CV701: Computer Vision Assignment 2

Abdul Ahad Butt
abdul.butt@mbzuai.ac.ae

Muhammad Abdullah Sohail
MAbdullah.Sohail@mbzuai.ac.ae

Ilmuz Zaman Mohammed Zumri
mohammed.zumri@mbzuai.ac.ae

1. Task 1

1.1. Canny Edge Detector

The Canny edge detector is an edge detection operator that uses a multistage algorithm to detect a wide range of edges in images. The process of Canny edge detection algorithm can be broken down to six different steps:

- **Convert to Grayscale**: If an image is in RGB, we first convert the image to grayscale
- **Gaussian Blur**: Apply a Gaussian filter to smooth the image in order to reduce noise
- **Calculate Gradient**: Find the intensity of the gradients using the *Sobel filter* [2]
- **Non-Maximal Suppression**: Thin edges by suppressing pixels that are not part of an edge
- **Double Thresholding**: Classify pixels as strong, weak, or invalid based on intensity
- **Hysteresis**: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges

Figure 1 shows the original fence.jpg image we will apply our canny edge detector on.

1.1.1 Convert to Grayscale

We first convert our image from it's original color space to grayscale. The following codeblock accomplishes this task:

```
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

This function converts the image from RGB to Grayscale using the formula [1]:

$$Y = 0.299R + 0.587G + 0.114B$$

The resulting grayscale image is shown in 2

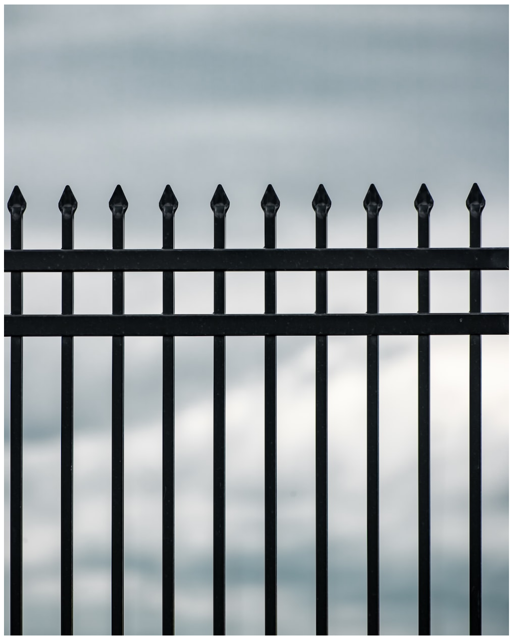


Figure 1. fence.jpg

1.1.2 Gaussian Blur

We apply a Gaussian filter on the new grayscale image. This helps in reducing noise and unwanted details in the image. The Gaussian function is given as:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \qquad (1)$$

where x and y are the distances from the center of the kernel in horizontal and vertical direction, and σ is the standard deviation and controls the "spread" of our bell curve.

The following code applies convolves a gaussian filter with the image in grayscale.

```
def get_gaussian_custom(shape:int,
↪   sigma:float=0.5):
    x = np.linspace(-shape, shape, shape)
    y = np.linspace(-shape, shape, shape)
```

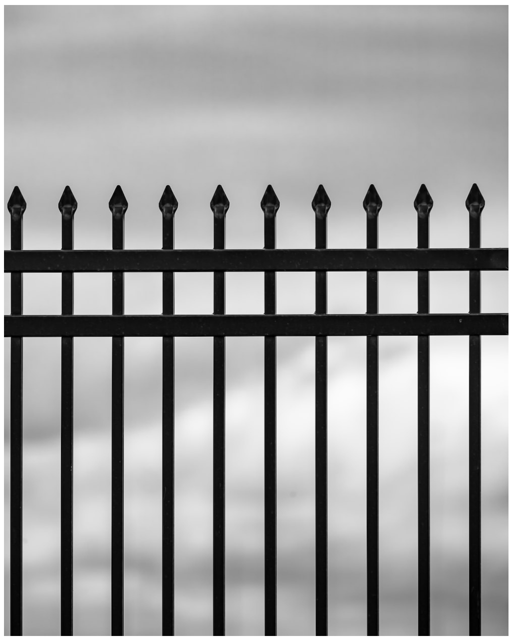


Figure 2. fence.jpg converted to grayscale

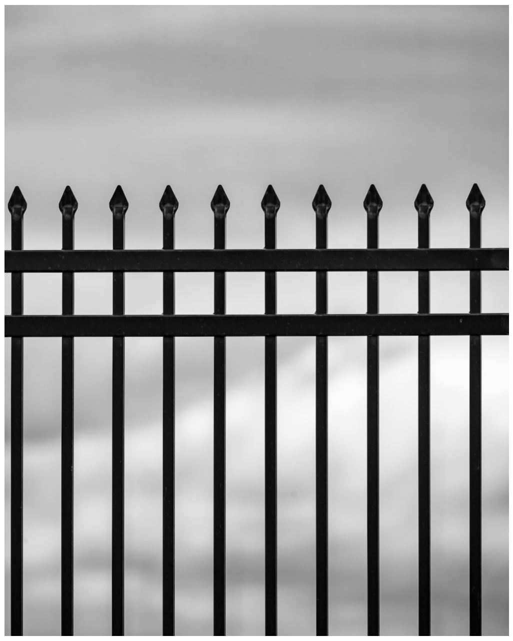


Figure 3. Applying Gaussian Blur

```
    x, y = np.meshgrid(x, y)

    h = np.exp( -(x*x + y*y) / (2.*sigma*sigma) )

    # ? Normalizing
    sumh = h.sum()
    if sumh != 0:
        h /= sumh
    return h


def gaussian_blur_custom(image:np.ndarray,
↪   kernel_size:int, sigma:float=0.5) ->
↪   np.ndarray:
    kernel = get_gaussian_custom(kernel_size)
    blurred_img = cv2.filter2D(image, ddepth=-1,
    ↪   kernel=kernel)

    return blurred_img
```

The resulting image is shown in 3.

1.1.3 Gradient Calculation

The next step is to calculate the gradient of the blurred grayscale image using Sobel Filters [2]. We calculate the edge directions in the x and y directions by convolving Sobel Kernels with the image. The sobel kernels are given below:

$$G_x = \begin{pmatrix} -1 & 0 & -1 \\ -2 & 0 & -2 \\ -1 & 0 & -1 \end{pmatrix} \tag{2}$$

$$G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 2 \end{pmatrix} \tag{3}$$

where G_x denotes the horizontal sobel filter and G_y denotes the vertical sobel filter.

We convolve the sobel filters with our smoothed grayscale image to get edges in the x and y directions:

$$I_\text{x} = G_x * I_\text{smoothed, grayscale} \tag{4}$$
$$I_\text{y} = G_y * I_\text{smoothed, grayscale} \tag{5}$$

At each point of the image, we calculate the gradient and direction using the following formuala:

$$M = \sqrt{I_x^2 + I_y^2} \tag{6}$$
$$\theta = \arctan\left(\frac{I_y}{I_x}\right) \tag{7}$$

where M is the magnitude of the edges and θ is the direction of the edges.

We use the following codeblock to calculate the magnitude and direction of the gradient:

```
def get_sobel_kernels(ksize:int):
    kernel_x = np.array([
        [1, 0, -1],
        [2, 0, -2],
        [1, 0, -1]
    ])

    kernel_y = np.array([
```

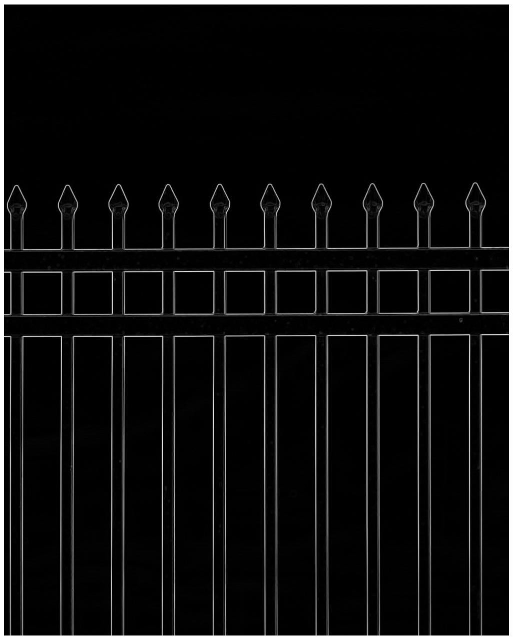


Figure 4. Magnitude of Edge Detection Algorithm

```
        [ 1,  2,  1],
        [ 0,  0,  0],
        [-1, -2, -1]
    ])

    return kernel_x, kernel_y


def calc_gradient_custom(img:np.ndarray,
↪   ksize:int):
    ddepth = cv2.CV_64F

    k_x, k_y = get_sobel_kernels(ksize=ksize)
    grad_x = cv2.filter2D(img, ddepth, k_x)
    grad_y = cv2.filter2D(img, ddepth, k_y)


    magnitude = np.sqrt(np.square(grad_x) +
    ↪   np.square(grad_y))
    # ? Converting from rads to degrees
    angle = np.arctan2(grad_y, grad_x) * (180 /
    ↪   np.pi)


    # ? Scaling the magnitude between 0 and 255
    magnitude = (magnitude - magnitude.min()) /
    ↪   (magnitude.max() - magnitude.min()) * 255
    return magnitude, angle
```

The magnitude of the image is given in 4.

1.1.4 Non Maximal Suppression

Ideally edges should be thin, however the result from our Gradient Calculation has multiple pixels in a neighbourhood depicting an edge.

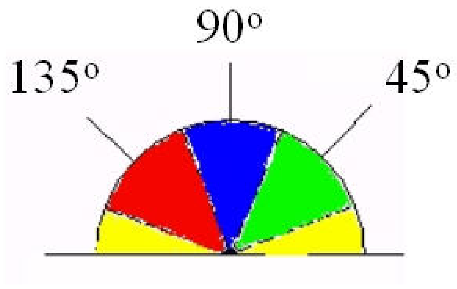


Figure 5. Angle Categories for Non Maximal Suppression

The idea behind Non Maximal Suppression to check whether the current pixel is the strongest (i.e., has the highest gradient magnitude) compared to its neighbors in the gradient direction. If it's not the strongest, it's suppressed (set to zero).

First off, the gradient direction is quantized into 4 bins:

- 0° - θ between 0° and 22.5° or between 157.5° and 180° - compare with left and right neighbors
- 45° - θ between 22.5° and 67.7° - compare with diagonal neighbors (top-right and bottom-left)
- 90° - θ between 67.5° and 112.5° - compare with top and bottom neighbors
- 135° - θ between 112.5° and 157.5° - compare with diagonal neighbors (top-left and bottom-right)

This quantization is illustrated in 5.

Then depending on the comparison from the angle quantization, we either consider a pixel an edge if it's intensity is higher than it's neighbours, or set it to 0 otherwise.

We use the following codeblock to apply non-maximal suppression to the edge image:

```
def non_maximal_custom(magnitude: np.ndarray,
↪   angle: np.ndarray):
    non_max = np.zeros_like(magnitude)

    # ? Only getting angles from 0 to 180 degrees
    angle = angle % 180

    # ? Angles are divided into 4 regions
    for i in range(1, magnitude.shape[0] - 1):
        for j in range(1, magnitude.shape[1] -
        ↪   1):
            # Suppress pixels based on angle
            ↪   direction
            try:
                if (0 <= angle[i, j] < 22.5) or
                ↪   (157.5 <= angle[i, j] <=
                ↪   180):
                    q = magnitude[i, j+1]
                    r = magnitude[i, j-1]
                elif 22.5 <= angle[i, j] < 67.5:
                    q = magnitude[i+1, j-1]
```

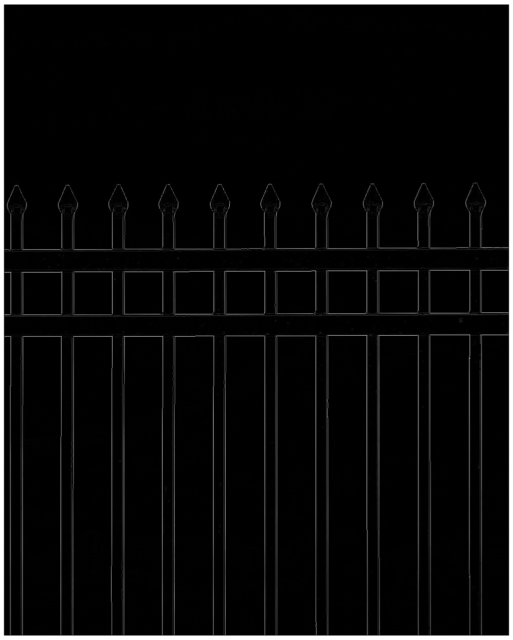


Figure 6. After Non-Maximal Suppression

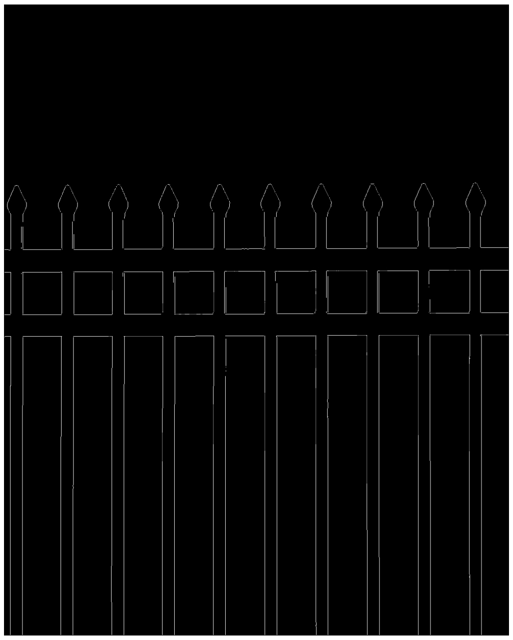


Figure 7. After Double Thresholding

```
                    r = magnitude[i-1, j+1]
                elif 67.5 <= angle[i, j] < 112.5:
                    q = magnitude[i+1, j]
                    r = magnitude[i-1, j]
                elif 112.5 <= angle[i, j] <
                ↪  157.5:
                    q = magnitude[i-1, j-1]
                    r = magnitude[i+1, j+1]

                if magnitude[i, j] >= q and
                ↪  magnitude[i, j] >= r:
                    non_max[i, j] = magnitude[i,
                    ↪  j]
                else:
                    non_max[i, j] = 0
            except IndexError:
                pass

    return non_max
```

The resulting image after applying non-maximal suppression is shown in 6. You can see that the edges have been ”thinned” out.

1.1.5 Double Thresholding

Next we differentiate between weak edges and strong edges using a double threshold. This will be used in the next step to refine weak edges. The high threshold ensures that only the most prominent and certain edges (high gradients) are kept, whereas the low threshold helps preserve weak edges that could be part of real edges but may have low gradients due to lighting conditions, surface texture, or noise (etc).

We use the following codeblock to perform double thresholding:

```
def double_thresh_custom(non_max: np.ndarray,
↪  thresh1:int, thresh2:int):
    # ? if inverted
    if thresh1 > thresh2:
        thresh1, thresh2 = thresh2, thresh1

    res = np.zeros_like(non_max)
    strong_i, strong_j = np.where(non_max >=
    ↪  thresh2)
    weak_i, weak_j = np.where((non_max < thresh2)
    ↪  & (non_max > thresh1))

    res[strong_i, strong_j] = thresh1
    res[weak_i, weak_j] = thresh2

    return res
```

The result after applying double thresholding can be seen at 7.

1.1.6 Hysteresis

Finally, we perform edge tracking via hysteresis. Basically, we define a weak edge to be a strong edge, if it is connected to another strong edge. If a weak edge is not connected to any strong edge, it is discarded. The following codeblock is used to perform edge-tracking via hysteresis:

```
def hysteresis_custom(result:np.ndarray, thresh1,
↪  thresh2):
    if thresh1 > thresh2:
        thresh1, thresh2 = thresh2, thresh1

    for i in range(1, result.shape[0] - 1):
        for j in range(1, result.shape[1] - 1):
            if (result[i, j] == thresh1):
```

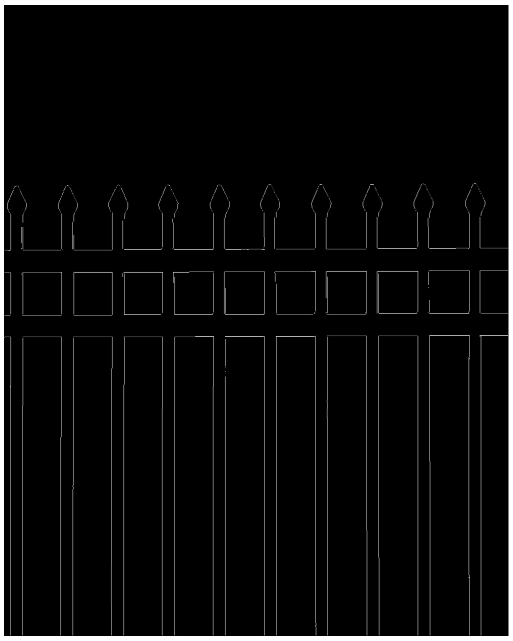


Figure 8. Our Custom Canny Edge Detector Result

Mean Squared Error	PSNR
1.75	45.71 db

Table 1. Distance metrics between the output image of our implementation and OpenCV's

```
            if ((result[i+1, j-1:j+2] ==
            ↪  thresh2).any() or
            ↪   (result[i-1, j-1:j+2] ==
            ↪  thresh2).any() or
                (result[i, [j-1, j+1]] ==
                ↪  thresh2).any()):
                result[i, j] = thresh2
            else:
                result[i, j] = 0

    result = result.astype(np.uint8)

    return result
```

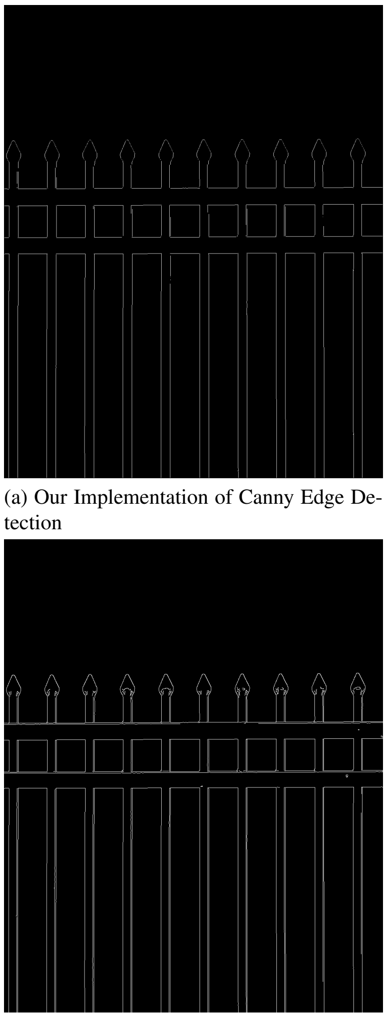
The final result of our custom Canny Edge Detector can be seen at 8.

1.1.7 Comparison with OpenCV

We show the the results of our implementation alongside OpenCV's at 9. OpenCV's implementation is more defined and the edges are more visible.

To calculate the difference between the output from our method and the output from OpenCV's implementation, we calculated distance metrics and show them in table 1.

We have also tabulated the time it takes for OpenCV to implement canny edge detection vs our approach in table 2.


(a) Our Implementation of Canny Edge Detection

(b) OpenCV Result of Canny Edge Detection

Figure 9. Our implementation vs OpenCV's

Our time	OpenCV's time
0.98767s	0.00058s

Table 2. Time taken between the output image of our implementation and OpenCV's

We can see that OpenCV's implementation is far quicker, possibly because of the matrix multiplications whereas we have used for loops.

1.2. Counting the Number of Posts

With our given edge map, we find it is very straighforward to count the number of posts.

Our approach is very task dependent, and will most likely only work with shapes with vertical edge lines.

We choose a horizontal line that goes through all the

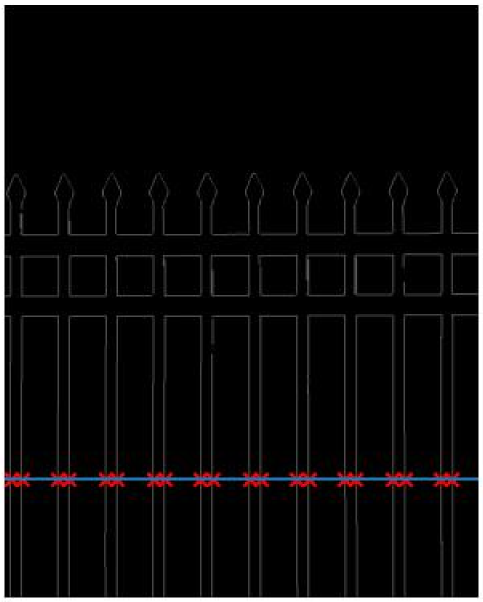


Figure 10. Counting the number of posts
red x denotes points on the blue line where the edge map is non zero

posts and count the number of times the intensity is higher than 0 in the edge map. This number is 2×the number of posts. We simply divide this number by 2 to get posts.

Our code is available below:

```
def naive_count(edge_img:np.ndarray, y:int):
    line = edge_img[y, :]
    edge_points = np.where(line != 0)
    return edge_points[0].tolist()


Y_ = 1000
edge_points = naive_count(custom_edges, Y_)
print(f'Total Posts: {len(edge_points) // 2}')
```

Our approach is illustrated in 10. The number of times the point is non-zero on the blue line is represented by the red x. There are 20 red x's. Therefore the number of posts is simply $20/2 = 10$.

As stated before, this method is application dependent and requires user input (which y coordinate to choose).

2. Task 2

2.1. Blob detection using Laplacian of Gaussian

Blob detection locates circular or elliptical areas in an image that stand out from their surroundings. The Laplacian of Gaussian used here for blob detection combines the Laplacian operator with Gaussian smoothing to effectively detect blobs of various sizes. The process is outlined in the steps below:

1. **Gaussian Smoothing**:

   The Gaussian smoothing function reduces the noise in the image.

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

   where (x, y) are the coordinates in the image, σ is the standard deviation (scale) of the Gaussian filter. We used our Gaussian filter function here.

```
smoothed_image = gaussian_filter(image_data,
↪  sigma=stddev)
```

2. **Laplacian Operator**:

   The Laplacian operator is used to compute the second-order derivatives of the image. For a 2D image, the Laplacian is defined as:

$$\nabla^2 I(x, y) = \frac{\partial^2 I(x, y)}{\partial x^2} + \frac{\partial^2 I(x, y)}{\partial y^2}$$

   where $I(x, y)$ is the intensity of the image at coordinates (x, y).

   In practice, the Laplacian is often implemented using a discrete convolution operation with a kernel. We used the following kernel.

$$\text{Laplacian\_kernel} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

   The Laplacian serves as the tool that determines the areas where the variations in brightness are most intense. This helps in identifying the edges and borders.

3. **Laplacian of Gaussian (LoG)**: The Laplacian is applied on the image after Gaussian smoothing. To do this, the image filtered using the Gaussian filter is convolved with the Laplacian kernel.

$$LoG(x, y, \sigma) = \nabla^2 [G(x, y, \sigma) * I(x, y)]$$

   where $G(x, y, \sigma)$ is the Gaussian filter, and $LoG(x, y, \sigma)$ is the Laplacian of the Gaussian-smoothed image.

   The response is normalized by scaling it with σ^2.

   We use the following codeblock to perform LoG.

```
def calculate_log(image_data, stddev):
    blurred_image =
    ↪  gaussian_filter(image_data,
    ↪  sigma=stddev)
    laplacian_kernel = np.array([[0, 1, 0],
    ↪  [1, -4, 1], [0, 1, 0]])
    laplacian_result =
    ↪   convolve(blurred_image,
    ↪   laplacian_kernel)
    log = (stddev ** 2) * laplacian_result
    #log = laplacian_result
    return log
```

4. **Scale-Space Construction**:

   In order to detect blobs of different sizes, the Laplacian of Gaussian is applied at multiple scales. We used scales (σ) of 1, 2, 3, 4, 6, 8, 10 and 12. The response of the image for each of these scales forms a "scale-space".

```
# Create a list to store responses at
↪   different scales
scale_responses = []
sigma_levels = []

for sigma in scales:
    log_result =
    ↪   calculate_log(grayscale_image, sigma)
    log_result = log_result ** 2
    scale_responses.append(log_result)
    sigma_levels.append(sigma)
```

5. **Non-Maximum Suppression in Scale-Space**:

   Non maximum suppression is then performed on the scale-space in both spatial and scale dimensions to detect the location of blob centers. This ensures that only the local maxima are retained.

```
scale_responses = np.array(scale_responses)

neighborhood_shape = np.ones((3, 3, 3))
maxima_in_scale_space =
↪   maximum_filter(scale_responses,
↪   footprint=neighborhood_shape,
↪   mode='constant')

local_maxima = (scale_responses ==
↪   maxima_in_scale_space)
local_maxima[scale_responses < threshold] = 0
blob_coordinates = np.argwhere(local_maxima)
```

6. **Blob Detection**:

   The coordinates of the blobs are then identified by finding local maxima in the filtered scale-space. The size of each blob is estimated based on the corresponding sigma value. Only blobs that fall within a specified size range are kept.

```
identified_blobs = []
for coordinate in blob_coordinates:
    scale_idx, y, x = coordinate
    sigma = sigma_levels[scale_idx]
    blob_radius = sigma * np.sqrt(2)
    if min_size <= blob_radius <= max_size:
        identified_blobs.append((x, y,
        ↪   blob_radius,
        ↪   scale_responses[scale_idx, y, x],
        ↪   sigma))
```

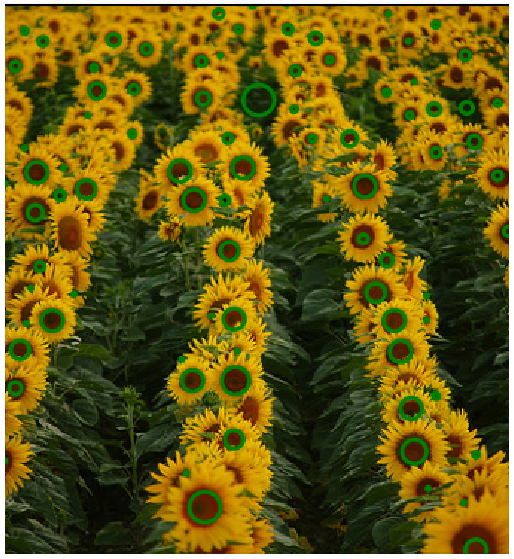


Figure 11. Blob Detection using custom function

7. **Merging Nearby Blobs**: In order to avoid redundant detections, blobs that are close to each other are merged. The process ensures that blobs within a certain distance are combined while retaining only the one with the highest response.

```
final_blobs = []
for i, (x1, y1, r1, response1, sigma1) in
↪   enumerate(identified_blobs):
    merged = False
    for j, (x2, y2, r2, response2, sigma2) in
    ↪   enumerate(final_blobs):
        distance = np.sqrt((x1 - x2) * 2 +
        ↪   (y1 - y2) * 2)
        if distance < proximity:
            if response1 > response2:
                final_blobs[j] = (x1, y1, r1,
                ↪   response1, sigma1)
            merged = True
            break
    if not merged:
        final_blobs.append((x1, y1, r1,
        ↪   response1, sigma1))
```

8. **Final Blob Visualization**:

   Once the blobs have been detected and merged, they are visualized by drawing circles at the blob locations. The circles take the radius of the corresponding scale.

```
for (x, y, radius, _, sigma) in final_blobs:
    blob_circle = patches.Circle((x, y),
    ↪   radius, linewidth=2,
    ↪   edgecolor='green', facecolor='none')
    axis.add_patch(blob_circle)
plt.show()
```

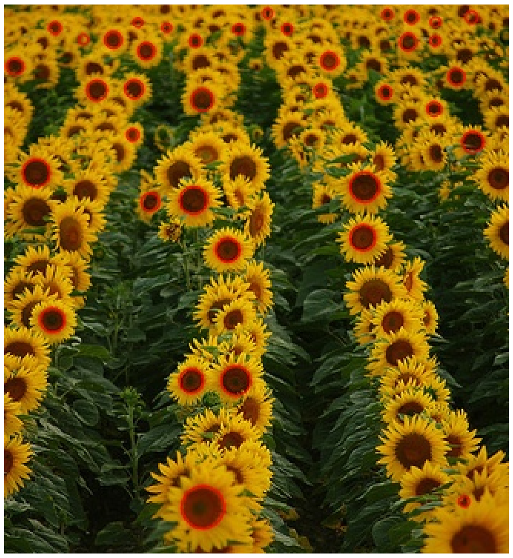


Figure 12. Blob Detection using OpenCV Function

2.1.1 Code for OpenCV Function

In order to compare our results with a pre-built function, we have used the OpenCV library with its in-built blob detection function. The code is given below.

```
import cv2
import numpy as np

image = cv2.imread('flowers.jpg')
if image is None:
    raise ValueError("Image not found. Ensure the
    ↪  filename is correct.")
detector = cv2.SimpleBlobDetector_create()

keypoints = detector.detect(image)

blank = np.zeros((1, 1))
blobs = cv2.drawKeypoints(image, keypoints,
↪  blank, (0, 0, 255),
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

cv2.imwrite('flower_with_blobs.jpg', blobs)

cv2.imshow("Blobs Detected", blobs)
```

2.1.2 Comparison

Fig. 11 and 12 portray the the outputs of our custom function for LoG and the in-built OpenCV Function used for blob detection. As we can see that the result for the In-built function for OpenCV is more accurate and smooth than our custom function. The reason for this is the fact that even we fine-tuned our hyperparameters (e.g. threshold) for the function still an in-built function carries a higher optimization efficiency, algorithm complexity and parameter fine-tuning which results in better blob detection.

2.2. Performance of the blob detection method with and without normalized scale variations

Fig. 13 shows the result of our custom blob detection function with the same threshold value set in the previous part, however, without normalized scaling applied. No blobs our detected in the image due to the fact that for our set threshold, the Laplacian response completely dies out, and thus, no blobs are detected. The solution for this is to decrease the threshold to see the Laplacian response which is shown in the next part.

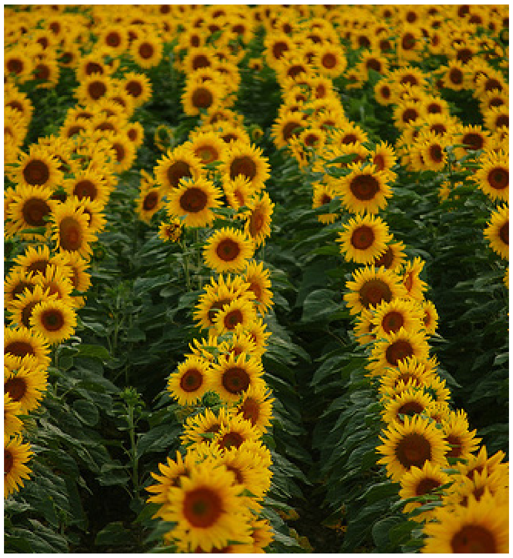
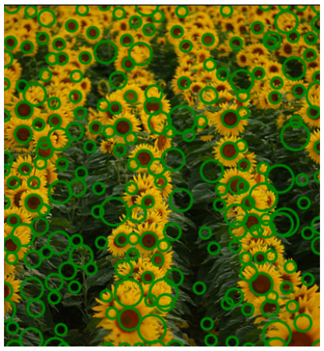


Figure 13. Blob Detection for same threshold but without normalised scaling

2.3. Performance of the blob detection method with and without normalized scale variations under different threshold values.

Fig. 14,15, and 16 show our blob detection function results for threshold = 0.1,10 and 200 respectively, with and without applying normalized scaling. As we can notice from the images as the threshold increases, the number of blobs on the image decreases because the blobs less than the threshold are rejected by the algorithm. There is also a significant difference in the result for all three thresholds when we remove the normalized scaling. The number of blobs is significantly less than the output with normalized scaling. The result is as expected because removing the scaling makes the Laplacian response die out for higher thresholds. Thus, the thresholds have to be significantly decreased to make the blobs appear.


(a) With Normalized Scaling

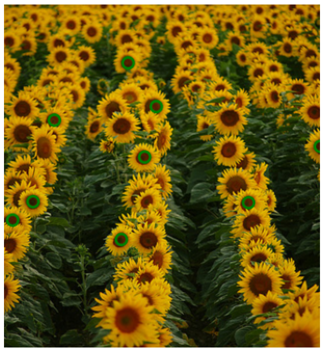
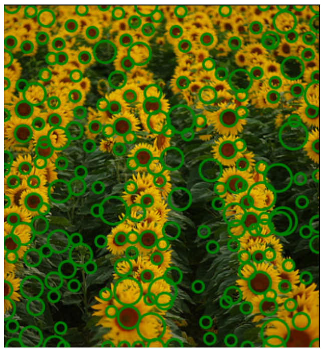

(b) Without Normalized Scaling

Figure 15. Threshold=10


(a) With Normalized Scaling

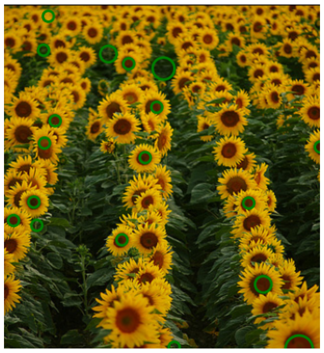
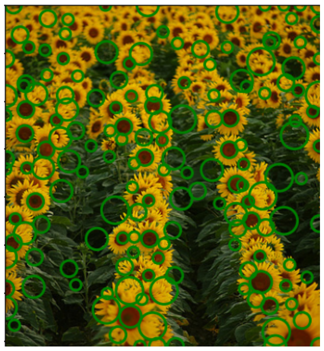

(b) Without Normalized Scaling

Figure 14. Threshold=0.1


(a) With Normalized Scaling

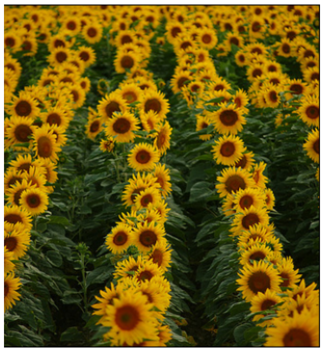

(b) Without Normalized Scaling

Figure 16. Threshold=200

References

[1] How opencv converts to grayscale. https://stackoverflow.com/questions/19181323/what-grayscale-conversion-algorithm-does-opencv-cvtcolor-use. 1

[2] Sobel operator. https://en.wikipedia.org/wiki/Sobel_operator. 1, 2