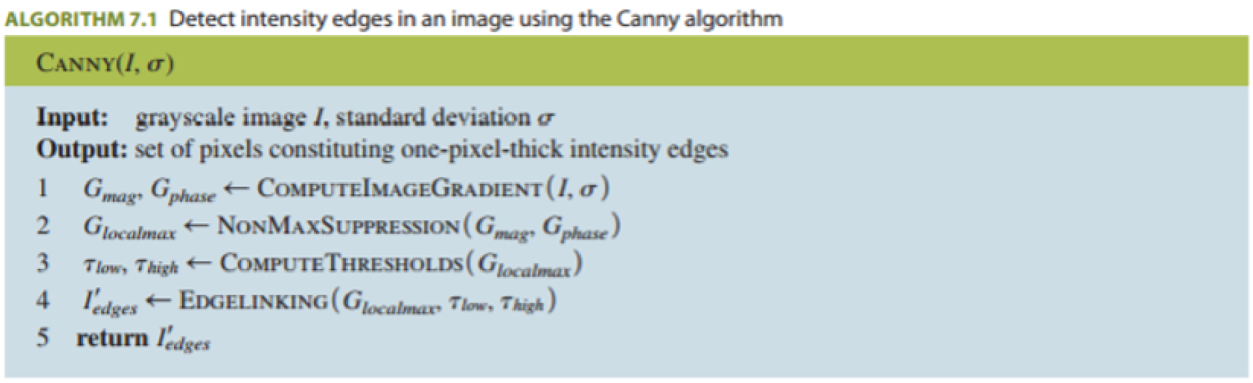
ELE510 Image Processing with robot vision: LAB, Exercise 6, Image features detection.

Problem 1

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

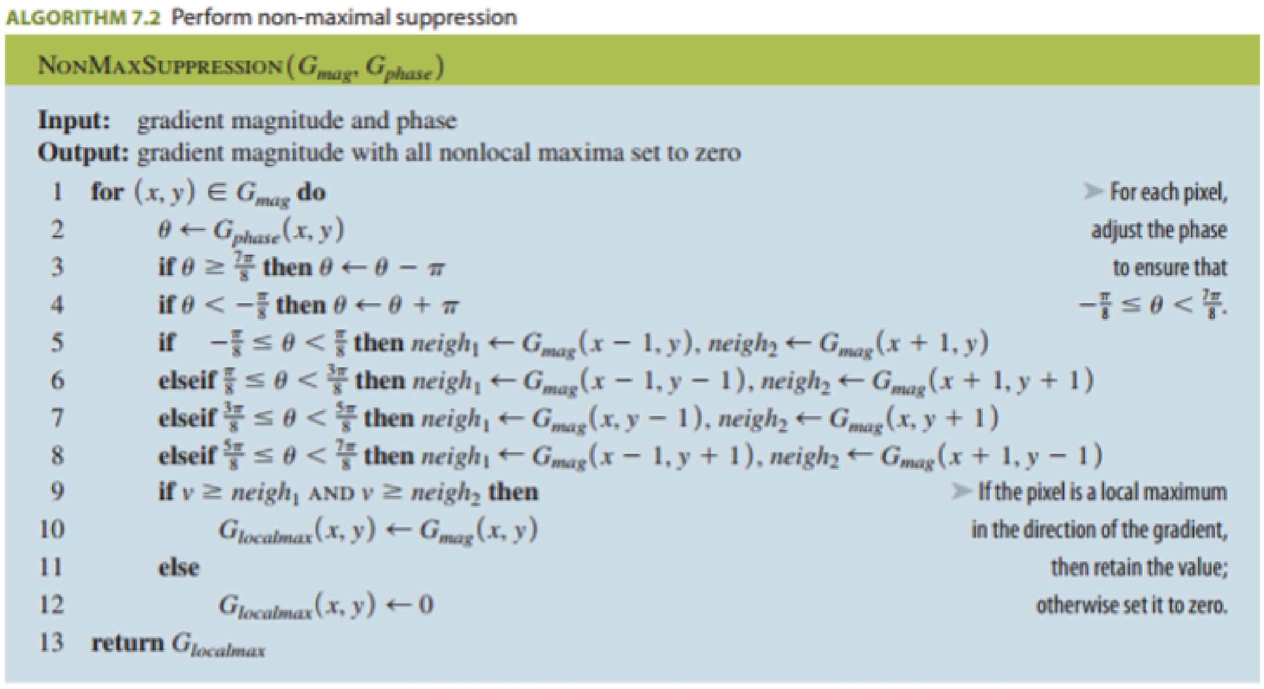
The **Canny edge detector** is a classic algorithm for detecting intensity edges in a grayscale image that relies on the gradient magnitude. The algorithm was developed by John F. Canny in 1986. It is a multi-stage algorithm that provides good and reliable detection.

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 [ ]:
```python
import cv2
import numpy as np

# Sobel operator to find the first derivate in the horizontal and vertical directions
def computeImageGradient(Im):
    # Sobel operator  to find the first derivate in the horizontal and vertical direc
    ## TODO: The default ksize is 3, try different values and comment the result
    g_x = cv2.Sobel(Im, cv2.CV_64F, 1, 0, ksize=3)
    g_y = cv2.Sobel(Im, cv2.CV_64F, 0, 1, ksize=3)


    ############################
    # Calculate the magnitude and the gradient direction like it is performed during
    G_mag = np.sqrt(g_x**2 + g_y**2)
    G_phase = np.arctan2(g_y, g_x)

    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
    for i in range(1, G_mag.shape[0] - 1):
        for j in range(1, G_mag.shape[1] - 1):
            angle = G_phase[i, j]

            # Ensure angle is within the range of -pi/8 to 7*pi/8
            if angle < -np.pi / 8:
                angle += np.pi

            if angle >= 7 * np.pi / 8:
                angle -= np.pi

            # Define neighboring pixel coordinates based on gradient direction
            if -np.pi / 8 <= angle < np.pi / 8 or 7 * np.pi / 8 <= angle:
                neighbor1 = G_mag[i, j - 1]
                neighbor2 = G_mag[i, j + 1]
            elif np.pi / 8 <= angle < 3 * np.pi / 8:
                neighbor1 = G_mag[i - 1, j - 1]
                neighbor2 = G_mag[i + 1, j + 1]
            elif 3 * np.pi / 8 <= angle < 5 * np.pi / 8:
                neighbor1 = G_mag[i - 1, j]
                neighbor2 = G_mag[i + 1, j]
            else:
                neighbor1 = G_mag[i - 1, j + 1]
                neighbor2 = G_mag[i + 1, j - 1]

            # Compare the current pixel's magnitude with its neighbors
            if G_mag[i, j] >= neighbor1 and G_mag[i, j] >= neighbor2:
                G_localmax[i, j] = G_mag[i, j]


    return G_localmax
```

In [ ]:
```python
def edgeLinking(G_localmax, t_low, t_high):
    width, height = G_localmax.shape
    I_edges = np.zeros((width, height), dtype=int)  # Initialize the binary image for
    frontier = []  # Initialize the frontier as an empty list

    # Step 1: Push initial pixels into the frontier
    for x in range(width):
        for y in range(height):
            if G_localmax[x, y] > t_high:
                frontier.append((x, y))
                I_edges[x, y] = 1  # Mark as an edge pixel

    # Step 5: Edge linking process
    while frontier:
        x, y = frontier.pop()

        # Define 8-neighbor coordinates
        neighbors = [(x - 1, y - 1), (x - 1, y), (x - 1, y + 1),
                     (x, y - 1),                 (x, y + 1),
                     (x + 1, y - 1), (x + 1, y), (x + 1, y + 1)]

        for neighbor_x, neighbor_y in neighbors:
            if 0 <= neighbor_x < width and 0 <= neighbor_y < height:
                if G_localmax[neighbor_x, neighbor_y] > t_low and I_edges[neighbor_x,
                    frontier.append((neighbor_x, neighbor_y))
```

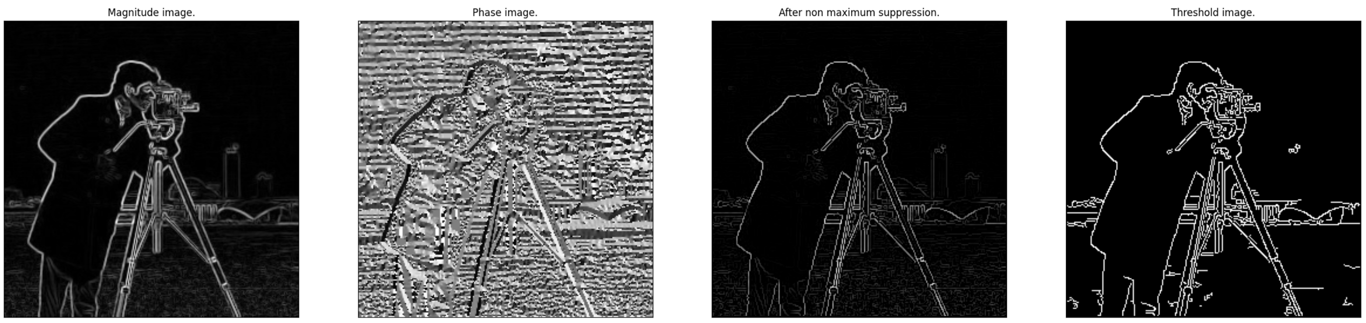
```
                        I_edges[neighbor_x, neighbor_y] = 1  # Mark as an edge pixel

    return I_edges
```

In [ ]:

```
"""
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
```

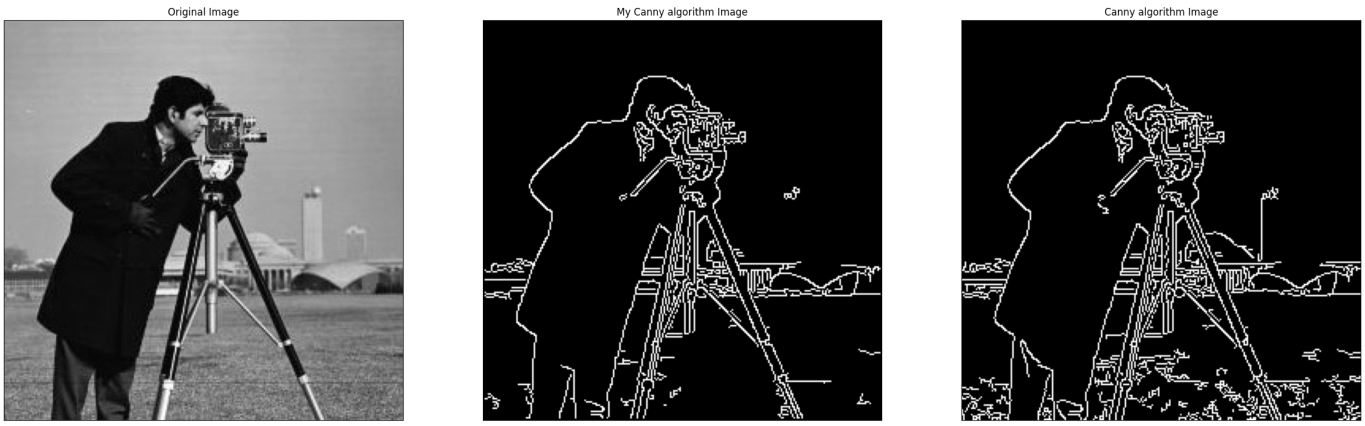
In [ ]:

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

Im = cv2.imread("images/cameraman.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='gra
plt.title('Canny algorithm Image'), plt.xticks([]), plt.yticks([])
plt.show()
```



The two outcomes of the Canny algorithm are very similar, only few differences appear: the most noticable one is the side of the background turrent disappears in my version, on the other hand there is more noise in the grass behind the subject.

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.

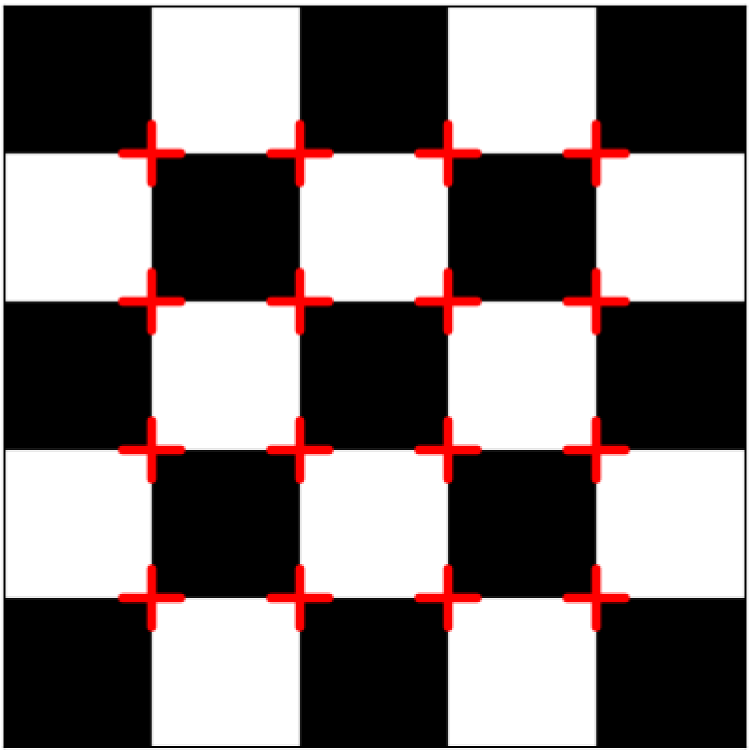
c) Detect the corners using the images **./images/arrow_1.jpg**, **./images/arrow_2.jpg** and **./images/arrow_3.jpg**; describe and compare the results in the three images.

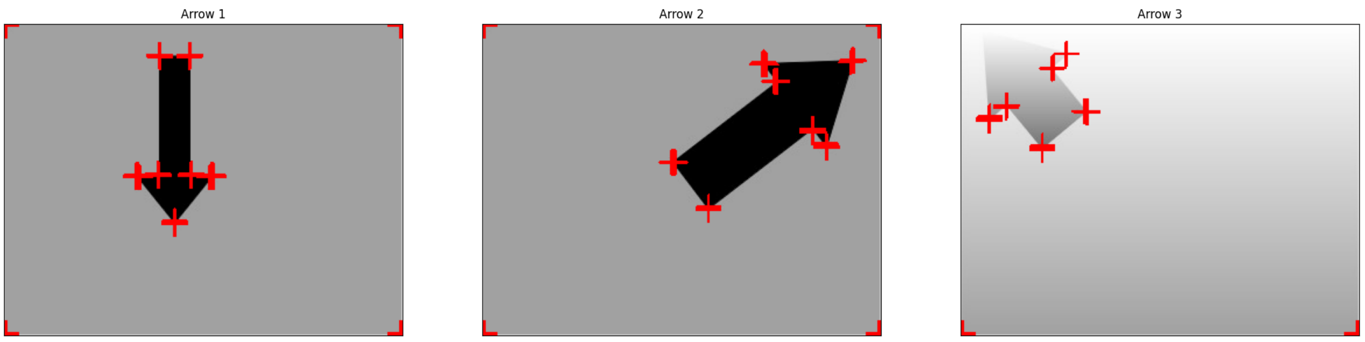
d) What happen if you change (increase/decrease) the `k` constant for the "corner points"?

In [ ]:
```
def detect_and_draw_corners(image, marker_size, thikness, block_size=3, ksize=3, k=0.
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    harris = cv2.cornerHarris(gray, block_size, ksize, k)

    corners = np.where(harris > 0.001 * harris.max())

    if verbose:
        #print the number of corners found
        print("Number of corners found: {}".format(len(corners[0])))

    # Create an image copy for drawing markers
    image_with_markers = image.copy()

    for y, x in zip(*corners):
        cv2.drawMarker(image_with_markers, (x, y), color=(255, 0, 0), markerType=cv2.
    return image_with_markers
```

In [ ]:
```
image = cv2.imread("images/chessboard.png")

image_with_markers = detect_and_draw_corners(image, 150, 20)

# Display the images
plt.imshow(image_with_markers)
plt.title('Image with Detected Corners'), plt.xticks([]), plt.yticks([])
plt.show()

# Load the additional images
arrow_images = [
    cv2.imread("./images/arrow_1.jpg"),
    cv2.imread("./images/arrow_2.jpg"),
    cv2.imread("./images/arrow_3.jpg")
]

# Process and display the corner detection results for each arrow image
plt.figure(figsize=(25, 20))
for i, arrow_image in enumerate(arrow_images):
    result_image = detect_and_draw_corners(arrow_image, 20, 1)
    plt.subplot(1, 3, i + 1)
    plt.imshow(result_image)
    plt.title(f'Arrow {i+1}')
    plt.xticks([]), plt.yticks([])

plt.show()
```

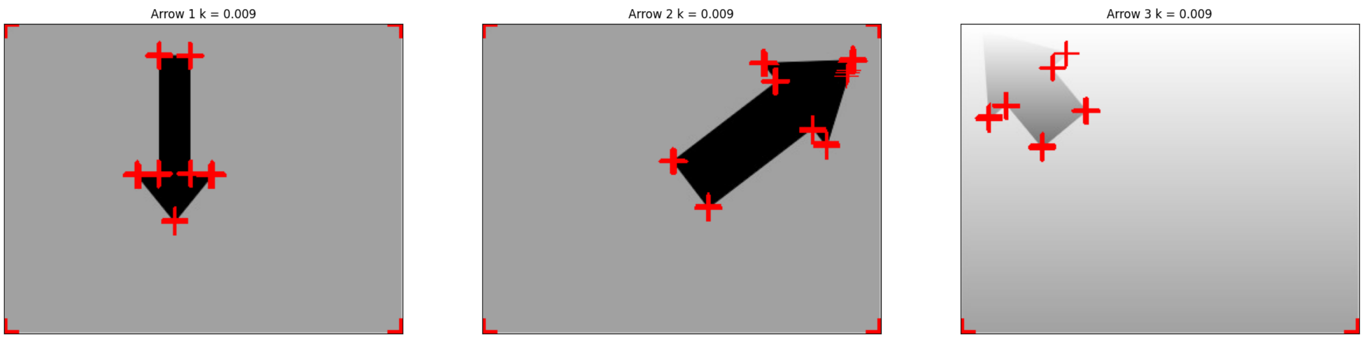
Image with Detected Corners



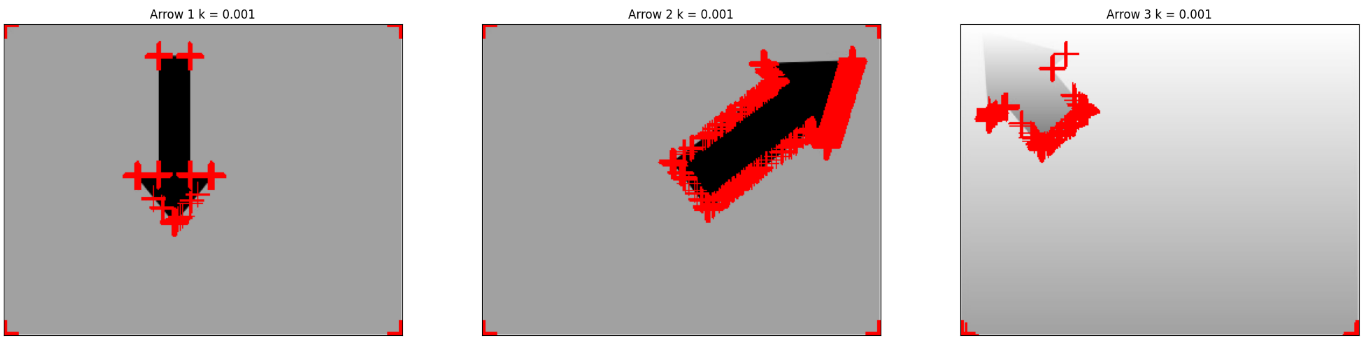


In [ ]:
```
block_size = 3
kernel_size = 3
k = 0.009

plt.figure(figsize=(25, 20))
for i, arrow_image in enumerate(arrow_images):
    result_image = detect_and_draw_corners(arrow_image, 20, 1, block_size=block_size,
    plt.subplot(1, 3, i + 1)
    plt.imshow(result_image)
    plt.title(f'Arrow {i+1} k = 0.009')
    plt.xticks([]), plt.yticks([])

plt.show()

block_size = 3
kernel_size = 3
k = 0.001

plt.figure(figsize=(25, 20))
for i, arrow_image in enumerate(arrow_images):
    result_image = detect_and_draw_corners(arrow_image, 20, 1, block_size=block_size,
    plt.subplot(1, 3, i + 1)
    plt.imshow(result_image)
    plt.title(f'Arrow {i+1} k = 0.001')
    plt.xticks([]), plt.yticks([])

plt.show()
```

```
Number of corners found: 146
Number of corners found: 155
Number of corners found: 97
```



```
Number of corners found: 186
Number of corners found: 683
Number of corners found: 245
```



The k value in the Harris corner detector influences how the detector evaluates the corner response of pixels. Adjusting k allows you to control the balance between the impact of intensity variations and the trace of the gradient covariance matrix.

$$\text{dst}(x, y) = \det(M(x, y)) - k \cdot (\text{trace}(M(x, y)))^2 \tag{1}$$

dst(x, y) represents the corner response at pixel (x, y).

det(M(x, y)) is the determinant of the gradient covariance matrix at pixel (x, y).

k is a scaling factor.

trace(M(x, y)) is the trace of the gradient covariance matrix at pixel (x, y), which is the sum of its diagonal elements.

Changing the value from 0.009 to 0.001 as seen in the code above shows the different results in the number of corner found by the detector. The lower the k value the more corners are found going for example in the second arrow from 155 to 683 corners.

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) The SIFT (Scale-Invariant Feature Transform) Approach:

SIFT is a feature-based object recognition method. It is designed to identify and describe local features in images, which are distinctive and invariant to changes in scale, rotation, and illumination. The SIFT approach involves several key steps:

- Scale-space Extrema Detection:

  In this step, the image is progressively smoothed with Gaussian filters at different scales. This creates an image pyramid with varying levels of blur. At each scale level, the algorithm looks for local extrema (maxima or minima) in the difference of Gaussian (DoG) images. These extrema represent potential key points.

- Keypoint Localization:

  For each candidate keypoint, SIFT performs precise localization by fitting a 3D quadratic function to the nearby DoG values. This localization process helps to refine the keypoint's position and scale.

- Orientation Assignment:

  SIFT assigns an orientation to each keypoint based on local image gradients. This allows SIFT descriptors to be rotation-invariant.

- Descriptor Extraction:

  A descriptor is computed for each keypoint. This descriptor is based on the gradient information in a local region around the keypoint. It encodes information about the keypoint's appearance and its surroundings.

- Keypoint Matching:

  To match keypoints between images, SIFT uses a distance metric (e.g., Euclidean distance) to compare the descriptors of keypoints in different images. Keypoints with the most similar descriptors are considered matches.

b) Popularity of SIFT vs. Harris Detector:

SIFT is more popular than the Harris corner detector for several reasons:

Invariance: SIFT features are designed to be invariant to scale, rotation, and minor changes in illumination. This makes them highly reliable in a wide range of situations, whereas the Harris corner detector is less invariant.

SIFT descriptors provide rich and distinctive information about keypoints, making them well-suited for accurate matching between images, it can also handle variations in viewpoint, lighting, and occlusion, which are common challenges in object recognition.

Harris Detector might have an improvement in distinguishing between edges and corners which in many situation can be useful, but it is not as robust as SIFT.

c) Difference between Feature Detector and Feature Descriptor:

A feature detector is responsible for finding distinctive points or regions in an image while feature detectors locate positions in an image that are unique or salient, making them suitable for matching and recognition tasks. For example, the Harris corner detector is a feature detector.

A feature descriptor, on the other hand, provides a quantitative representation of the local image content around a detected keypoint. It encodes information about the intensity and gradient of pixels in the vicinity of the keypoint. Feature descriptors are used to create a compact and distinctive representation of the region around a keypoint, which can be used for matching and recognition. SIFT descriptors are a prominent example of feature descriptors.

Delivery (dead line) on CANVAS: 13.10.2023 at 23:59

Contact

Course teacher

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

Teaching assistant

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

Jorge Garcia Torres Fernandez, room E-401 E-mail: jorge.garcia-torres@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.