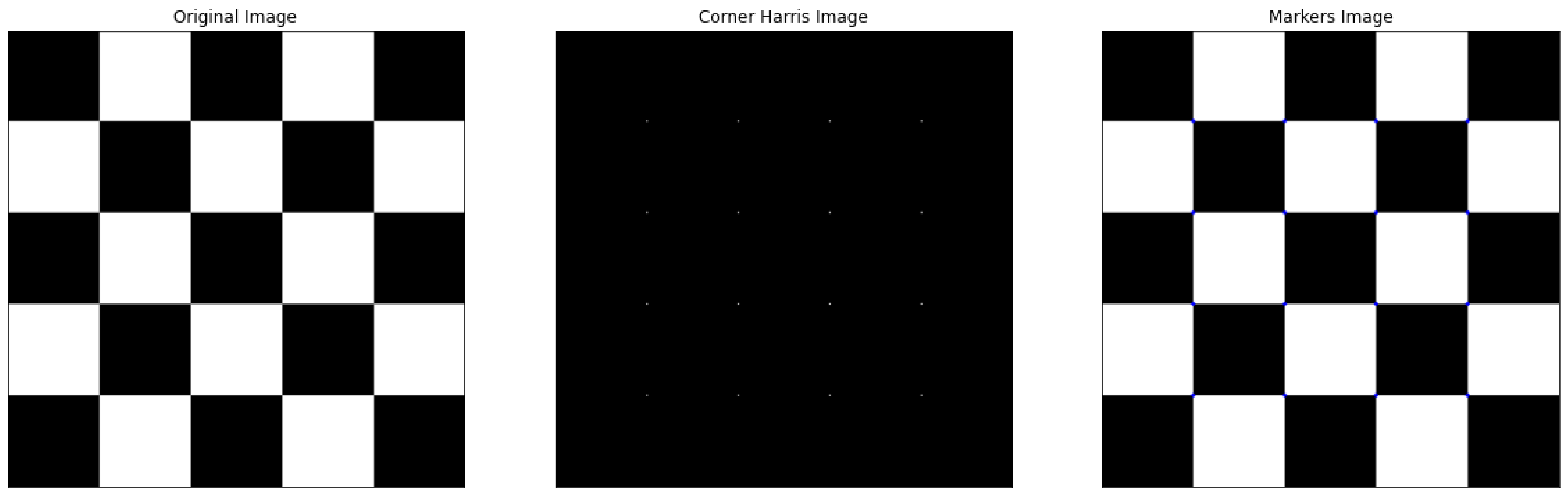


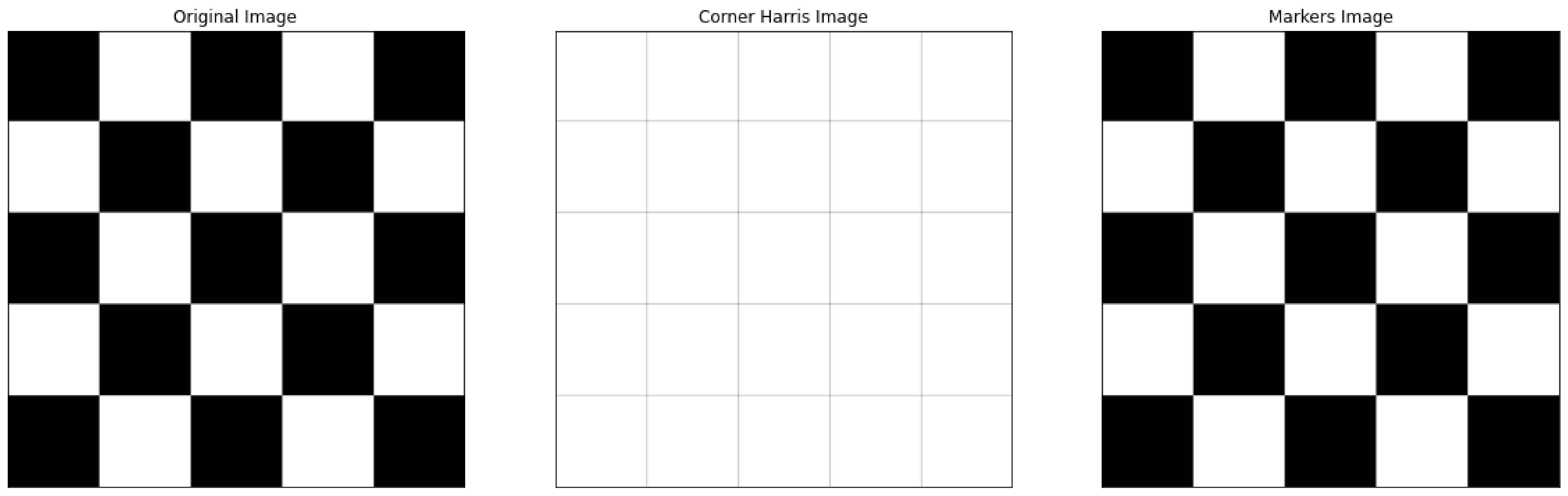
d) What happen if you change (increase/decrease) the `k` constant for the "corner points"?

An decrease of the 'k' constant results with the detection algorithm becoming more sensitive to changing pixels, which means that the algorithm will detect more corners.

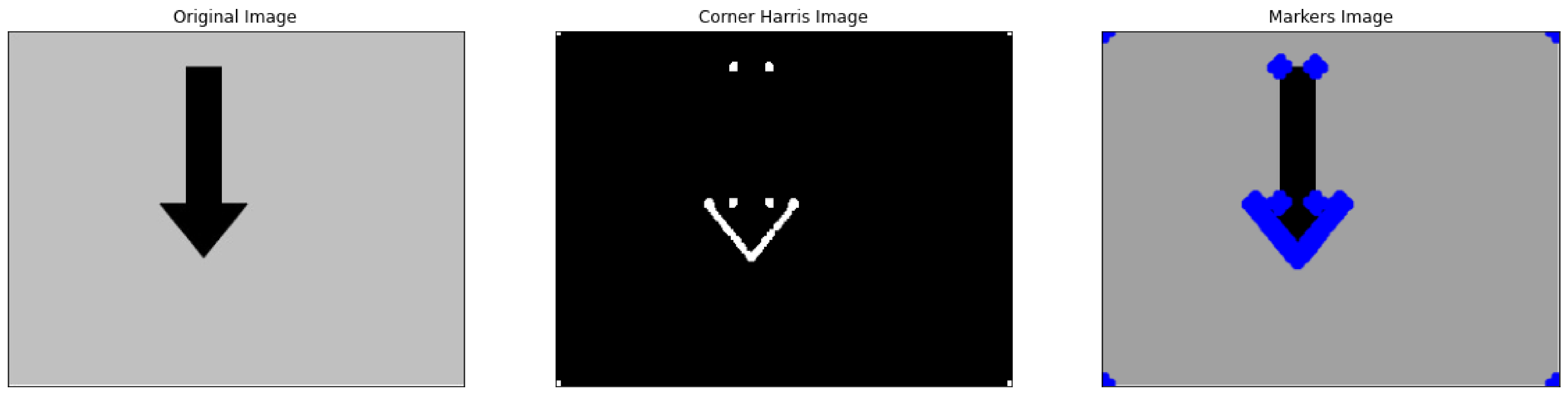
An increase of the 'k' constant results with the detection algorithm becoming less sensitive to changing pixels, which means that the algorithm will detect less corners.

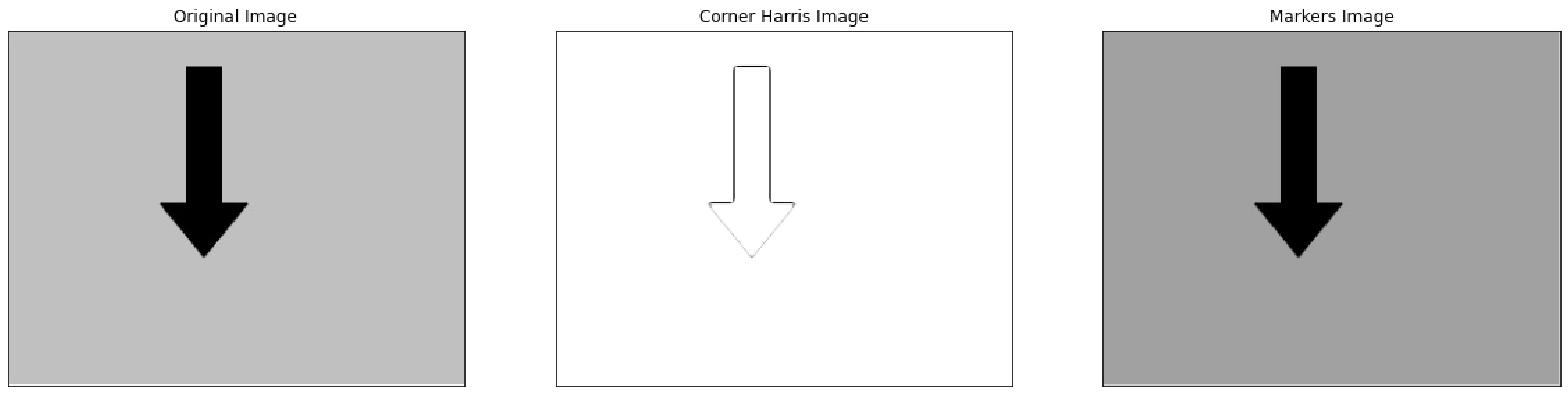
In [ ]:
```
Detect_And_Process_Markers('images/chessboard.png',k=0)
Detect_And_Process_Markers('images/chessboard.png',k=1)
```



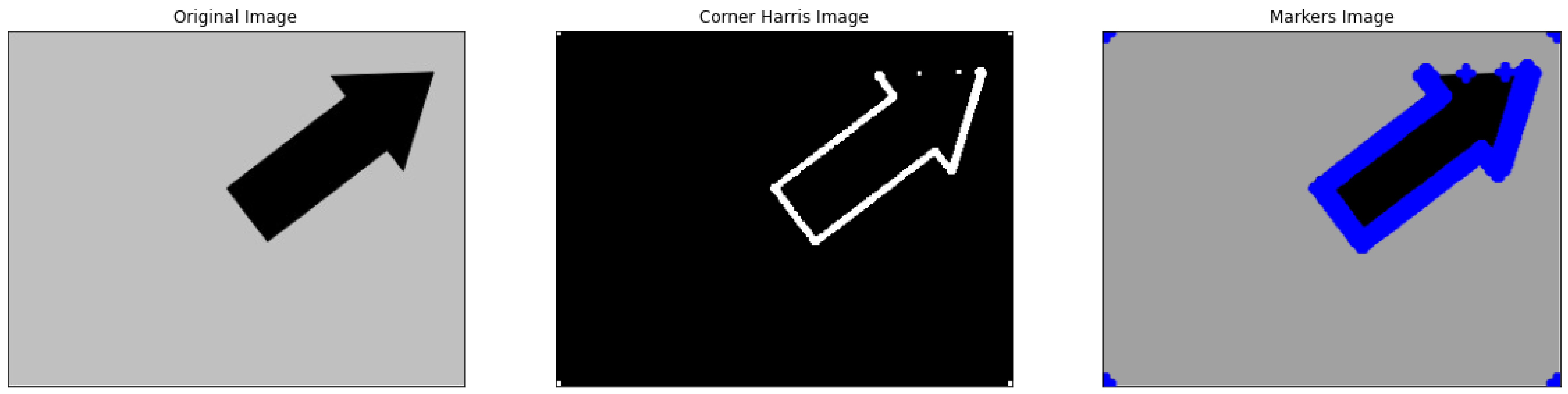


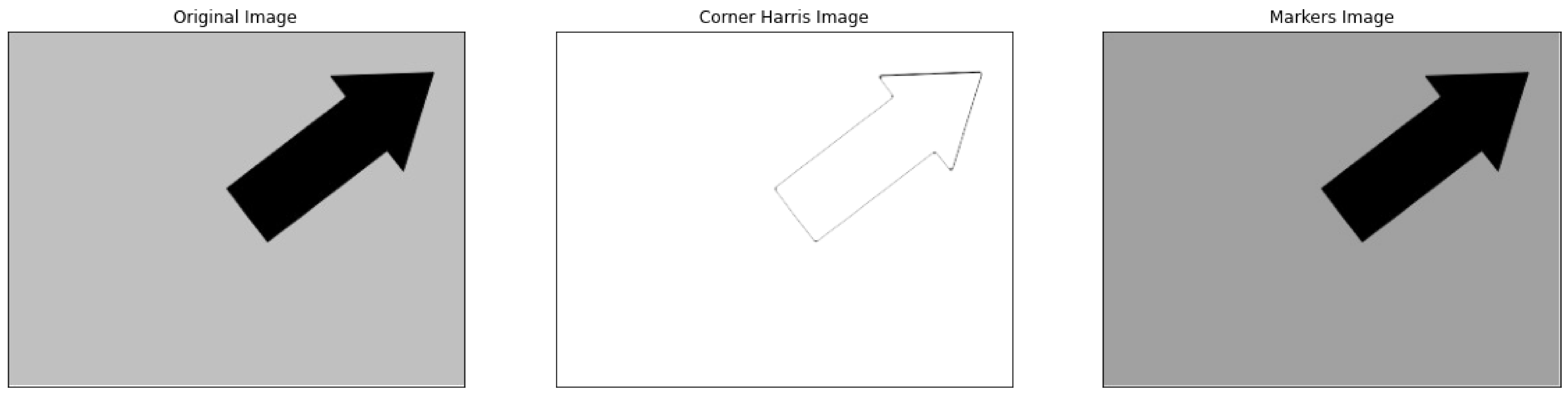
In [ ]:
```
Detect_And_Process_Markers('images/arrow_1.jpg',k=0)
Detect_And_Process_Markers('images/arrow_1.jpg',k=1)
```



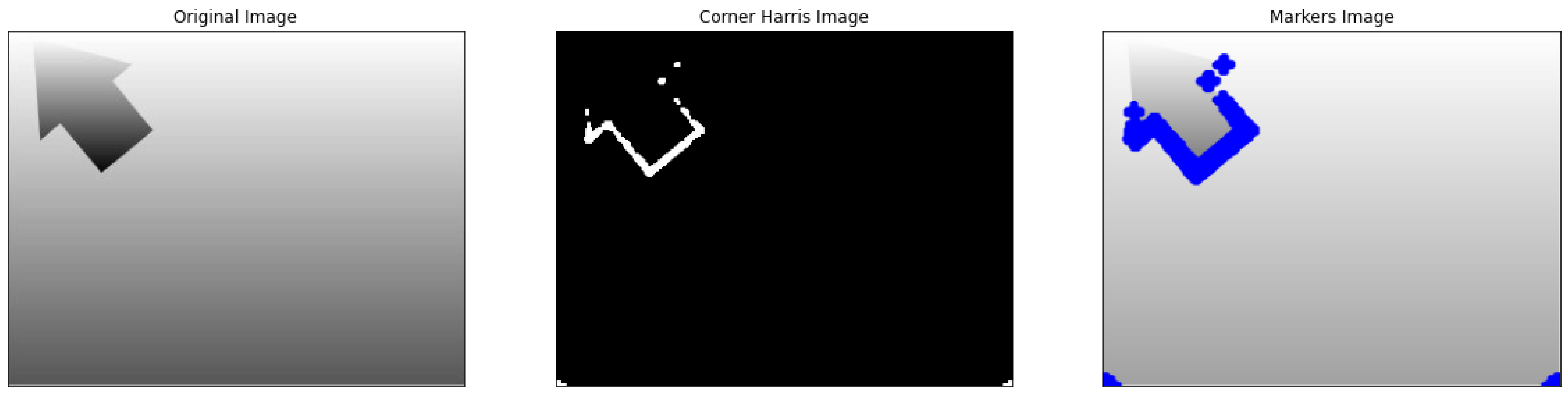


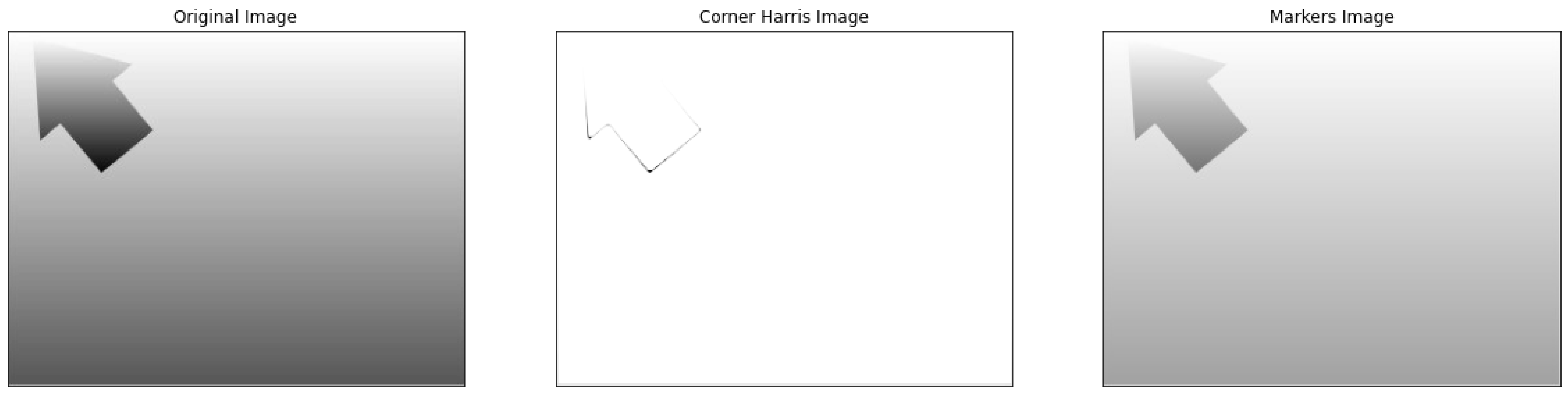
In [ ]:
```
Detect_And_Process_Markers('images/arrow_2.jpg',k=0)
Detect_And_Process_Markers('images/arrow_2.jpg',k=1)
```





In [ ]:
```
Detect_And_Process_Markers('images/arrow_3.jpg',k=0)
Detect_And_Process_Markers('images/arrow_3.jpg',k=1)
```





Problem 3

a) What is the SIFT approach? Describe the steps involved.

b) Why this approach is more popular than the Harris detector?

c) Explain the difference between a feature detector and a feature descriptor.

---

a) What is the SIFT approach? Describe the steps involved.

The SIFT approach is a feature detection algorithm that consists of the following steps:

1. Scale Space Construction
   The image is blurred with a Gaussian filter with different sigma values, and the result is stored in a scale space.
   Then it is down-scaled to produce a new image. This is repeated until the image is too small.
2. Difference of Gaussians
   The difference of Gaussians is calculated for each image in the scale space.
   The result is stored in a DoG scale space.
3. Extreme point extraction / detection The DoG scale space is scanned for local extrema.
   The local extrema are stored in a list of keypoints.
4. Keypoint localization
   After the previous step, the extracted keypoints might be to many.
   To reduce the number of keypoints, the keypoints are eliminated by the following criteria:
   - Low-Contrast KeyPoints.
   - Edge Response KeyPoints.
5. Orientation assignment
   The orientation of the keypoints is calculated.
   The orientation is calculated by calculating the gradient of the image at the keypoint location.
   This is done so that the keypoints can be used for rotation invariant matching.

b) Why this approach is more popular than the Harris detector?

This proccess is more popular than the Harris detector because it is more accurate and it is more robust to noise.

c) Explain the difference between a feature detector and a feature descriptor.

A feature detector is an algorithm that detects features in an image.
While a feature descriptor usually is a dataset, vector or matrix that contains the features of an image.

Delivery (dead line) on CANVAS: 14.10.2022 at 23:59

Contact

Course teacher

Professor Kjersti Engan, room E-431, E-mail: kjersti.engan@uis.no

Teaching assistant

Tomasetti Luca, room E-401 E-mail: luca.tomasetti@uis.no

Saul Fuster Navarro, room E-401 E-mail: saul.fusternavarro@uis.no

References

[1] S. Birchfeld, Image Processing and Analysis. Cengage Learning, 2016.

[2] I. Austvoll, "Machine/robot vision part I," University of Stavanger, 2018. Compendium, CANVAS.