Assignment #3

HARNEET SINGH - 400110275

COMPENG 4TN4: Image Processing

February 13, 2022

Theory

1. Edge Detection, Sobel Operator

Detecting changes in intensity can be used to find edges and this can be achieved using first- or second-order derivatives. Image gradient tells us about the edge strength and direction at (x, y) location in the image. At a point, gradient points in the direction of maximum rate of change and when we use gradient over an entire image, vectors (composed of magnitude and phase) can be seen in the image which correlate to edges.
Note that the direction of an edge and gradient vector (edge normal) at a point are orthogonal to one another. For example, if we have a horizontal edge, then the gradient vector would be pointing either up or down (depending on the intensities in either directions).
Since Sobel operators have a sum of zero, the areas with constant intensity will yield zero vector. To compute the gradients, we need to find partial derivatives in x and y directions. This can be done using the Sobel operators which mimic the partial derivative operations.
Following example explains how the magnitude and phase of the gradient are calculated, however a python script will be used to compute gradient vector values for all the pixels.
For this particular example, I will be choosing pixel at (3, 4) location, that has a value of 1.

$$Sobel: \ M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} and \ M_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & 1 \end{bmatrix}$$

From the given image, picking sub-matrix from location (3, 4) as the center position, we get:

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	0	0
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	0	0	1	1	1	1	1	0	0
0	0	0	0	0	0	1	1	1	1	0	0
0	0	1	0	0	0	0	1	1	1	0	0
0	0	0	1	0	0	0	0	1	1	0	0
0	0	0	0	1	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

$$Sub - matrix \ = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Applying the Sobel operators: Convolving sub-matrix with both M_x and M_y gives gradient in x (G_x) and gradient in y G_y:
$G_x = 0 * (-1) + 0 * 1 + 0 * 1 + 0 * (-2) + 1 * 0 + 1 * 2 + 0 * (-1) + 0 * 0 + 1 * 1 = 3$
$G_y = 0 * 1 + 0 * 2 + 0 * 1 + 0 * 0 + 1 * 0 + 1 * 0 + 0 * (-1) + 0 * (-2) + 1 * 1 = 1$

Magnitude of the gradient $= \sqrt{{G_x}^2 + {G_y}^2} = \sqrt{3^2 + 1^2} = \sqrt{10} = 3.2 \ (approx.)$

Phase of the gradient $= tan^{-1}\left(\frac{G_y}{G_x}\right) = tan^{-1}\left(\frac{1}{3}\right) = 18.4°$

Now, this methodolgy can be applied on the entire image using Python code, which is shown below:

```
import numpy as np

# creating image array
img = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0],
                [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0],
                [0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0],
                [0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0],
                [0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
imgShape = int(img.shape[0])

# creating Sobel operators
kernelSobelX = np.array([[-1, 0, 1],
                         [-2, 0, 2],
                         [-1, 0, 1]])
kernelSobelY = np.copy(kernelSobelX)
kernelSobelY = np.transpose(kernelSobelY)
kernelShape = int(kernelSobelX.shape[0])

# applying Sobel operators to the image
def sobelFilter(imgSize, n, kernel):
    filteredImg = np.zeros_like(img)

    for k in range(imgSize - n + 1):
        for p in range(imgSize - n + 1):
            subMatrix = np.zeros((n, n))
            subMatrix = img[k:k + n, p:p + n]
            multipliedValue = np.multiply(subMatrix, kernel)
            sumAllElements = np.sum(multipliedValue)
```

```
                filteredImg[k + 1, p + 1] = sumAllElements
        return filteredImg

# finding gradient and phase
def gradientMagPhase(gradX, gradY):
    gradXSquare = np.square(gradX)
    gradYSquare = np.square(gradY)
    gradxAddGradY = np.add(gradXSquare, gradYSquare)

    gMagnitude = np.sqrt(gradxAddGradY)
    gPhase = np.arctan2(gradY, gradX) * 180 / np.pi

    return gMagnitude, gPhase


# Computing the gradient in x and y directions
Gx = sobelFilter(imgShape, kernelShape, kernelSobelX)
Gy = sobelFilter(imgShape, kernelShape, kernelSobelY)
# print(f"Gx =\n {Gx}")
# print(f"Gy =\n {Gy}")

# Computing the gradient magnitude and phase in degree
gradMag, gradPhase = gradientMagPhase(Gx, Gy)
gradMag = np.around(gradMag, decimals=1)
gradPhase = np.around(gradPhase, decimals=1)
# print(f"Mag =\n {gradMag}")
print(f"Phase =\n {gradPhase}")
```

Results:
After applying the Sobel operators:

G_x:

```
Gx =
 [[ 0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  1  1  0  0  0  0  0 -1 -1  0]
 [ 0  0  2  3  1  0  0  0  0 -3 -3  0]
 [ 0  0  1  3  3  1  0  0  0 -4 -4  0]
 [ 0  0  0  1  3  3  1  0  0 -4 -4  0]
 [ 0  1  0 -1  1  3  3  1  0 -4 -4  0]
 [ 0  2  1 -2 -1  1  3  3  1 -4 -4  0]
 [ 0  1  2  0 -2 -1  1  3  3 -3 -4  0]
 [ 0  0  1  2  0 -2 -1  1  3 -1 -3  0]
 [ 0  0  0  1  2 -1 -2  0  1  0 -1  0]
 [ 0  0  0  0  1  0 -1  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0]]
```

G_y:

```
Gy =
 [[ 0  0  0  0  0  0  0  0  0  0  0  0]
 [ 0  0  1  3  4  4  4  4  4  3  1  0]
 [ 0  0  0  1  3  4  4  4  4  3  1  0]
 [ 0  0 -1 -3 -3 -1  0  0  0  0  0  0]
 [ 0  0  0 -1 -3 -3 -1  0  0  0  0  0]
 [ 0  1  2  1 -1 -3 -3 -1  0  0  0  0]
 [ 0  0  1  2  1 -1 -3 -3 -1  0  0  0]
 [ 0 -1 -2  0  2  1 -1 -3 -3 -1  0  0]
 [ 0  0 -1 -2  0  2  1 -1 -3 -3 -1  0]
 [ 0  0  0 -1 -2 -1  0  0 -1 -2 -1  0]
 [ 0  0  0  0 -1 -2 -1  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0  0  0]]
```

Gradient Magnitude:

$$||\Delta f|| = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.4 & 3.2 & 4 & 4 & 4 & 4 & 4 & 3.2 & 1.4 & 0 \\ 0 & 0 & 2 & 3.2 & 3.2 & 4 & 4 & 4 & 4 & 4.2 & 3.2 & 0 \\ 0 & 0 & 1.4 & 4.2 & 4.2 & 1.4 & 0 & 0 & 0 & 4 & 4 & 0 \\ 0 & 0 & 0 & 1.4 & 4.2 & 4.2 & 1.4 & 0 & 0 & 4 & 4 & 0 \\ 0 & 1.4 & 2 & 1.4 & 1.4 & 4.2 & 4.2 & 1.4 & 0 & 4 & 4 & 0 \\ 0 & 2 & 1.4 & 2.8 & 1.4 & 1.4 & 4.2 & 4.2 & 1.4 & 4 & 4 & 0 \\ 0 & 1.4 & 2.8 & 0 & 2.8 & 1.4 & 1.4 & 4.2 & 4.2 & 3.2 & 4 & 0 \\ 0 & 0 & 1.4 & 2.8 & 0 & 2.8 & 1.4 & 1.4 & 4.2 & 3.2 & 3.2 & 0 \\ 0 & 0 & 0 & 1.4 & 2.8 & 1.4 & 2 & 0 & 1.4 & 2 & 1.4 & 0 \\ 0 & 0 & 0 & 0 & 1.4 & 2 & 1.4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Gradient Phase: (in degree)

$$\theta = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 45 & 71.6 & 90 & 90 & 90 & 90 & 90 & 108.4 & 135 & 0 \\ 0 & 0 & 0 & 18.4 & 71.6 & 90 & 90 & 90 & 90 & 135 & 161.6 & 0 \\ 0 & 0 & -45 & -45 & -45 & -45 & 0 & 0 & 0 & 180 & 180 & 0 \\ 0 & 0 & 0 & -45 & -45 & -45 & -45 & 0 & 0 & 180 & 180 & 0 \\ 0 & 45 & 90 & 135 & -45 & -45 & -45 & -45 & 0 & 180 & 180 & 0 \\ 0 & 0 & 45 & 135 & 135 & -45 & -45 & -45 & -45 & 180 & 180 & 0 \\ 0 & -45 & -45 & 0 & 135 & 135 & -45 & -45 & -45 & -161.6 & 180 & 0 \\ 0 & 0 & -45 & -45 & 0 & 135 & 135 & -45 & -45 & -108.4 & -161.6 & 0 \\ 0 & 0 & 0 & -45 & -45 & -135 & 180 & 0 & -45 & -90 & -135 & 0 \\ 0 & 0 & 0 & 0 & -45 & -90 & -135 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

As you can see, our calculation of magnitude as 3.2 and phase as 18.4° can be seen in the row 3 and column 4.

2. Edge Detection, Canny Operator

a.

Technique from Theory 1 is used to find gradient in this part as well. Python Script shown below:

```python
import numpy as np

# creating the given image
img = np.zeros((8,8))
givenImg = np.array([[127, 127, 127, 0, 0, 0],
                     [0, 127, 127, 127, 0, 0],
                     [0, 0, 80, 80, 80, 0],
                     [0, 0, 0, 127, 127, 127],
                     [0, 0, 0, 0, 127, 127],
                     [0, 0, 0, 0, 0, 0]])
img[1:7, 1:7] = givenImg
imgShape = int(img.shape[0])

# creating Sobal operators
kernelSobelX = np.array([[-1, 0, 1],
                         [-2, 0, 2],
                         [-1, 0, 1]])
kernelSobelY = np.copy(kernelSobelX)
kernelSobelY = np.transpose(kernelSobelY)
kernelShape = int(kernelSobelX.shape[0])

# applying Sobel filter
def sobelFilter(imgSize, n, kernel):
    filteredImg = np.zeros_like(img)

    for k in range(imgSize - n + 1):
        for p in range(imgSize - n + 1):
            subMatrix = np.zeros((n, n))
            subMatrix = img[k:k + n, p:p + n]
            multipliedValue = np.multiply(subMatrix, kernel)
            sumAllElements = np.sum(multipliedValue)

            filteredImg[k + 1, p + 1] = sumAllElements
    return filteredImg
```

```python
# applying magnitude and phase of the gradient
def gradientMagPhase(gradX, gradY):
    gradXSquare = np.square(gradX)
    gradYSquare = np.square(gradY)
    gradxAddGradY = np.add(gradXSquare, gradYSquare)

    gMagnitude = np.sqrt(gradxAddGradY)
    gPhase = np.arctan2(gradY, gradX) * 180 / np.pi

    return gMagnitude, gPhase


# Computing the gradient in x and y directions
Gx = sobelFilter(imgShape, kernelShape, kernelSobelX)
Gy = sobelFilter(imgShape, kernelShape, kernelSobelY)
# print(f"Gx =\n {Gx}")
# print(f"Gy =\n {Gy}")

# Computing the gradient magnitude and phase in degree
gradMag, gradPhase = gradientMagPhase(Gx, Gy)
print(f"Mag =\n {gradMag}")
# print(f"Phase =\n {gradPhase}")
```

Script Output:

G_x:

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	381	127	-254	-381	-127	0	0
2	0	381	334	-47	-381	-334	-80	0
3	0	127	287	287	0	-287	-287	0
4	0	0	80	334	381	47	-461	0
5	0	0	0	127	381	254	-381	0
6	0	0	0	0	127	127	-127	0
7	0	0	0	0	0	0	0	0

G_y:

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	127	381	508	381	127	0	0
2	0	-381	-428	-141	193	240	80	0
3	0	-127	-381	-381	0	381	381	0
4	0	0	-80	-240	-193	141	301	0
5	0	0	0	-127	-381	-508	-381	0
6	0	0	0	0	-127	-381	-381	0
7	0	0	0	0	0	0	0	0

Gradient Magnitude: (rounded to 1 decimal place)

$$||\Delta f|| = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 401.6 & 401.6 & 568 & 538.8 & 179.6 & 0 & 0 \\ 0 & 538.8 & 542.9 & 148.6 & 427.1 & 411.3 & 113.1 & 0 \\ 0 & 179.6 & 477 & 477 & 0 & 477 & 477 & 0 \\ 0 & 0 & 113.1 & 411.3 & 427.1 & 148.6 & 550.6 & 0 \\ 0 & 0 & 0 & 179.6 & 538.8 & 568 & 538.8 & 0 \\ 0 & 0 & 0 & 0 & 179.6 & 401.6 & 401.6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Gradient Phase: (in degree) - rounded to 1 decimal place

$$\theta = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 18.4 & 71.6 & 116.6 & 135 & 135 & 0 & 0 \\ 0 & -45 & -52 & -108.4 & 153.1 & 144.3 & 135 & 0 \\ 0 & -45 & -53 & -53 & 0 & 127 & 127 & 0 \\ 0 & 0 & -45 & -35.7 & -26.9 & 71.6 & 146.9 & 0 \\ 0 & 0 & 0 & -45 & -45 & -63.4 & -135 & 0 \\ 0 & 0 & 0 & 0 & -45 & -71.6 & -108.4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

b. Non Maximum Suppression

Non Maximum suppression is applied using the gradient magnitude which is computed in part a. Based on the gradient angle, adjacent gradient values are compared. If the gradient magnitude is less than either or both of the adjacent pixels gradient magnitude, then it is swapped with a value of zero. For example, if the edge normal vector is pointing down, then the middle left and middle right values are compared with the middle value. Read *nonMaximumSupprression* for all the values. Following script is used to compute part b and c:

Python Script:

```python
import numpy as np
import cv2

lowThreshold = 20
highThreshold = 120

img = np.zeros((8, 8))
givenImg = np.array([[127, 127, 127, 0, 0, 0],
                     [0, 127, 127, 127, 0, 0],
                     [0, 0, 80, 80, 80, 0],
                     [0, 0, 0, 127, 127, 127],
                     [0, 0, 0, 0, 127, 127],
                     [0, 0, 0, 0, 0, 0]])
img[1:7, 1:7] = givenImg
# img = img.astype('uint8')
imgShape = int(img.shape[0])

kernelSobelX = np.array([[-1, 0, 1],
                         [-2, 0, 2],
                         [-1, 0, 1]])
kernelSobelY = np.copy(kernelSobelX)
kernelSobelY = np.transpose(kernelSobelY)
kernelShape = int(kernelSobelX.shape[0])

# applying gaussian filter to the image
def gaussianFilter(img, kernelSize):
    blurredImg = cv2.GaussianBlur(img, (kernelSize, kernelSize), 0)
    return blurredImg

# applying sobel filter
def sobelFilter(image, imgSize, n, kernel):
    filteredImg = np.zeros_like(image)

    for k in range(imgSize - n + 1):
        for p in range(imgSize - n + 1):
            subMatrix = np.zeros((n, n))
            subMatrix = image[k:k + n, p:p + n]
            multipliedValue = np.multiply(subMatrix, kernel)
            sumAllElements = np.sum(multipliedValue)

            filteredImg[k + 1, p + 1] = sumAllElements
    return filteredImg
```

```python
# finding gradient magnitude and phase
def gradientMagPhase(gradX, gradY):
    gradXSquare = np.square(gradX)
    gradYSquare = np.square(gradY)
    gradxAddGradY = np.add(gradXSquare, gradYSquare)

    gMagnitude = np.sqrt(gradxAddGradY)
    gPhase = np.arctan2(gradY, gradX) * 180 / np.pi

    return gMagnitude, gPhase

# running through the gradient magnitude values and applying nms
# values are calculated based on the gradient angle
def nonMaximumSuppression(mag, phase, imgSize, kernelSize):
    suppressedImg = np.copy(mag)
    maxGradValue = np.amax(mag)

    for k in range(imgSize - kernelSize + 1):
        for p in range(imgSize - kernelSize + 1):
            tempPhase = phase[k+1][p+1]

            if (-22.5 <= tempPhase < 22.5) or (157.5 <= tempPhase < 180) or (-180 <= tempPhase < -157.5):
                middle = mag[k + 1][p + 1]
                topCenter = mag[k][p + 1]
                bottomCenter = mag[k + 2][p + 1]
                if middle < topCenter or middle < bottomCenter:
                    suppressedImg[k + 1][p + 1] = 0

            elif (22.5 <= tempPhase < 67.5) or (-157.5 <= tempPhase < -112.5):
                middle = mag[k + 1][p + 1]
                topLeft = mag[k][p]
                bottomRight = mag[k + 2][p + 2]
                if middle < topLeft or middle < bottomRight:
                    suppressedImg[k + 1][p + 1] = 0

            elif (67.5 <= tempPhase < 112.5) or (-112.5 <= tempPhase < -67.5):
                middle = mag[k + 1][p + 1]
                middleLeft = mag[k + 1][p]
                middleRight = mag[k + 1][p + 2]
                if middle < middleLeft or middle < middleRight:
                    suppressedImg[k + 1][p + 1] = 0
```

```
            elif (112.5 <= tempPhase < 157.5) or (-67.5 <= tempPhase < -22.5):
                middle = mag[k + 1][p + 1]
                topRight = mag[k][p + 2]
                bottomLeft = mag[k + 2][p]
                if middle < topRight or middle < bottomLeft:
                    suppressedImg[k + 1][p + 1] = 0

            else:
                raise Exception("Sorry, could not classify")

    # binding the values to range of 0 to 255
    suppressedImg = (suppressedImg / maxGradValue) * 255
    return np.round(suppressedImg)

# applying threshold given in the question to classify strong and weak edges
def threshold(image, lowThresh, highThresh):

    threshResult = np.zeros_like(image, dtype=np.int32)

    weak = np.uint8(lowThresh)
    strong = np.uint8(255)

    strong_i, strong_j = np.where(image >= highThresh)
    zeros_i, zeros_j = np.where(image < lowThresh)
    weak_i, weak_j = np.where((image <= highThresh) & (image >= lowThresh))

    threshResult[strong_i, strong_j] = strong
    threshResult[weak_i, weak_j] = weak
    threshResult[zeros_i, zeros_j] = 0

    return threshResult

# applying hysteresis to connect strong and weak edges
def hysteresis(image, weak):
    maxValue = 255
    edge = 1
    hysteresisRes = np.zeros_like(image)

    for i in range(imgShape-kernelShape+1):
        for j in range(imgShape-kernelShape+1):
            if image[i, j] == weak:
                if ((image[i + 1, j - 1] == maxValue) or (image[i + 1, j] == maxValue) or
```

```
124     for i in range(imgShape-kernelShape+1):
125         for j in range(imgShape-kernelShape+1):
126             if image[i, j] == weak:
127                 if ((image[i + 1, j - 1] == maxValue) or (image[i + 1, j] == maxValue) or
128                         (image[i + 1, j + 1] == maxValue) or (image[i, j - 1] == maxValue) or
129                         (image[i, j + 1] == maxValue) or (image[i - 1, j - 1] == maxValue) or
130                         (image[i - 1, j] == maxValue) or (image[i - 1, j + 1] == maxValue)):
131                     image[i, j] = maxValue
132                 else:
133                     image[i, j] = 0
134 
135     hysteresisRes = np.where(image == 255, 1, 0)
136     return hysteresisRes
137 
138 
139 # computing the gaussian blur on the image to smoothen it
140 blurredImg = gaussianFilter(img, kernelShape)
141 
142 # Computing the gradient in x and y directions
143 Gx = sobelFilter(blurredImg, imgShape, kernelShape, kernelSobelX)
144 Gy = sobelFilter(blurredImg, imgShape, kernelShape, kernelSobelY)
145 
146 # Computing the gradient magnitude and phase in degree
147 gradMag, gradPhase = gradientMagPhase(Gx, Gy)
148 # print(gradMag)
149 # print(gradPhase)
150 
151 nmsImg = nonMaximumSuppression(gradMag, gradPhase, imgShape, kernelShape)
152 print(nmsImg)
153 thresholdResult = threshold(nmsImg, lowThreshold, highThreshold)
154 # print(thresholdResult)
155 hysteresisResult = hysteresis(thresholdResult, lowThreshold)
156 # print(hysteresisResult)
157 
158 
159 
160 # output using Canny Edge Detection of OpenCV for comparison with my method
161 img2 = np.zeros_like(img)
162 edgeDetected = cv2.Canny(img.astype('uint8'), lowThreshold, highThreshold, img2, kernelShape, True)
```

Script Output: Note that the gradient values have been rounded (see code)

$$\text{NMS with Gradient} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 568 & 539 & 0 & 0 & 0 \\ 0 & 539 & 0 & 0 & 0 & 411 & 0 & 0 \\ 0 & 0 & 477 & 477 & 0 & 477 & 477 & 0 \\ 0 & 0 & 0 & 411 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 539 & 568 & 539 & 0 \\ 0 & 0 & 0 & 0 & 0 & 402 & 402 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

After finding NMS using magnitude of gradients, values have been bounded to a range of 0 to 255. Check line 97 and 98 in the code section.

$$\text{NMS with Pixel} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 255 & 242 & 0 & 0 & 0 \\ 0 & 244 & 0 & 0 & 0 & 185 & 0 & 0 \\ 0 & 0 & 214 & 214 & 0 & 214 & 214 & 0 \\ 0 & 0 & 0 & 185 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 242 & 255 & 242 & 0 \\ 0 & 0 & 0 & 0 & 0 & 180 & 180 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Method 2: Apply gaussian before computing G_x and G_y values. Code is attached in the report, results shown below

NMS with Gradient =

```
[[  0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0. 141.   0. 333.   0.   0.   0.]
 [  0. 290. 267.   0.   0. 312.   0.   0.]
 [  0.   0. 346. 279.   0. 278. 325.   0.]
 [  0.   0.   0. 312.   0.   0.   0.   0.]
 [  0.   0.   0.   0. 341.   0. 250.   0.]
 [  0.   0.   0.   0.   0. 277.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.]]
```

NMS with Pixel =

```
[[  0.   0.   0.   0.   0.   0.   0.   0.]
 [  0.   0. 104.   0. 245.   0.   0.   0.]
 [  0. 213. 197.   0.   0. 230.   0.   0.]
 [  0.   0. 255. 206.   0. 205. 240.   0.]
 [  0.   0.   0. 230.   0.   0.   0.   0.]
 [  0.   0.   0.   0. 251.   0. 184.   0.]
 [  0.   0.   0.   0.   0. 204.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0.]]
```

As you can notice, we get slightly better result with gaussian filtering pre-processing step.

c. Hysteresis Thresholding

Steps involved in Hysteresis thresholding:

1. Classify edges as weak and strong based on the low and high threshold values. Any value greater than or equal to high threshold are strong edges, any values below low threshold are discarded and weak edges have the range of values between low and high threshold.
2. As the first step of edge linkage, join adjacent strong edges.
3. For each weak edge, we need to perform the following steps: let's start with a weak pixel and call it 'a', look at the neighbor values to locate strong or weak pixel.
4. If strong or weak pixel is found around the weak pixel 'a' - store it in a list and let's call the new pixel 'b'. We need to store the weak edge a's location, because we need its path to join it with other weak or strong edges, if the path exists.
5. Then repeat the searching process on pixel 'b', continue the process until it encounters a value of all zeros or a strong edge. If neighboring values are all zeros, then drop the weak edge because it means that this weak edge does not connect with a strong edge. On the other hand, if the process leads to discovery of a strong edges, then entitle this weak edge as an edge because it has a connection with strong edges on both ends.

Now that we have non maximum suppression matrix, we can apply the low (L=20) and high (H=120) threshold given in the question. In the following matrix, I have replaced non-zero values with the a value of 1 to apply thresholding step. 1 represents an edge below:

Edges Detected image =
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

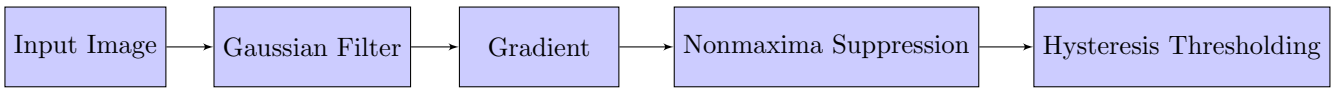
However, if we apply thresholding technique on method 2 (pre-processing with Gaussian filter), we will get slightly better result, as shown below.

Edges Detected filtered image =
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

For completeness sake, here is the result from the script:

Edges Detected =
```
[[0 0 0 0 0 0 0 0]
 [0 0 0 1 1 0 0 0]
 [0 1 0 0 0 1 0 0]
 [0 0 1 1 0 1 1 0]
 [0 0 0 1 0 0 0 0]
 [0 0 0 0 1 1 1 0]
 [0 0 0 0 0 1 1 0]
 [0 0 0 0 0 0 0 0]]
```

d. Block Diagram



- First step is to smooth the input image by applying Gaussian filter
- Next, we need to compute gradient magnitude and angle for every pixel in the image
- To limit the edge thickness, we need to apply nonmaxima suppression technique to the gradient magnitudes
- Finally, we need to define two threshold values - low and high, to consider strong edges and weak edges only if they are linked to strong edges.

e.

No, Canny operator can not give an edge with width more than 1. This is because Canny edge detector is applying nonmaxima suppression technique which gets rid of lower valued adjacent pixels, if present, along the width of the edge. Local maximas obtained around an edge during gradient computation are eradicated by higher valued neighboring pixels. Therefore, for an edge, Canny operator should detect a single point edge. Another technique that helps Canny operator to get rid of noise is edge linkage, which is applied based on hysteresis.

3. Convolution Theorem

For the 2D discrete, convolution theorem is:

$$G[i,j] = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i-u, j-v]$$

$$G = h \star f(x,y) = (H.F)(u,v)$$

Here, $\star$ denotes the notation for convolution, and F and H are Fourier transforms of f and h. Basically, it states that Fourier transform of the spatial convolution of h and f is the product of their fourier (F and H). Applying inverse fourier transform of the product: F.H will result in convolution of f and h (f$\star$h).
Also, note that F (filter) is flipped in both i- and j- directions and then, cross-correlation is applied. Above stated equations show the product of two functions in the frequency domain which, by the convolution theorem, implies convolution in the spatial domain.
Explanation: In spatial domain, convolution step involves element wise multiplication of the kernel with the sub-matrix extracted from image (sub matrix's size is identical to kernel size), and after the element wise multiplication, all the values are summed together. Convolution is applied on each pixel in the image and for the corner pixels, padding is required. Another way to think about convolution is by turning kernel and image sub-matrix (image patch) into vectors and performing dot product between the two vectors which involves element-wise multiplication and summing the results. Note that convolution and cross-correlation are closely related, except in cross-correlation kernel is not flipped in either direction. Instead of working in the spatial domain, we can convert the two functions to frequency domain which will allow us to carry out the multiplication of two functions. Finally, the result can be brought back to spatial domain by taking inverse fourier transform of the resulted product.
Convolution is useful in template matching, because template is moved over the entire image and at each pixel, convolution depicts the similarity between the template and the image sub-matrix. Care should be taken when working with image with varying intensity levels, because higher intensity/bright in an area will generate large convolution result even though the template is not matching the image patch. This is why normalization step is added to produce numbers between -1 and 1, and this is called normalized correlation. If the image region is identical to the template/kernel, then the result will be 1. This normalized correlation step is applied to the entire image to filter out template in the given image.

4. Sharpening an Image

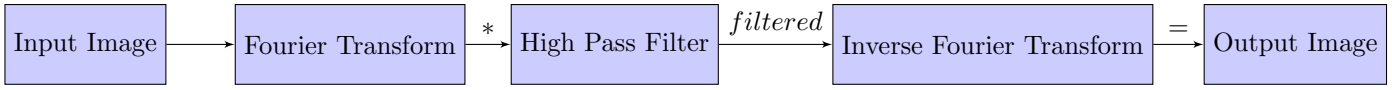
In both frequency and spatial domain, we can subtract a low pass filter transfer function from 1 (or another constant) which outputs the corresponding highpass filter transfer function. In short,

$$TF_{HPF} = 1 - TF_{LPF}$$

where TF denotes transfer function, HPF for high pass filter and LPF for low pass filter.
Another simple way of getting a high pass filtered image is to first run a low pass filter on the image and then, subtract the input image from the filtered image. This will leave high frequency components in the image.

Proposal to make an image sharper :

In frequency domain:



From the block diagram, we can see that image can be converted to get frequency domain representation using Fourier transform. Since image and the filter are both in frequency domain, the filter can applied to the image with multiplication step instead of convolution. After getting the filtered image, inverse Fourier transform can be applied to get the filtered image back to spatial domain.

In spatial domain: Note image blurring can be achieved by using the averaging (smoothing) filter and this is analogous to integration. In the same way, image sharpening is analogous to differentiation because high pass filters are comprised of differences between center pixel and the neighboring pixels. Based on this knowledge, we can design high pass filters/kernel and convolude it with the image to sharpen it. Thus, this filter has to be applied at each pixel to remove non-varying (or slow varying) intensities and obtain edges.

Implementation

1. DoG

Python Script:

```python
import numpy as np
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('lp.jpg')
# attempt to try it in gray scale

# setting kernel/sigma values
kernelSizeGaussian_1 = 3       # 17
kernelSizeGaussian_2 = 1       # 15
# letting function compute sigma values
sigmaX_1 = 0
sigmaX_2 = 0

# calling gaussian filter twice to set up DoG
gaussianFiltered_1 = cv2.GaussianBlur(img, (kernelSizeGaussian_1, kernelSizeGaussian_1), sigmaX_1)
gaussianFiltered_2 = cv2.GaussianBlur(img, (kernelSizeGaussian_2, kernelSizeGaussian_2), sigmaX_2)
# cv2.imshow("low standard deviation", gaussianFiltered_1)
# cv2.imshow("large standard deviation", gaussianFiltered_2)
# cv2.waitKey(0)

# DoG function: subtracting one from another
DoGImg = (cv2.subtract(gaussianFiltered_1, gaussianFiltered_2))
cv2.imshow("DoG", DoGImg)

if cv2.waitKey(0) == ord('s'):
    cv2.imwrite('Impl_1.jpg', DoGImg)

cv2.destroyAllWindows()
```

Script shown above reads the image, and sets the kernel size (after trying different values to get the best result). We want to choose kernel size close to one another because this will limit the frequency range being passed through the DoG filter (as explained below). Sigma values are zero to allow the GaussianBlur() to compute them automatically. Idea is to define two Gaussian filters such that standard deviation of one is greater than another, i.e. $\sigma_1 > \sigma_2$. Higher standard deviation filter will blur the high frequency components more as compared to the other filter. After subtracting the two filtered images, we can retrieve the frequencies that lie between the two standard deviations and which have not been suppressed.

Script Output:

Following output shows DoG of colored image, note that standard deviation is controlled by the kernel size. In this output, we can see that most of the colored information is removed except the edges. These edges represent the frequencies that lie between the two standard deviations of the GaussianBlur(). Perhaps this image can be improved by applying post processing step on this filtered image.

Kernel Sizes = 17 and 15:

We can further improve the outputs by applying a post-processing step of binarization to the image to sharpen the edges.

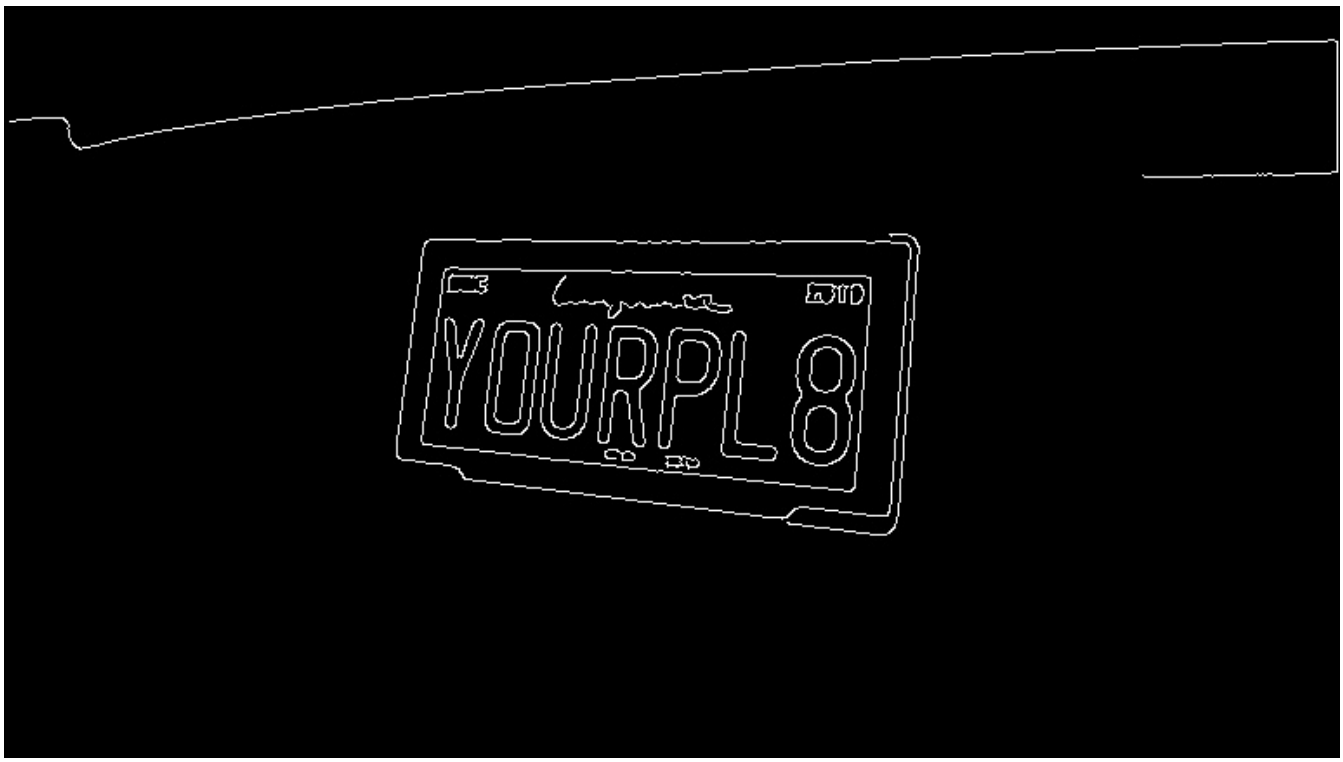
1.1

Python Script:

```
import numpy as np
import cv2

img = cv2.imread('lp.jpg')

# preprocessing image using gaussian filter
kernelSizeGaussian = 9
gaussianFiltered = cv2.GaussianBlur(img, (kernelSizeGaussian, kernelSizeGaussian),
                                    0, borderType=cv2.BORDER_CONSTANT)

# based on sigma value and median of the image, low and high threshold will be
# calculated. The idea is to pick values that are close to the median.
sigma = 3
median = np.median(img)
lowThreshold = int(max(0, (1.0 - sigma) * median))
highThreshold = int(min(255, (1.0 + sigma) * median))
print(f'low threshold = {lowThreshold}, high threshold = {highThreshold}')

# calling Canny() function to get the license plate only
cannyOperatedImg = cv2.Canny(gaussianFiltered, lowThreshold, highThreshold,
                             None, L2gradient=True)

cv2.imshow("Canny Operated Image", cannyOperatedImg)
if cv2.waitKey(0) == ord('s'):
    cv2.imwrite('Impl_1_1.jpg', cannyOperatedImg)

cv2.destroyAllWindows()
```

In the script above, after reading the image, GaussianBlur() filter is applied to pre-process the image for Canny Operator. Advantage of pre-processing is that it will remove noise that will reduce the computation steps in the Canny function and it will make it easier for the function to find the edges.
Also, note that the low and high threshold are selected by finding the median value of the image. Based on this median value, low and high threshold values are determined by expanding the range about median using sigma value of 3. This will set up a range for us that will exist in the image the most. Note that in the Canny() L2gradient is set to True, this will ensure that it will not use the approximation ($|G_x| + |G_y|$), instead it will use the gradient magnitude equal to $\sqrt{G_x^2 + G_y^2}$.

Script Output:
As you can see below, most of the background has been removed and we can see the licence plate numbers clearly.


YOURPL8

2.

When we have information about the scale of our object, Gaussian filter is more useful because then we can control the kernel size. Based on this kernel size, standard deviation will be computed to get eliminate the frequencies from the image. Canny operator is more useful to remove random noise as it is a more sophisticated method and in most cases, the result from Canny algorithm is far better than Gaussian filters output. However, with Canny operator, we can not control the kernel size, we can only control the low and high threshold values which can be irrelevant to the object scale. DoG method can create thicker edges as it preserves frequency range in between the two standard deviations. With Canny operator, we get sharp edges of pixel width equal to 1.

3. Template Matching

Python Script:

```python
import numpy as np
import cv2
from matplotlib import pyplot as plt

# reading image and applying Canny Operator to remove noise
img = cv2.imread('messi.jpg', cv2.IMREAD_GRAYSCALE)
cannyImg = cv2.Canny(img, 50, 150, L2gradient=True)
img2 = img.copy()
img = cannyImg
width, height = img.shape[::-1]

# reading template and applying Canny operator to remove noise
template = cv2.imread('circle.bmp', cv2.IMREAD_GRAYSCALE)
template = cv2.Canny(template, 50, 150, L2gradient=True)
w, h = template.shape[::-1]

# computing the cross correlation terms, here computing for
# template window
meanTemplate = np.mean(template)
templMinusMean = np.subtract(template, meanTemplate)
templMinusMeanSquare = np.square(templMinusMean)
templMinusMeanSqSum = np.sum(templMinusMeanSquare)

matchedRes = np.zeros((height-h+1, width-w+1))

# convoluting with the image by passin the window at every pixel
# in the for loop, calculating cross correlation terms because
# it will change for the image each time
for k in range(height - h + 1):
    for p in range(width - w + 1):
        subMat = img[k:k+h, p:p+w]
        meanSubMat = np.mean(subMat)
        subMatMinusMean = np.subtract(subMat, meanSubMat)
        subMatMinusMeanSq = np.square(subMatMinusMean)
        subMatMinusMeanSqSum = np.sum(subMatMinusMeanSq)

        numMultiply = np.multiply(templMinusMean, subMatMinusMean)
        numerator = np.sum(numMultiply)
```

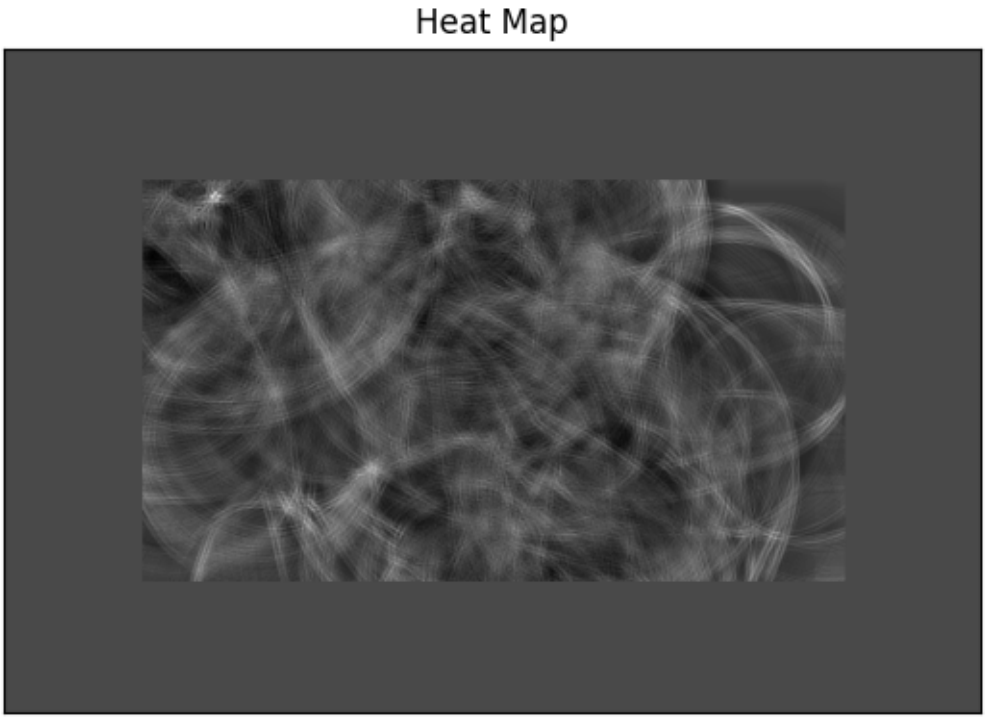
Unpadded Image:

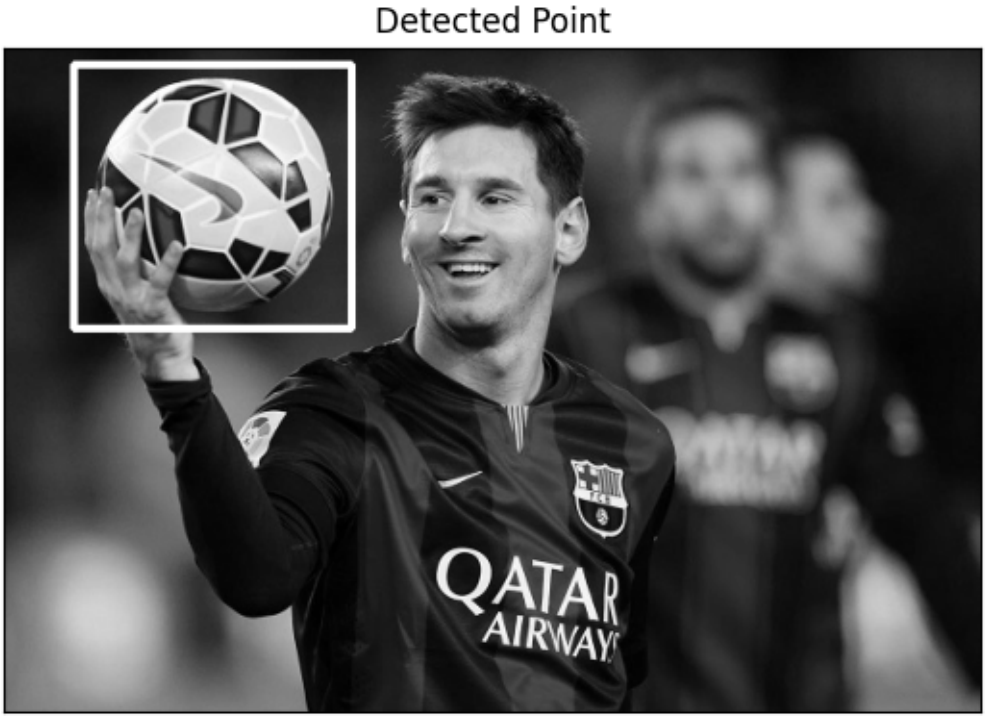
In the script above (part 1 of 2), image and template are read in gray scale and Canny Opertor is applied to work with the edges of the image. In the 'for' loop, *subMat* variable stores the sub-matrix that has the same shape as the template. After calculating the cross correlation term values, final value is stored in *matchedRes*.

In the script below (part 2 of 2) *Zero Divison* error is checked before storing the final value in the *matchedRes*. *matchedRes* is transferred into an array of image size to easily locate the circle at the right spot. Finally, max value and location are computed to place a rectangle on the image of same width and height as the template.

```
            denMultiply = templMinusMeanSqSum * subMatMinusMeanSqSum
            denominator = (denMultiply) ** (1/2)

            if denominator == 0:
                print(f'Zero Division at {k}, {p}')
                break
            else:
                matchedRes[k, p] = numerator/denominator


# saving the heat map in the image shape array
result = np.zeros((img.shape))
result[h//2:(h//2)+254, w//2:(w//2)+445] = matchedRes

# finding min and max values and location
# but will require max value and location in this case
min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)
print(f'max value: {max_val} at {max_loc}')

# setting corner values for the rectange that will be drawn on the
# image
top_left = max_loc[0] - (w//2), max_loc[1] - (h//2)
bottom_right = (top_left[0] + w, top_left[1] + h)
cv2.rectangle(img2, top_left, bottom_right, 255, 4)

# plotting the image with its heat map
plt.subplot(211), plt.imshow(result, cmap = 'gray')
plt.title('Heat Map'), plt.xticks([]), plt.yticks([])
plt.subplot(212), plt.imshow(img2, cmap = 'gray', vmin=0, vmax=255)
plt.title('Detected Point'), plt.xticks([]), plt.yticks([])
plt.show()
```

Script Output:





On the heat map, you should be able to see a white dot that represents the max value of cross correlation formula. Also, white rectangle is drawn at the same position using max location as center point. This white dot shows max similarity value between the template and the messi image. Cross correlation ranges between -1 and 1, value of 1 denotes exact match and value of -1 denotes max dissimilarity level between the template and the image.

Python Script:

```
1  import numpy as np
2  import cv2
3  from matplotlib import pyplot as plt
4
5  # reading template and applying Canny operator to remove noise
6  template = cv2.imread('circle.bmp', cv2.IMREAD_GRAYSCALE)
7  template = cv2.Canny(template, 50, 150, L2gradient=True)
8  w, h = template.shape[::-1]
9  # computing the cross correlation terms, here computing for
10 # template window
11 meanTemplate = np.mean(template)
12 templMinusMean = np.subtract(template, meanTemplate)
13 templMinusMeanSquare = np.square(templMinusMean)
14 templMinusMeanSqSum = np.sum(templMinusMeanSquare)
15
16 # reading image and applying Canny Operator to remove noise
17 # creating padding around the image to go through all the pixels
18 # in the image
19 img = cv2.imread('messi.jpg', cv2.IMREAD_GRAYSCALE)
20 img = cv2.copyMakeBorder(img, w//2, w//2, h//2, h//2, cv2.BORDER_REFLECT)
21 cannyImg = cv2.Canny(img, 47, 75, L2gradient=True)
22 # cv2.imshow("Canny Image", cannyImg)
23 # cv2.waitKey(0)
24 img2 = img.copy()
25 img = cannyImg
26 width, height = img.shape[::-1]
27
28 matchedRes = np.zeros((height-h+1, width-w+1))
29
30 # convoluting with the image by passin the window at every pixel
31 # in the for loop, calculating cross correlation terms because
32 # it will change for the image each time
33 for k in range(height - h + 1):
34     for p in range(width - w + 1):
35         subMat = img[k:k+h, p:p+w]
36         meanSubMat = np.mean(subMat)
37         subMatMinusMean = np.subtract(subMat, meanSubMat)
38         subMatMinusMeanSq = np.square(subMatMinusMean)
39         subMatMinusMeanSqSum = np.sum(subMatMinusMeanSq)
```

Padded Image:

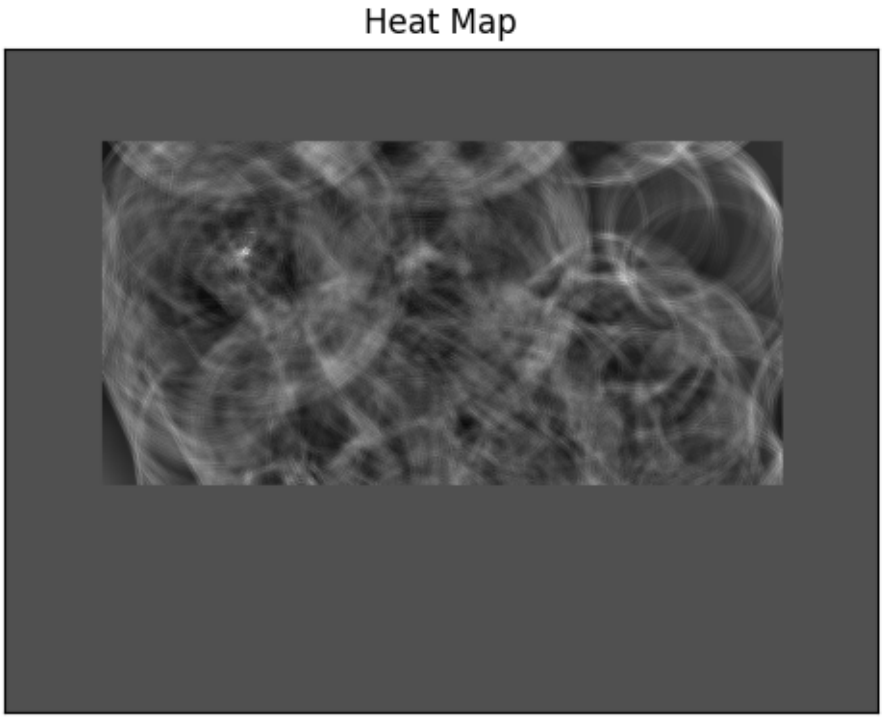
Note that on line 20, image has been padded using *copyMakeBorder* and *cv2.BORDER_REFLECT* is used to mirror the image on all sides.

In the script above (part 1 of 2), image and template are read in gray scale and Canny Opertor is applied to work with the edges of the image. In the 'for' loop, *subMat* variable stores the sub-matrix that has the same shape as the template. After calculating the cross correlation term values, final value is stored in *matchedRes*.

In the script below (part 2 of 2) *Zero Divison* error is checked before storing the final value in the *matchedRes*. *matchedRes* is transferred into an array of padded image size to easily locate the circle at the right spot. Finally, max value and location are computed to place a rectangle on the image of same width and height as the template.

```
            subMatMinusMeanSq = np.square(subMatMinusMean)
            subMatMinusMeanSqSum = np.sum(subMatMinusMeanSq)

            numMultiply = np.multiply(templMinusMean, subMatMinusMean)
            numerator = np.sum(numMultiply)

            denMultiply = templMinusMeanSqSum * subMatMinusMeanSqSum
            denominator = (denMultiply) ** (1/2)

            if denominator == 0:
                print(f'Zero Division at {k}, {p}')
                break
            else:
                matchedRes[k, p] = numerator/denominator

    # saving the heat map in the image shape array
    result = np.zeros((img.shape))
    result[h//2:(h//2)+height-h+1, w//2:(w//2)+width-w+1] = matchedRes

    # finding min and max values and location
    # but will require max value and location in this case
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)
    print(f'max value: {max_val} at {max_loc}')

    # setting corner values for the rectange that will be drawn on the
    # image
    top_left = max_loc[0] - (w//2), max_loc[1] - (h//2)
    bottom_right = (top_left[0] + w, top_left[1] + h)
    cv2.rectangle(img2, top_left, bottom_right, 255, 4)

    # plotting the image with its heat map
    plt.subplot(211), plt.imshow(result, cmap = 'gray')
    plt.title('Heat Map'), plt.xticks([]), plt.yticks([])
    plt.subplot(212), plt.imshow(img2, cmap = 'gray', vmin=0, vmax=255)
    plt.title('Detected Point'), plt.xticks([]), plt.yticks([])
    plt.show()
```

Script Output:



On the heat map, you should be able to see a white dot that represents the max value of cross correlation formula. Also, white rectangle is drawn at the same position using max location as center point.

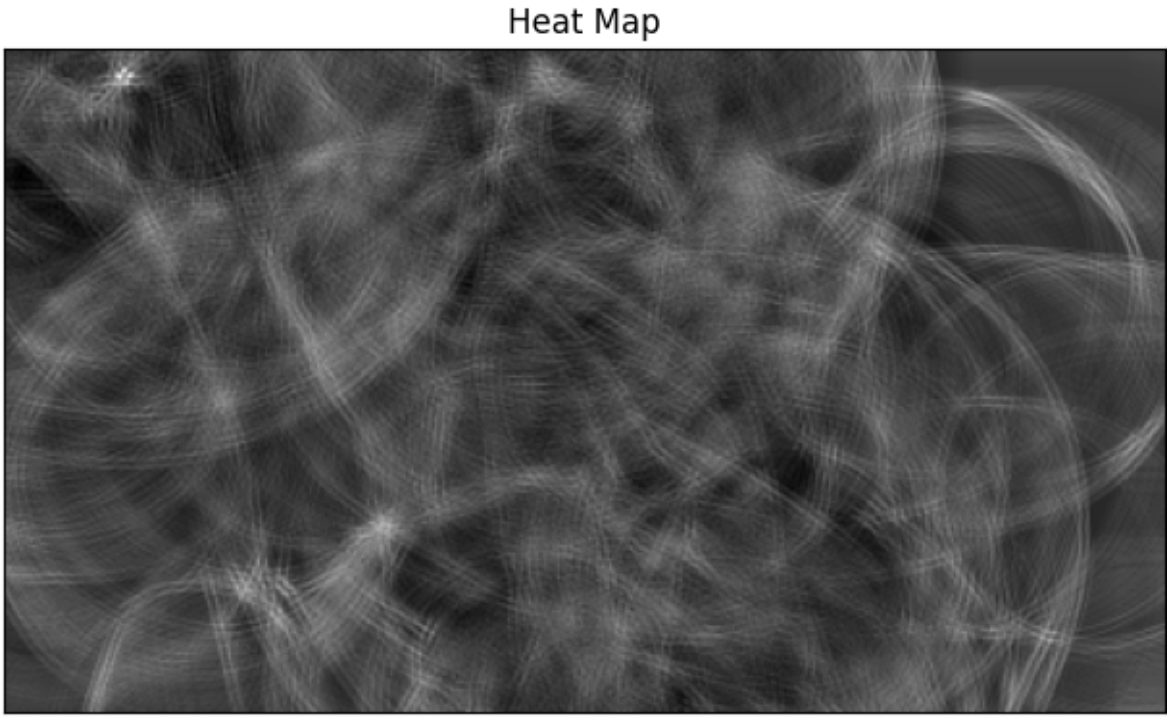
3.1

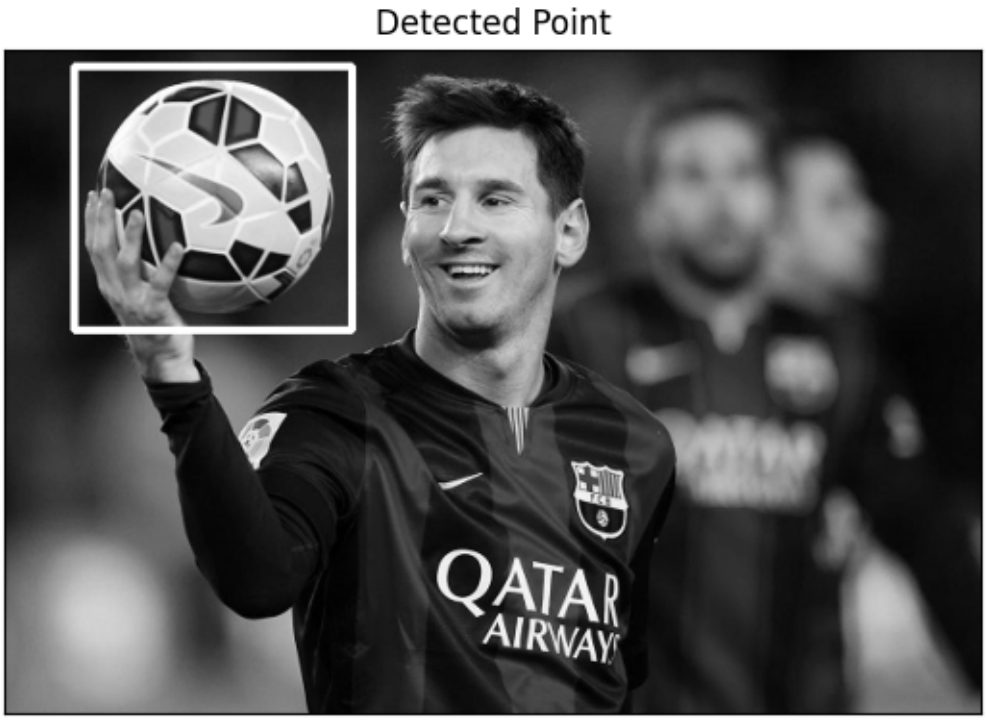
Python Script:

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

# reading the image and creating a copy to place a
# rectangle on the circle
img = cv2.imread('messi.jpg', cv2.IMREAD_GRAYSCALE)
cannyImg = cv2.Canny(img, 50, 150, L2gradient=True)
img2 = img.copy()
img = cannyImg

template = cv2.imread('circle.bmp', cv2.IMREAD_GRAYSCALE)
template = cv2.Canny(template, 50, 150, L2gradient=True)
w, h = template.shape[::-1]

# using matchTemplate() to template match
matchedRes = cv2.matchTemplate(img, template, cv2.TM_CCOEFF_NORMED)
min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(matchedRes)
print(max_val, max_loc)

# these two variables represent the corners of the rectangle
top_left = max_loc
bottom_right = (top_left[0] + w, top_left[1] + h)
cv2.rectangle(img2, top_left, bottom_right, 255, 4)

# ploting the heat map with image
plt.subplot(211), plt.imshow(matchedRes, cmap = 'gray')
plt.title('Heat Map'), plt.xticks([]), plt.yticks([])
plt.subplot(212), plt.imshow(img2, cmap = 'gray', vmin=0, vmax=255)
plt.title('Detected Point'), plt.xticks([]), plt.yticks([])
plt.show()
```

In the script above, image and template have been read as gray scale to apply the builtin *matchTemplate()*. After computing the max value and location, rectangle is drawn on the image to locate it.

Script Output:





On the heat map, you should be able to see a white dot that represents the max value of cross correlation formula. Also, white rectangle is drawn at the same position using max location as center point.

3.2

Proposal to find circles at different scales (Resize Image):

Follow the steps listed below to find the desired template.

1. Read the image to find desired template,
2. Apply Template match filter *matchTemplate()* at this scale
3. Store result: max value and its location (in this case, max is needed but in some cases min may be required)
4. Reduce the size of the image by a fixed amount, for instance resize the image to 90% of its initial size
5. Go back to step 2. Break out of this pattern, when you have finished template matching at all the desired scaling levels, for example going from 100% down to 10%.
6. Compare the results from all the scaling factor: max values at all percentage levels.
7. Using max value from previous step, draw a rectangle on the circle.

Note: Instead of reducing the size of the image to match the template, we can also increase the template size to test it against the given image. This will be shown after presenting the results for above-mentioned steps.

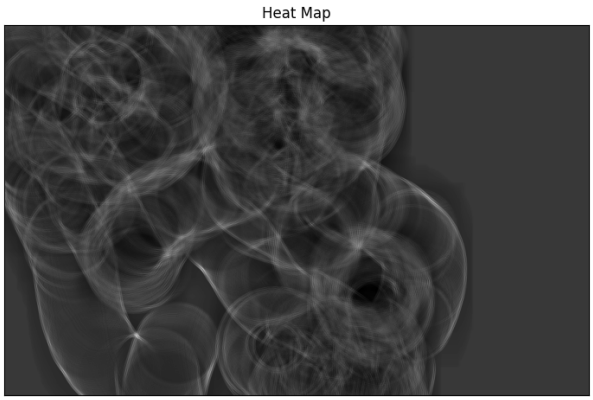
Python Script:

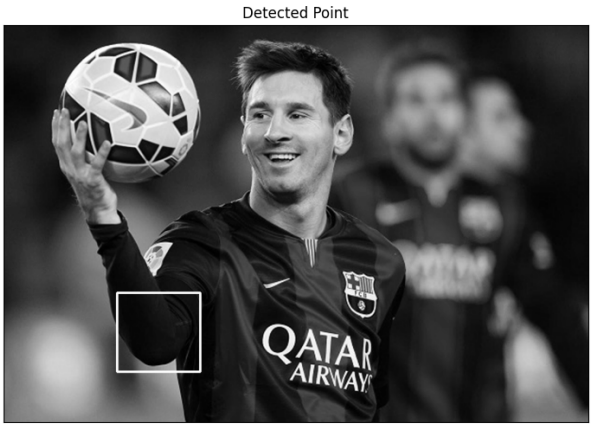
```python
import numpy as np
import cv2
from matplotlib import pyplot as plt

# reading the template as gray scale
template = cv2.imread('circle.bmp', cv2.IMREAD_GRAYSCALE)
template = cv2.Canny(template, 40, 90, L2gradient=True)
w, h = template.shape[::-1]

# reading the given image and resize it to a factor of 2
# this resized image will be used to find the circle
img = cv2.imread('messi.jpg', cv2.IMREAD_GRAYSCALE)
resizedImg = cv2.resize(img, None, fx=2, fy=2, interpolation=cv2.INTER_CUBIC)

img2 = img.copy()
img = resizedImg
width, height = img.shape[::-1]

# 0.1 scaling level is used, e.g. going from 100% to 90% to 80% and so on...
percentSpacing = 0.1
scalingFactor = np.arange(1, 0, -percentSpacing)

# looping over all the scaling factors to match the template
# result is shown as we apply the matchTemplate()
for index, scale in enumerate(scalingFactor):
    resizeImg = cv2.resize(resizedImg, None, fx=scale, fy=scale, interpolation=cv2.INTER_AREA)
    cannyImg = cv2.Canny(resizeImg, 40, 140, L2gradient=True)
    # cv2.imshow("resized image", cannyImg)
    # cv2.waitKey(0)

    matchedRes = cv2.matchTemplate(cannyImg, template, cv2.TM_CCOEFF_NORMED)
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(matchedRes)
    print(max_val, max_loc)

    top_left = max_loc
    bottom_right = (top_left[0] + w, top_left[1] + h)
    cv2.rectangle(resizeImg, top_left, bottom_right, 255, 4)
```

```
    plt.subplot(121), plt.imshow(matchedRes, cmap='gray')
    plt.title('Heat Map'), plt.xticks([]), plt.yticks([])
    plt.subplot(122), plt.imshow(resizeImg, cmap='gray', vmin=0, vmax=255)
    plt.title('Detected Point'), plt.xticks([]), plt.yticks([])
    plt.suptitle(f'Scaled down to {scale:.2f}, iteration = {index+1}')
    plt.show()
```

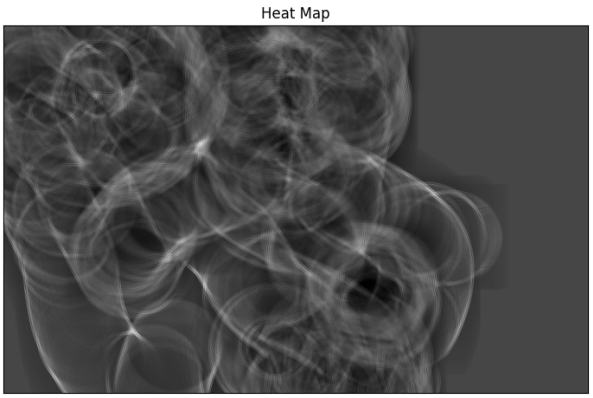
Script Output: Remember, image is initially at twice the size and image size is reducing

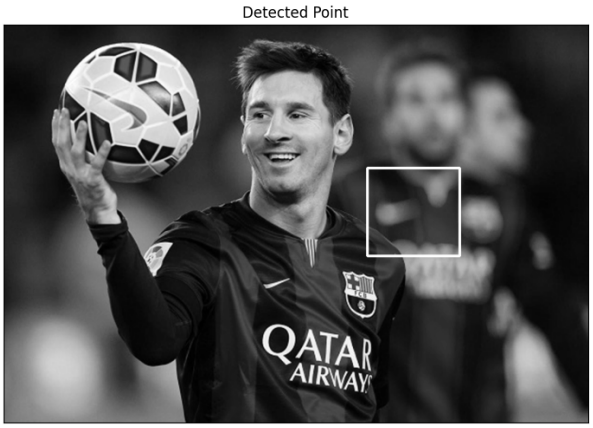
Iteration 1, Scale = 100%



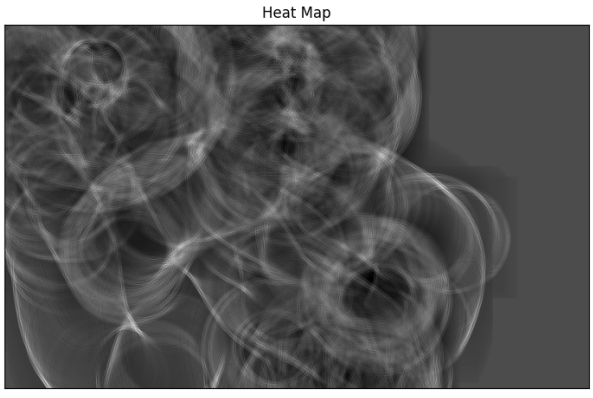


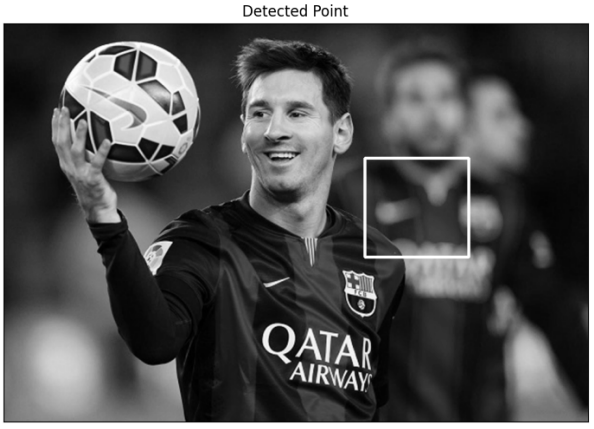
Iteration 2, Scale = 90%



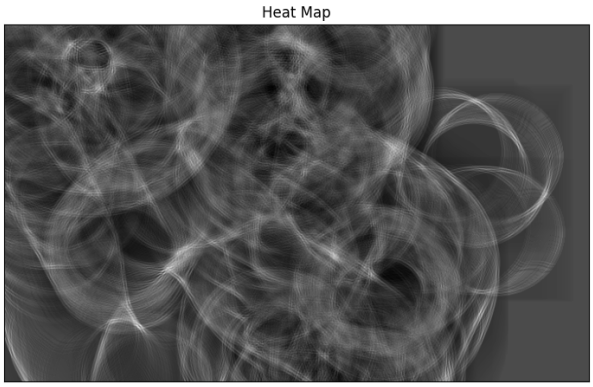


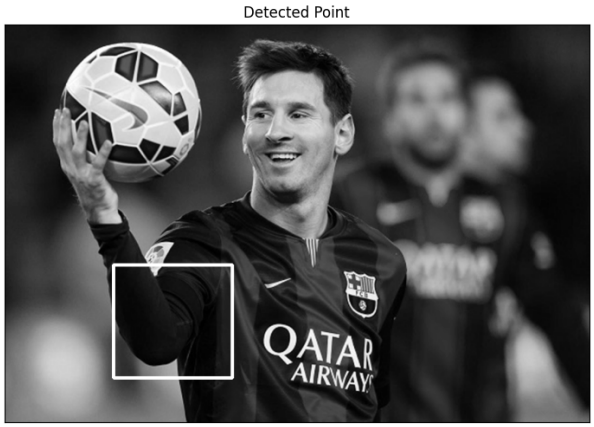
Iteration 3, Scale = 80%



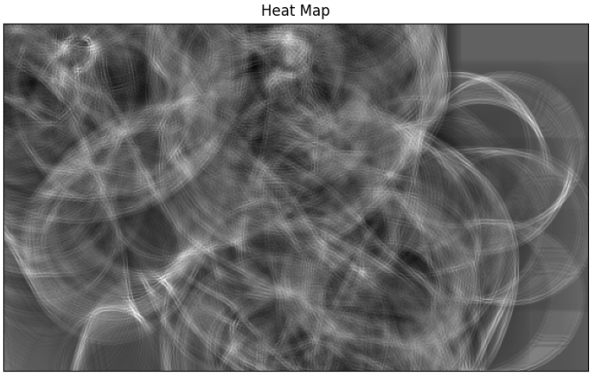


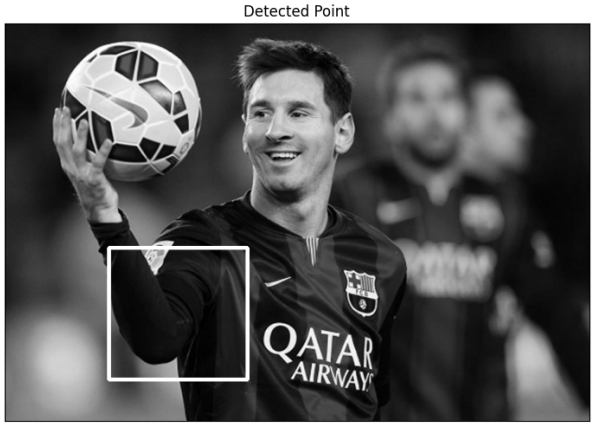
Iteration 4, Scale = 70%



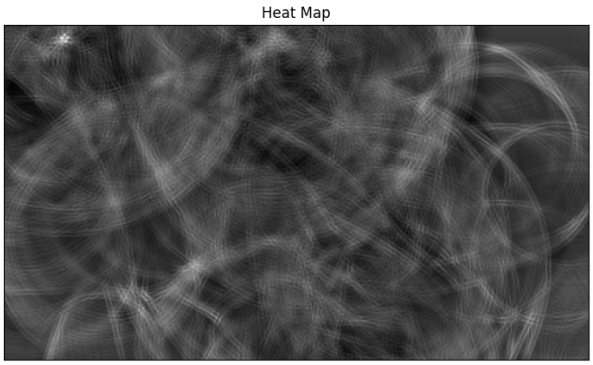


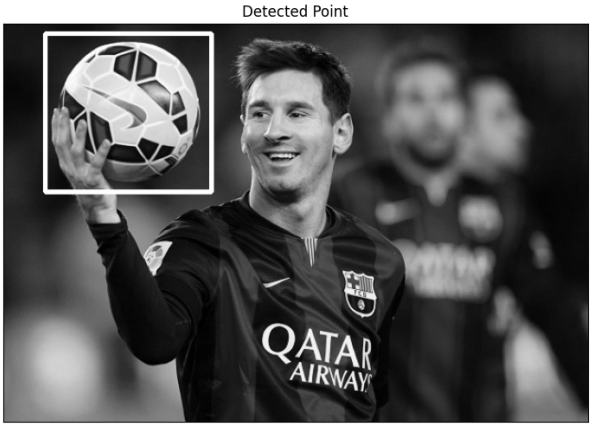
Iteration 5, Scale = 60%



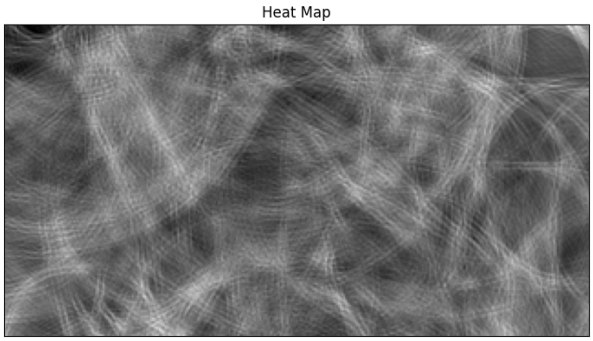


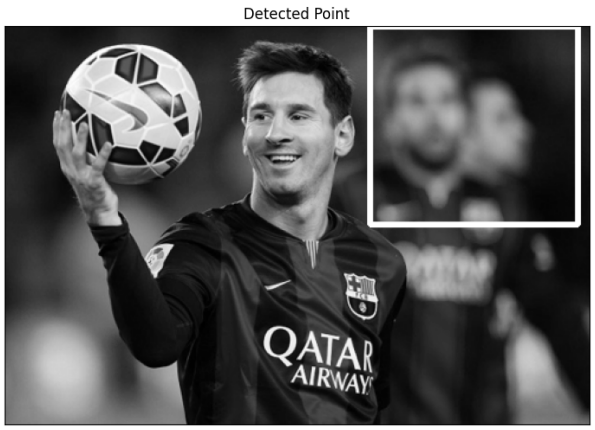
Iteration 6, Scale = 50%: As can be seen below, at this scale factor, *matchTemplate()* was able to find the circle because originally the image was at this size.



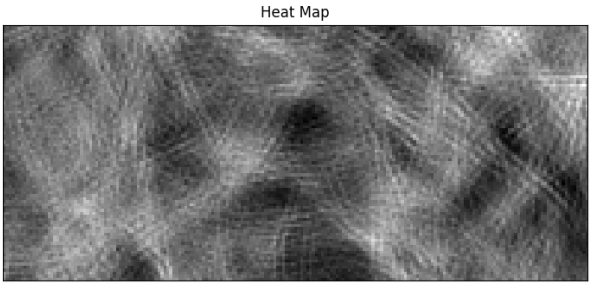


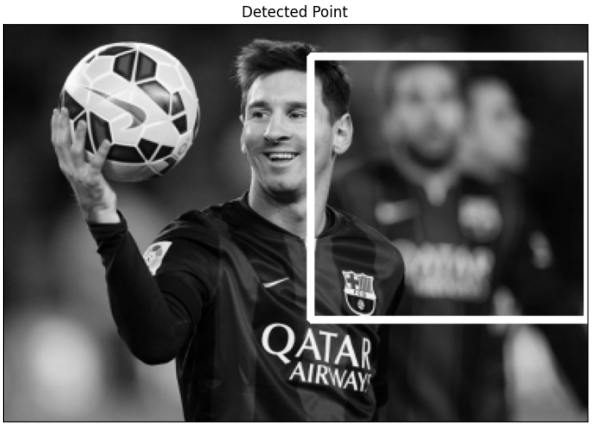
Iteration 7, Scale = 40%



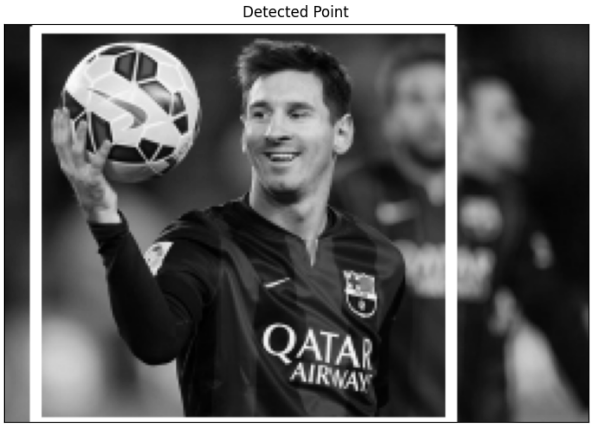


Iteration 8, Scale = 30%

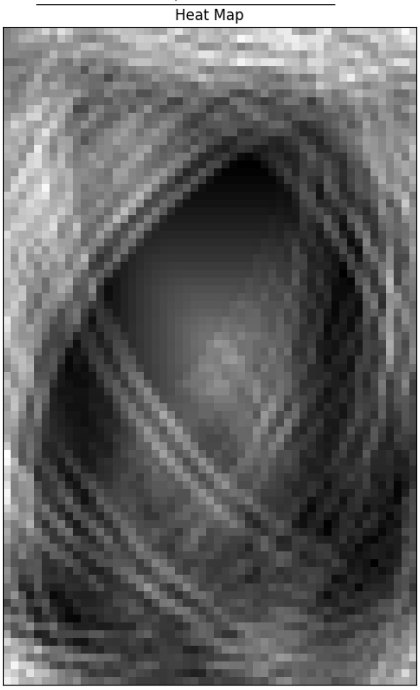


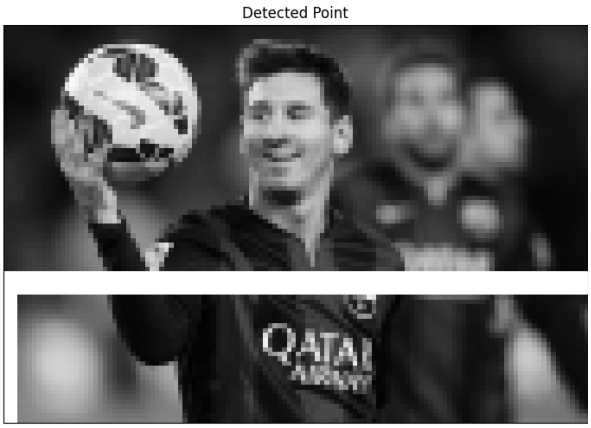


Iteration 9, Scale = 20%



Iteration 10, Scale = 10%





As explained earlier, we can modify the template size (instead of image size), it is shown below and we will see that we obtain similar results.

Proposal to find circles at different scales (Resize Template):

Follow the steps listed below to find the desired template.

1. Read the image to find desired template,
2. Apply Template match filter *matchTemplate()* at this scale
3. Store result: max value and its location (in this case, max is needed but in some cases min may be required)
4. Increase the size of the template by a fixed amount, for instance resize the template to 110% of its initial size
5. Go back to step 2. Break out of this pattern, when you have finished template matching at all the desired scaling levels, for example going from 100% up to 200%.
6. Compare the results from all the scaling factor: max values at all percentage levels.
7. Using max value from previous step, draw a rectangle on the circle.

Python Script:

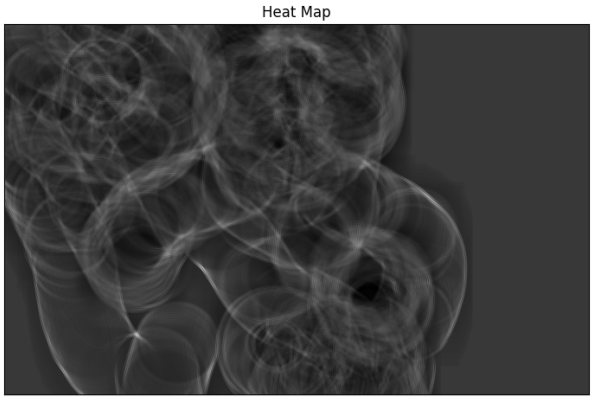
```python
import numpy as np
import cv2
from matplotlib import pyplot as plt

# read the template as gray scale
template = cv2.imread('circle.bmp', cv2.IMREAD_GRAYSCALE)

# read the image as gray scale and scale it up because we will be using
# the scaled up version (double size)
img = cv2.imread('messi.jpg', cv2.IMREAD_GRAYSCALE)
resizedImg = cv2.resize(img, None, fx=2, fy=2, interpolation=cv2.INTER_CUBIC)
cannyImg = cv2.Canny(resizedImg, 40, 140, L2gradient=True)

img2 = img.copy()
img = resizedImg
width, height = img.shape[::-1]

# increment the template size by 10% percent up to 200%
percentSpacing = 0.1
scalingFactor = np.arange(1, 2.1, percentSpacing)

# loop through all the scale levels to find the best match
for index, scale in enumerate(scalingFactor):
    resizeTemp = cv2.resize(template, None, fx=scale, fy=scale, interpolation=cv2.INTER_CUBIC)
    w, h = resizeTemp.shape[::-1]
    cannyTemp = cv2.Canny(resizeTemp, 40, 140, L2gradient=True)
    # cv2.imshow("resized image", cannyTemp)
    # cv2.waitKey(0)

    matchedRes = cv2.matchTemplate(cannyTemp, cannyImg, cv2.TM_CCOEFF_NORMED)
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(matchedRes)
    print(max_val, max_loc)

    img2 = resizedImg.copy()
    top_left = max_loc
    bottom_right = (top_left[0] + w, top_left[1] + h)
    cv2.rectangle(img2, top_left, bottom_right, 255, 4)
```

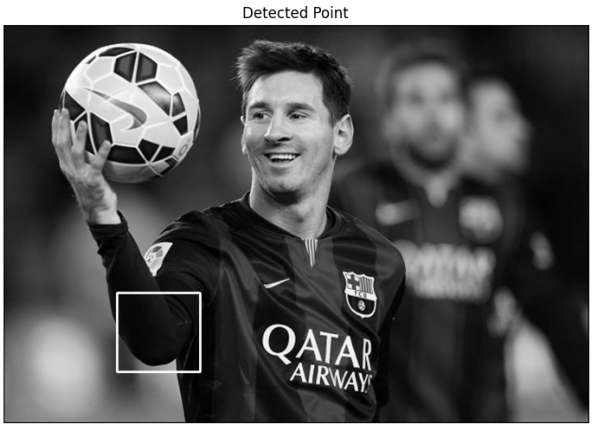
```python
    plt.subplot(121), plt.imshow(matchedRes, cmap='gray')
    plt.title('Heat Map'), plt.xticks([]), plt.yticks([])
    plt.subplot(122), plt.imshow(img2, cmap='gray', vmin=0, vmax=255)
    plt.title('Detected Point'), plt.xticks([]), plt.yticks([])
    plt.suptitle(f'Template scaled to {scale:.2f}, iteration = {index+1}')
    plt.show()
```

In the above script, template is being resized from 100% to 200% and as required image is resized to 200% initially to test the solution.

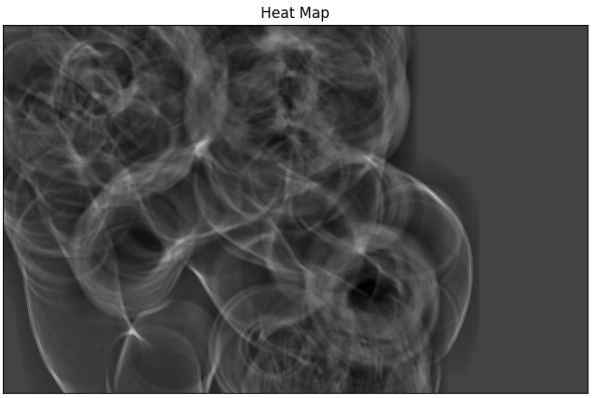
Script Output: Remember, template size is increasing in the following screenshots

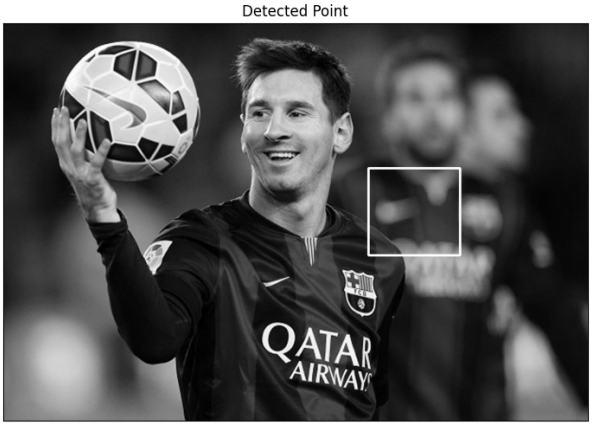
Iteration 1, Template Scale = 100%



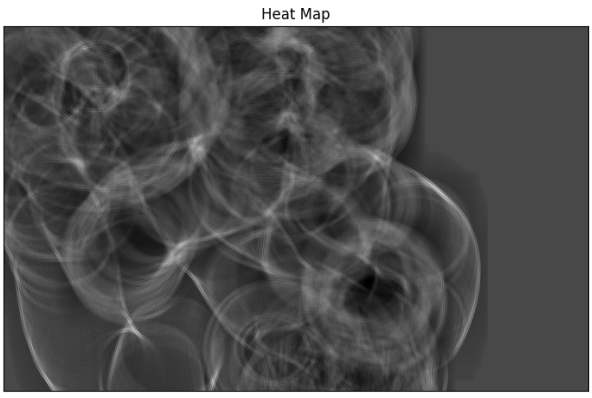


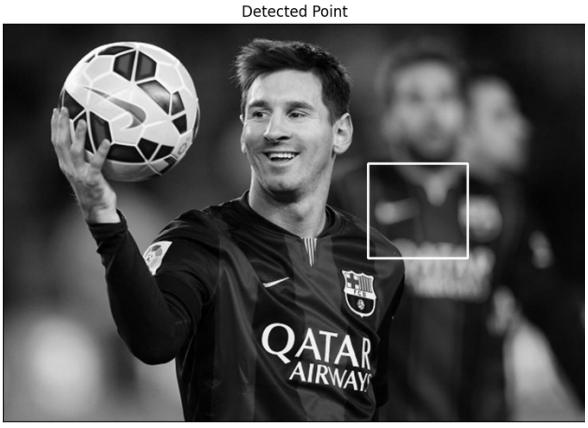
Iteration 2, Template Scale = 110%



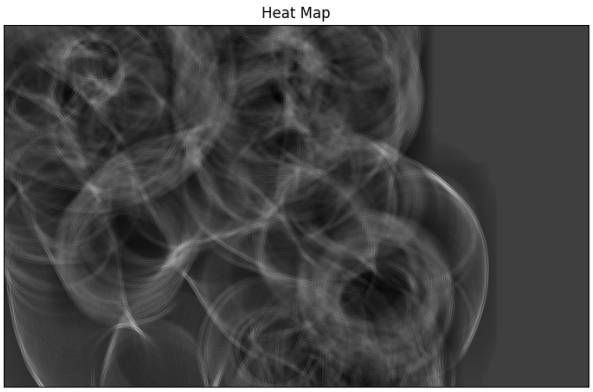


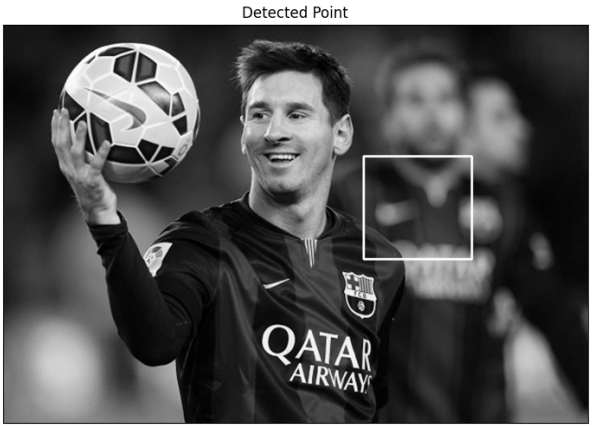
Iteration 3, Template Scale = 120%



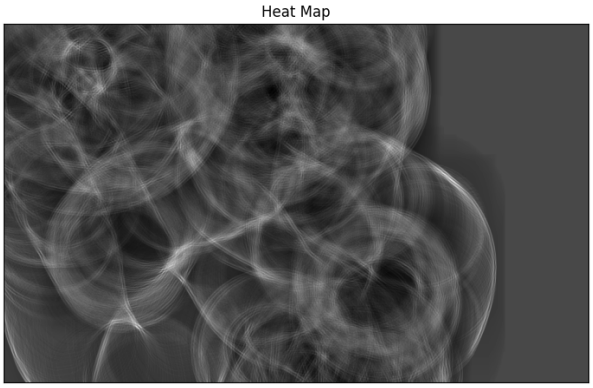


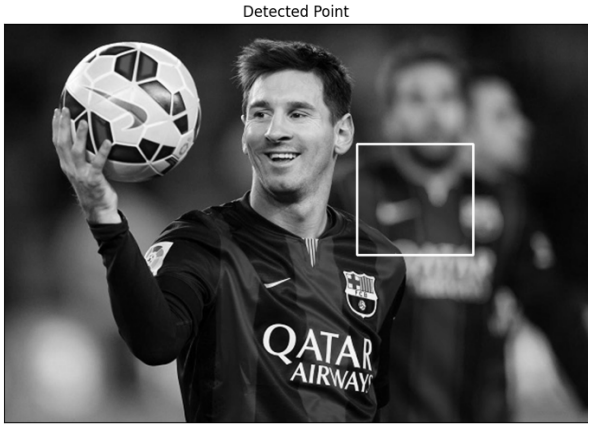
Iteration 4, Template Scale = 130%



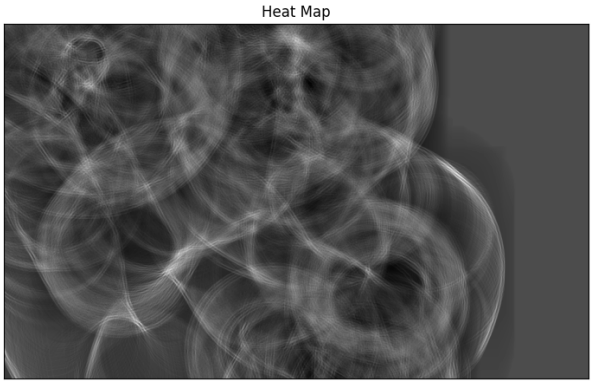


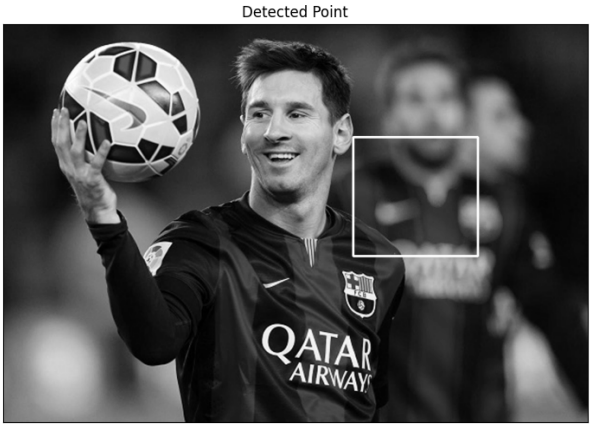
Iteration 5, Template Scale = 140%



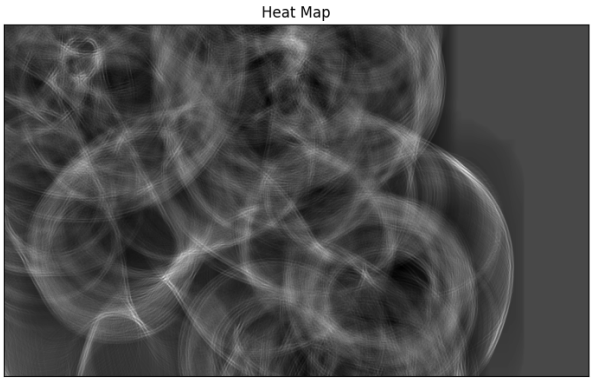


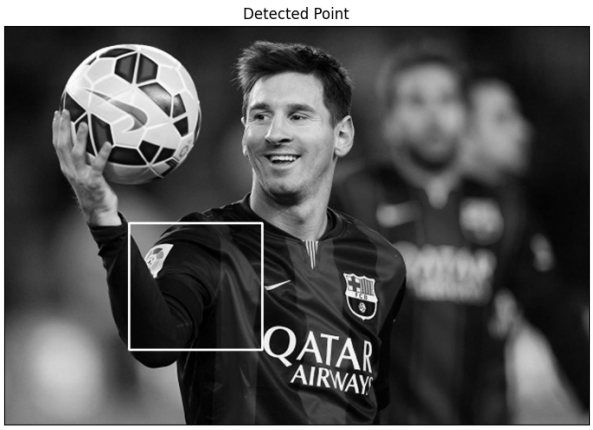
Iteration 6, Template Scale = 150%



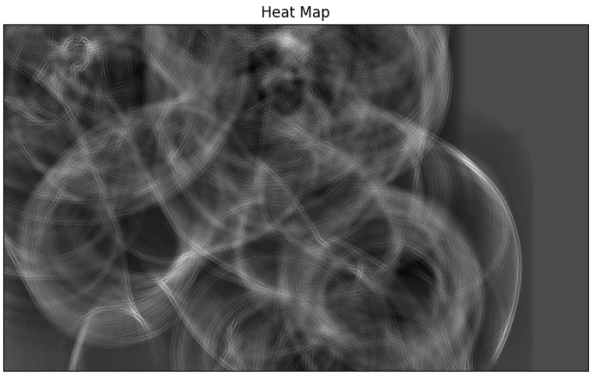


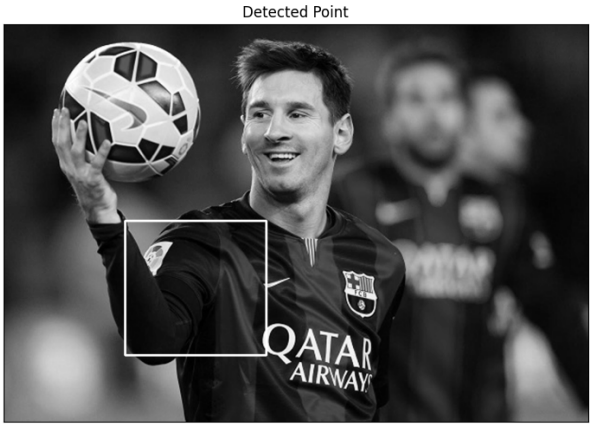
Iteration 7, Template Scale = 160%



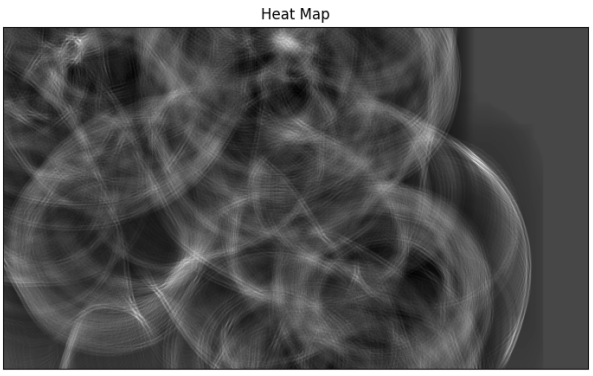


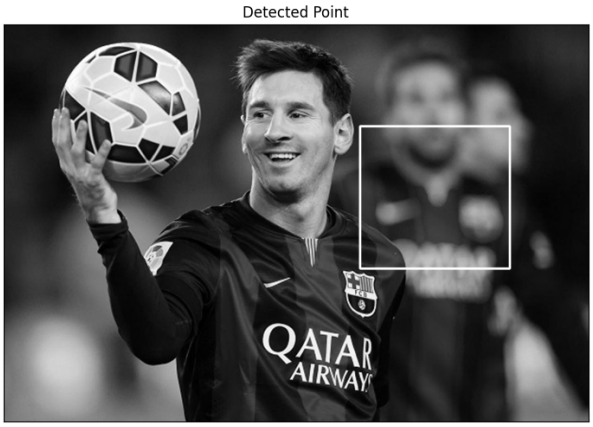
Iteration 8, Template Scale = 170%



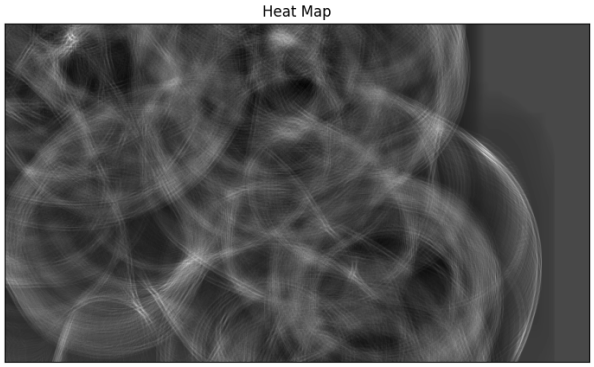


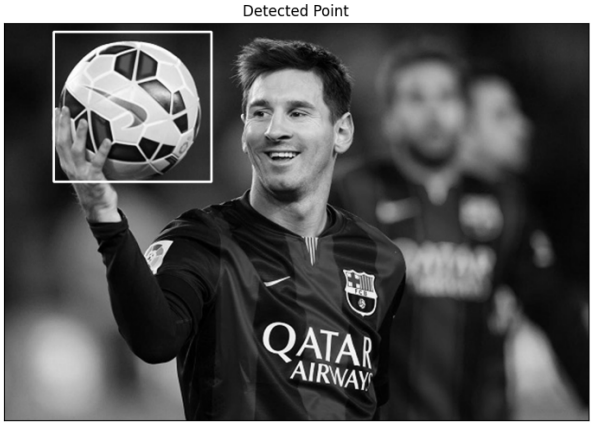
Iteration 9, Template Scale = 180%



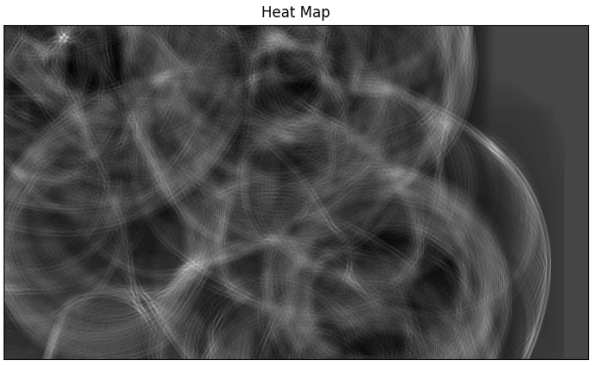


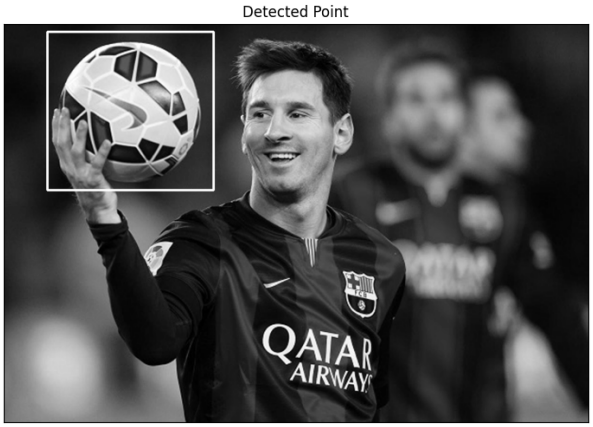
Iteration 10, Template Scale = 190%: At this template scaling factor, *matchTemplate()* was able to detect the circle.





Iteration 11, Template Scale = 200%: Circle detect - image and template are at the same size ratio now, as in Implement 3.





Bibliography

[1] Cpsc 425: Computer vision. https://www.cs.ubc.ca/~lsigal/425_2019W2/Lecture6b.pdf.

[2] Sahir Sofiane. Canny edge detection step by step in python — computer vision. https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123.

[3] temp. Cv gradient and laplacian filter, difference of gaussians dog. https://medium.com/temp08050309-devpblog/cv-3-gradient-and-laplacian-filter-difference-of-gaussians-dog-7c22e4a9d6cc.

[4] HuJ. X. Binjie. Template matching. https://www.sciencedirect.com/topics/engineering/template-matching.