Assignment IS2U - IS2U

CV701 - Human and Computer Vision

Khalil Alblooshi, Yevheniia Kryklyvets, Sebastian Cavada, Mohamed Insaf Ismithdeen

Task 1

Distinguishing objects from one another and background can be done via edge detection algorithms.

Task 1.1 - Canny edge detection

The Canny edge detection algorithm aims to identify and extract the edges of objects within an image by reducing noise and preserving important edge features. It analyzes the change in magnitude among pixel intensities to determine the existence of an edge. The way in which canny was implemented follows [2] and is summarized as:

- Step 1: **Smoothing the image using a Gaussian filter** 10. This step is important for the noise reduction in the input image. It is known that excess noise can introduce false edges, that could compromise the accuracy of the implemented algorithm.
- Step 2 and 3: **Computation of image derivatives and finding the magnitude and orientation of the gradients.** The gradient measures how fast intensity changes in each pixel's location. To achieve that the concept of derivatives is being used to determine both gradient and magnitude (Sobel kernel for X and Y axis).

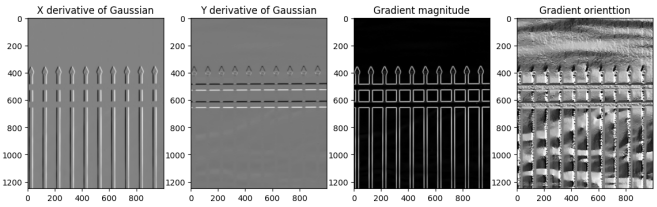


Figure 1: X/Y derivatives, Gradient magnitude, and orientation output

As is evident from the figure 1 the X derivative computation highlights the vertical edges, due to the fact that it calculates the difference between pixel intensities while sweeping it horizontally. Similarly in the Y derivative, the computation shows the horizontal edges as the calculation of pixels is conducted vertically.

The third image shows the summation of edges (Gradient magnitude), which is done by finding the Pythagorean of both derivatives ($\sqrt{(dx^2 + dy^2)}$)[5], thus representing the overall change in pixel intensity in all directions. ($\theta = 2 \times \arctan(dx, dy)$)[5] is the formula that we used to calculate the Gradient Orientation. This is needed in order to preserve the edges with a major change in intensity in a specific direction (fining the edges).

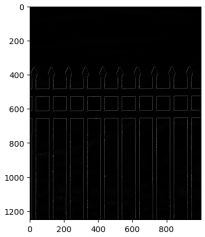


Figure 2: Fined edges

- Step 4: **Applying the Non-maximum suppression.** This step is crucial for representing actual edges in the image. The algorithm finds local maxima of gradient magnitude along the gradient direction and suppresses all the non-maximums.
- Step 5: **Applying high and low thresholds - hysteresis.** Within the double threshold part edges are categorized into three groups: non-edges (that are being discarded), strong edges (definite edges), and weak edges.

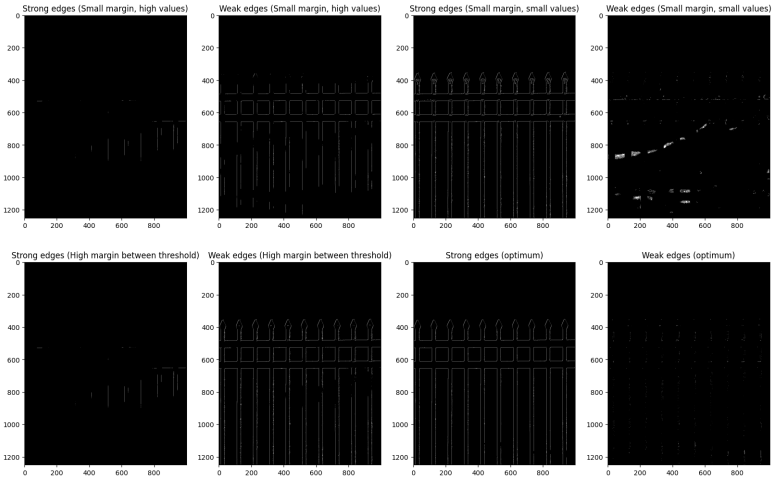


Figure 3: Comparison between different thresholding values

Our trials showed that neither big nor small differences between the threshold values work properly to highlight edges, since the small differences (when both thresholds are high in value) result in the elimination of major edges and thus failure in detection of continuous ones.

In case both thresholds are small in value, more edges with noise appear, which indicates it becomes harder for the algorithm to differentiate between weak and strong edges. When having a difference margin between the two thresholds, most of the edges are considered as weak edges which makes important ones dispensed.

In reference to the results of Canny's function in figure 4 our implementation shows exact same edge highlight, yet has different white shades on them. We have experimented with different values for the Gaussian kernel size, the results evidently show that 15 was the closest to the optimal value.

That is because it allows having clear highlights of the fence edges with the least noise, hence we considered using it with both implementations (pre-built and from scratch).

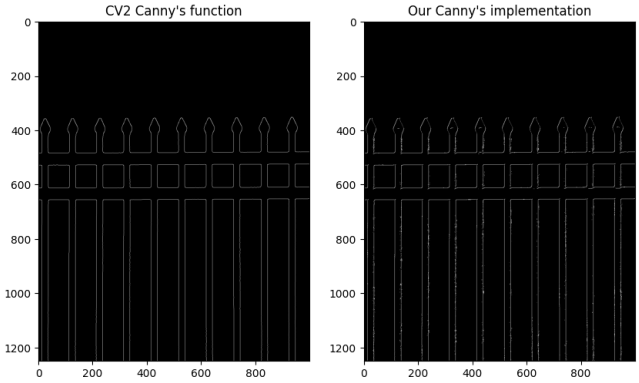


Figure 4: Comparison between OpenCV Canny's function and ours

Task 1.2 - Counting the number of posts

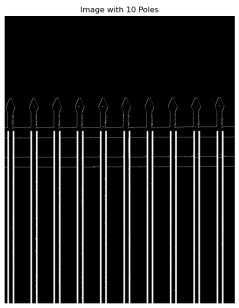


Figure 5: Counted posts

As is evident from figure 5, our implementation of Canny's edge detection and, post-count algorithm revealed that there are 10 posts presented in the image.

The algorithm is explained in the code C with comments. The main idea behind it is that the algorithm will loop over all the image columns and if a minimum number of white pixels is counted, then the column is marked as a single side of the post. The total number of vertical lines has to be divided by two to get the final answer since each post is made of two vertical lines.

Task 2

In the second half of the assignment, the Laplacian of Gaussian (LoG) filter for blob detection was implemented and analyzed in detail. The algorithm was implemented from scratch by following the steps outlined in the lecture notes [6] and some code was inspired by [4].

Implementation steps for the (LoG):

- Generate a number of kernels of different sizes and σ values. Different scales to find blobs of different sizes.
- Perform Laplacian of Gaussian (take the second derivative of an image with additional Gaussian smoothing), namely convolving kernels from the previous step with the original image.
- Detect blobs for every resulting image - save the center and radius of each activated area.
- Apply non-maximum suppression to reduce the number of blobs by removing duplication and/or overlapping.

Task 2.1 - Unnormalized Laplacian of Gaussian

Idea: Convolve an image with multiple different-scale "blob" filters and look for the zero-crossings in order to find edges in the specific scale space [6].

Laplacian is a second-order derivative scale invariant filter. Unlike first-order filters, Laplacian detects the edges at zero-crossings i.e. where the value changes from negative to positive and vice-versa. To reduce the noise effect, the image is first smoothed with a Gaussian filter and then we find the zero crossings using Laplacian. This two-step process is called the Laplacian of Gaussian (LoG) operation.

The formula for the unnormalized kernel can be found below:

$$LoG(x, y) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2 + y^2}{2\sigma^2}\right] \exp^{-\frac{x^2+y^2}{2\sigma^2}}$$

It is worth mentioning that the sign in front of the formula has a major impact on the detection process: minus means that dark blobs on a bright background are going to be detected. Conversely, a plus means that light blobs on a dark background will be found. [3]

The steps mentioned previously were meticulously carried out to detect dark blobs with light background. A detailed explanation and intermediate results on how the algorithm has been implemented can be found in the appendix section B. A great example of the final output for dark blob detection is found in Figure 6. An intermediate output before the non-maximum suppression can be found in Figure 16

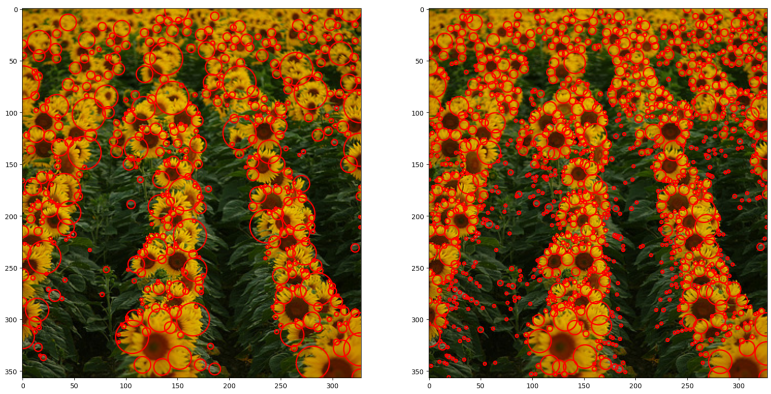


Figure 6: On the left is the result from our implementation, on the right - the output from the skimage library

Task 2.2 - Normalized Laplacian of Gaussian

To improve the existing implementation of the Laplacian of Gaussian filter a concept of normalization was introduced in the kernel creation step. Basically, all of the steps in this approach remain the same with the only change being the additional σ^2 term in the kernel-generating function. It transformed the first two steps of the LoG function into the following view [3]:

$$\nabla^2_{norm} L(X, \sigma) = \nabla^2_{norm} G(X, \sigma) * I(X)$$

$$\nabla^2_{norm} G(X, \sigma) = \sigma^2 \nabla^2 G(X, \sigma)$$

This scale-space normalization was introduced in order to minimize the decay of Laplacian response with the increase of sigma. The introduction of σ^2 (Appendix E) resolves the issue with the response of a derivative of the Gaussian filter to a perfect step edge decreasing with the increase of the scale figure. This helps with keeping the algorithm scale-invariant. The difference in responses for normalized and unnormalized can be seen in the figure 7

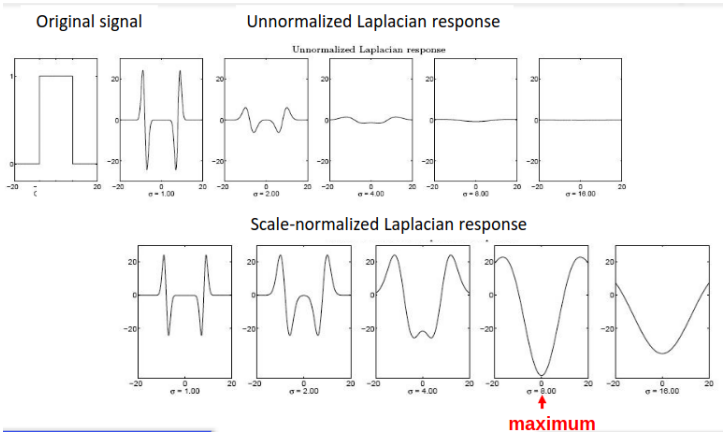


Figure 7: Difference between unnormalized and normalized function response

Complete comparison of the results with skimage library

The results were also tested and compared with the implementation of LoG on skimage [1].

As a result, we have three output images, figure 8: (1) - Normalized LoG, (2) - unnormalized Log, and (3) - Pre-built function. One difference is clear from the beginning, the algorithm of skimage has more granular results and the size of the detected blob is on average smaller than in our algorithms. This initial difference is mainly due to the parameters of the algorithm. Secondly, it can be shown that the center of the sunflower is not detected but rather the petals are. This means that the skimage algorithm is using the same function as us **but** with a plus in front of it. As per the two algorithms implemented from scratch, both resulted in precise detections. As expected, the normalized algorithm produced fewer outliers, due to the stronger response in the case of successful matching, and better results as it is more precise in finding the center of the blob. It has to be noted that the absence of smaller blobs in the first two images can be traced to one main factor: different sigma range selection than in the algorithm.

Task 2.3 - Comparison with different threshold values

When the kernels are convolved with an image, the outputs highlight the activation of the different kernels for every pixel [6]. Then these images are thresholded and only the values above some threshold are going to be used for blob detection. This effect can be seen in figure 9. In this step, the change of this parameter will be investigated.

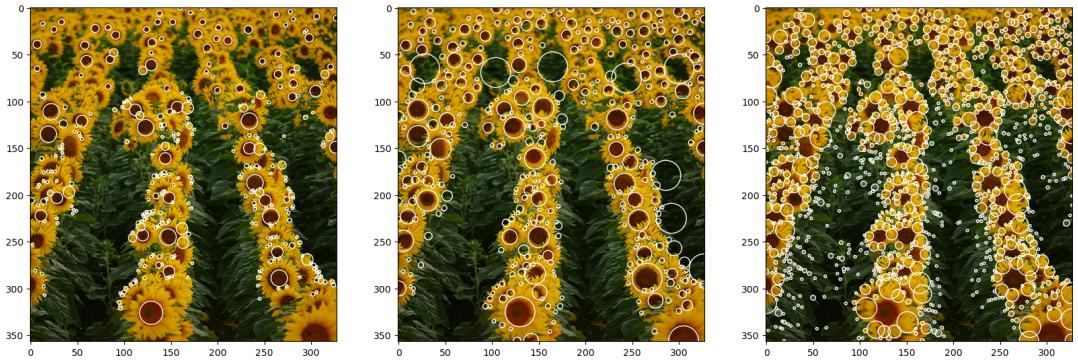


Figure 8: Blob detection using different algorithms (from left to right 1-2-3)

The threshold in this case, as shown in the code F, refers to the top x% of the pixels intensity in the image. E.g. if the maximum value is 0.45, then the threshold would be given by:

$$threshold = value_{max} - percentage * value_{max}$$

This trick had to be used instead of a constant value because in the first part of the experiment, the images produced by the different kernel convolutions had different maximum values as well as range of values.

In Figure 9 the effect of thresholding of the maximum value to a top $0.x$ percentage of the maximum value it is depicted. A further investigation on how our algorithm generalizes can be seen in the appendix section B.

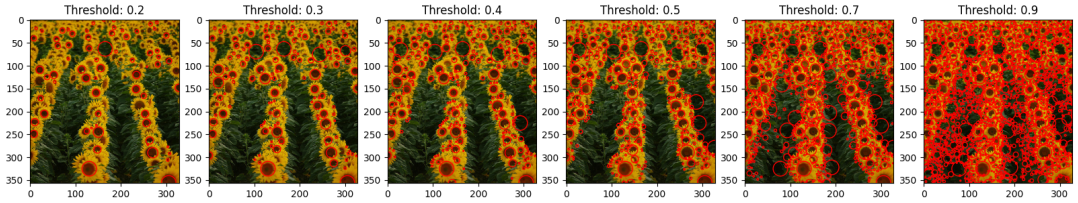


Figure 9: Various thresholds used to find the maximum percentage of the image that was allowed for each chunk size

References

[1] URL https://scikit-image.org/docs/stable/api/skimage.feature.html#skimage.feature.blob_log.

[2] S. Bashir. Educative answers - trusted answers to developer questions, 2023. URL https://www.educative.io/answers/what-is-canny-edge-detection.

[3] S. V. Fotin, D. F. Yankelevitz, C. I. Henschke, and A. P. Reeves. A multiscale laplacian of gaussian (log) filtering approach to pulmonary nodule detection from whole-lung ct scans, 2019.

[4] N. Kumar. Laplacian blob detector using python, Aug 2019. URL https://projectsflix.com/opencv/laplacian-blob-detector-using-python/.

[5] D. M. Yaqub. Cv701:lecture 4.1: Edge detection, 2023. URL https://lms.mbzuai.ac.ae/pluginfile.php/14442/mod_folder/content/0/CV701-Fall23-Lec4.1.pdf?forcedownload=1. Accessed on Oct 6, 2023.

[6] D. M. Yaqub. Cv701:lecture 5.1: Blob detection and local image features, 2023. URL https://lms.mbzuai.ac.ae/mod/folder/view.php?id=9848#:~:text=CV701%2DFall23%2DLec5.1.pdf. Accessed on Oct 6, 2023.

Appendices

A Figure for Noise reduction

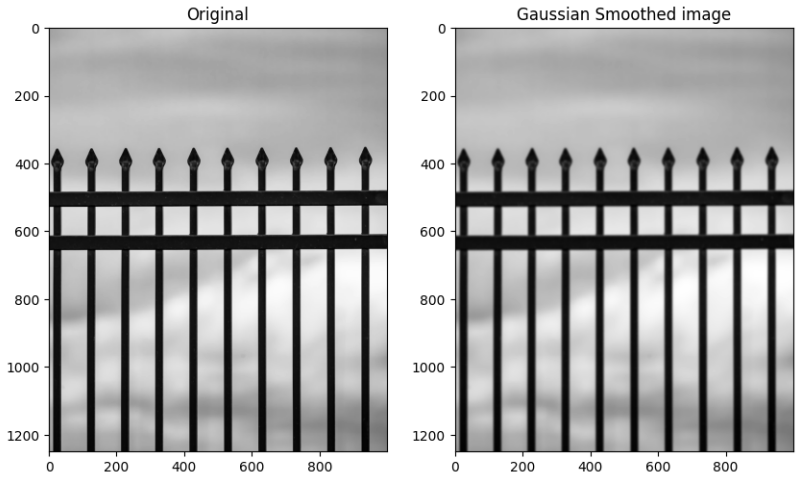


Figure 10: Noise reduction

The Gaussian kernel selected for our noise reduction was 15, this value was chosen after careful observation of Canny's performance on different window sizes.

B Detailed explanation of the LoG algorithm

Kernel generation

The kernels are generated at different scales to detect blobs of different dimensions. The sizes of the different scales are chosen according to some heuristics [3]. The sigma is multiplied by the i_{th} power of $\sqrt{2}$. This value has been proven to yield the best results. The code of reference is D.

The result of this first step can be clearly seen in figure 11

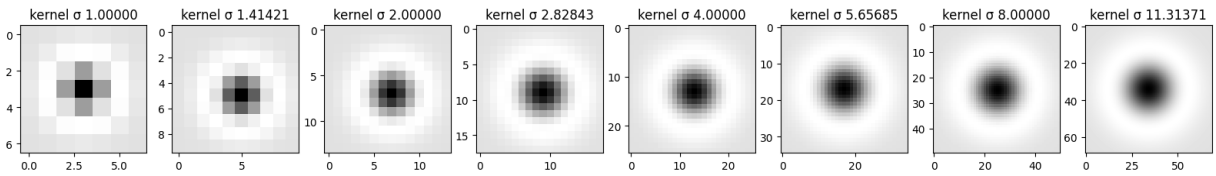


Figure 11: The kernels with different sigma and scale size

Image Convolution

This step involves the convolution of each kernel over the image. Figure 12 represents the results of convolutions with different kernels 11. The brightest spots represent the points in which the LoG has the most activation. These are the points that are supposed to be the blobs with the maximum likelihood.

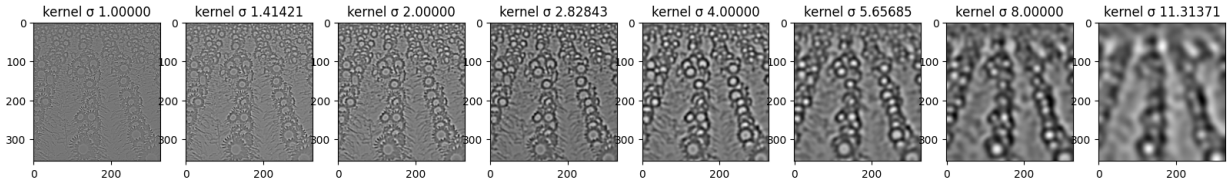


Figure 12: The convolved images with the respective sigma and kernel size

In this step, a further visualization of the convolution result was done to show the most active spots in the images. The various activations can be seen in Figure 13. It could be difficult to see for the first figures as the activation points are clusters of just a few pixels, however in the rightmost figure white spots that indicate where the blob is located are clearly seen.

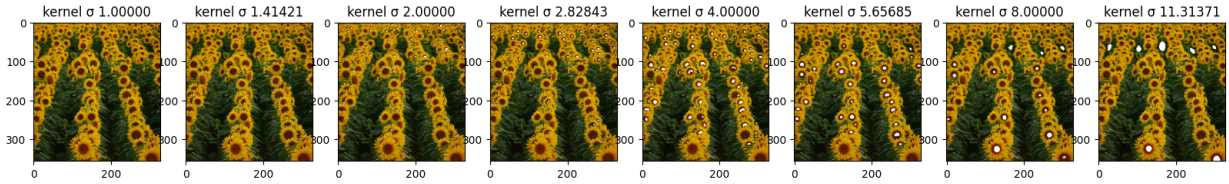


Figure 13: The most activated spots for every image scale

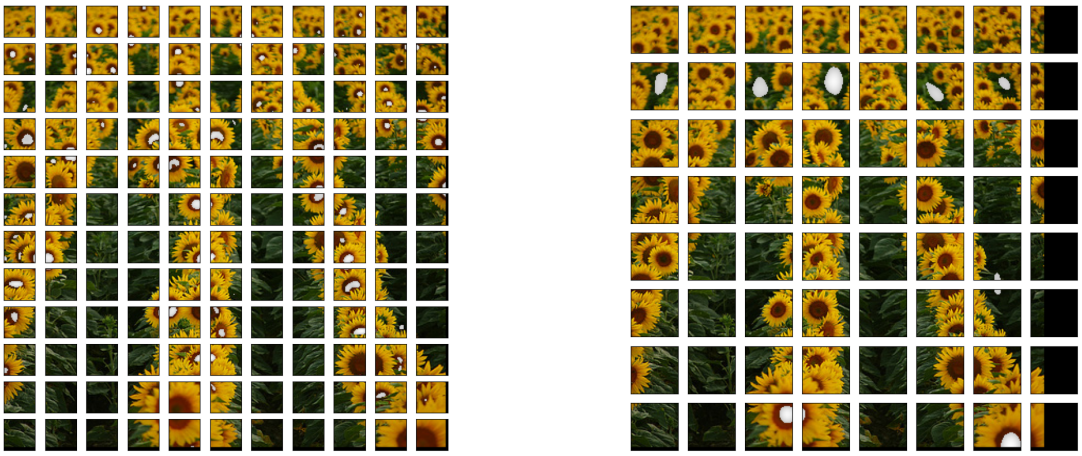
Blob Detection

First, we divide the image into chunks, with different sizes according to the sigma value. Then for every chunk, a maximum point is selected and it becomes the center of the blob for that particular value of sigma. We highlighted the patches where there is maximum activation in the image for the sigma of value 8. The most white part is in the middle of the blob usually and it is the exact point chosen for the center of the circle.

In order to achieve maximum accuracy we also needed to pad the image to divide the image without any pixel being left out. We padded the image with the black color as it doesn't affect the choice of the maximum point in the blob. Without this small tweaking, some blobs near the edge would have been left out. Figure 16b visually explains what is described in the previous lines.

After, we applied this method for every output image produced by the convolution and the result can be clearly seen in Figure 15.

This is how we found all the blobs. Last step is only needed to make the result more clear: non-maximum suppression algorithm.



(a) Size 30, with σ 4

(b) Size 45, with σ 11

Figure 14: Different examples for chunk division for blob detection

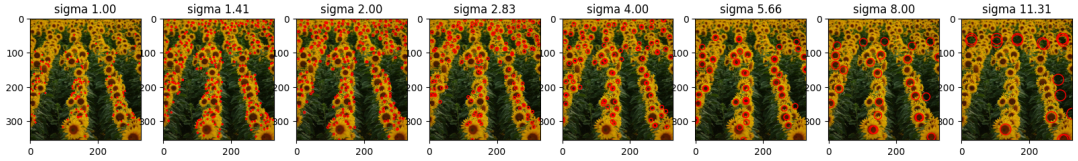
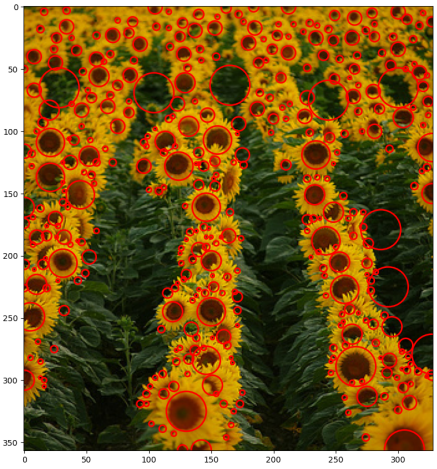


Figure 15: Multiple images with the different blobs found for every scale size

Non Maximum Suppression Algorithm

A very simple non-maximum-suppression algorithm was implemented which is both accurate and very fast. The idea is that we start from the bigger blobs and we give more importance to them. If smaller blobs are found inside the bigger blobs, they are being eliminated from the blobs' list. If the blobs are too close, less than the square root of the square of the radius of the two circles, then the second one is also eliminated. The list is shrunk dynamically so that the computational cost is always less than $\mathcal{O}(n^2)$.

(a) All blobs detected in a single image, overlapping and doubles, before non-maximum suppression



(b) Less number of blobs after the non-maximum suppression algorithm

Figure 16: Blobs detection, before and after

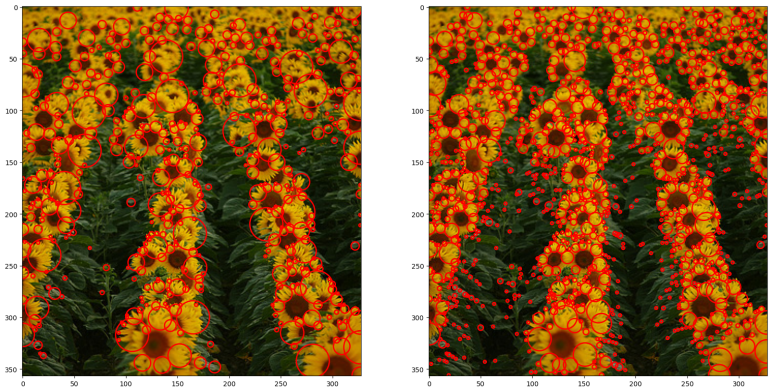


Figure 17: On the left is the result from our implementation, on the right - the output from the skimage library

Further investigation on different images

Following is just a small section about POC for our algorithms with other standard images. Using the figure 18 it can be proven that our algorithm is robust and performs well on different imput images. One thing that can also be noted for an accurate observer, is that in the first image the circles are evenly distributed along the poles, especially vertically, but there is a recurring distance between each other. This effect is likely to be due to the blob detection algorithm and the subdivision into chunks that leaves some space between blobs.

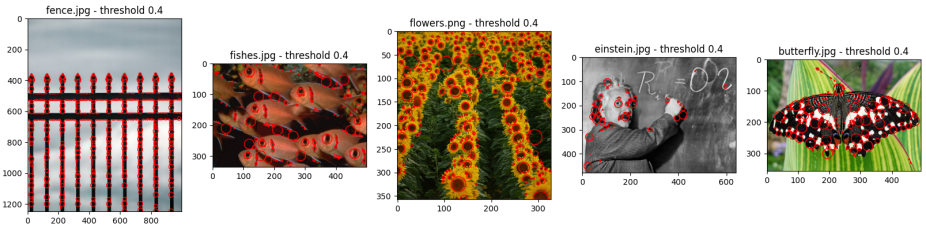


Figure 18: Testing the same algorithm on other images to check for robustness and generalization

C Code for counting the posts

```
def count_posts(edge_image, min_pixels=100, min_col_width=7):
    lines = 0                    #lines is set to 0 to count the number of lines
    col = 0                      #col is used to iterate over columns of the edge_image
    h = int(len(edge_image[0])) #height of the image
    w = int(len(edge_image[1])) #width of the image

    image_with_lines = edge_image.copy()

    while col < w:
        col_pixels = int(np.sum(edge_image[:, col]) / 255)
        #calculates the number of white pixels in the current column

        if col_pixels > min_pixels:
            lines += 1

            #Draw a thicker line to show the vertical line in the edge image
            col_start = col
            col_end = col + min_col_width
            image_with_lines[500:1250, col_start:col_end] = 255

            col += min_col_width
            #By incrementing col by min_col_width, the code skips over the width
            # of the detected line before continuing to search for the next line.
        col += 1
    posts = int(lines/2)
    return posts, image_with_lines
```

D Code for Laplacian 2d kernel creation

```
def laplacian_2d(sigma=1):

    n = np.ceil(sigma*6)
```

```
    y,x = np.ogrid[-n//2:n//2+1,-n//2:n//2+1]

    return -(1/(np.pi*sigma**4)) * (1- ((x**2 + y**2) / (2*sigma**2)))
        * np.exp(-(x**2+y**2)/(2*sigma**2))
```

E Code for Normalized Laplacian 2d kernel creation

```
def create_normalized_log_kernel(sigma = 1.0):
    # Step 1: Creating normalized LoG filter/kernel template
    # calculating LoG kernel - mix of gaussian smoothing and laplacian
    #second derivative to obtain kernel for convolving with the image in one step
    # main difference between this approach and the one without normalization
    # is additional multiplier - np.square(sigma)
    # It  is necessary to eliminate the effect of decreasing spatial derivatives
    # with the increase of scale (sigma)
    kernel_size = np.ceil(sigma * 6).astype(int)
    if kernel_size % 2 == 0:
        kernel_size += 1
    kernel = np.fromfunction(
        lambda x, y: (1 / (2 * np.pi * sigma ** 4)) *
                     np.array([
                      - (2.0 * sigma ** 2) + (x - (kernel_size//2)) ** 2
                      \+ (y - (kernel_size//2)) ** 2
                     ]) *
                     np.exp(-(((x - (kernel_size//2)) ** 2
                     \+ (y - (kernel_size//2)) ** 2)) / (2.0 * sigma ** 2)),
        (kernel_size, kernel_size)
    )
    return (kernel).squeeze()
```

F Code for convolving the kernels with an image

```
    def log_image(img_gray, n_kernels=9, threshold=0.2):

    """
    :param img: input image HAS TO BE GRAYSCALE (or single channel)
    :param min_sigma: minimum sigma value
    :param max_sigma: maximum sigma value
    :param steps: number of steps
    :return: log images
    """

    k = np.sqrt(2)
```

```
    sigma = 1

    kernels = []
    log_images = []
    sigma_list = []

    # threshold the log images
    log_images_thresholded = []

    for i in range(0,n_kernels):
        sigma_curr = sigma * k**i
        sigma_list.append(sigma_curr)

        kernel_log = laplacian_2d(sigma=sigma_curr)
        kernels.append(kernel_log)

        log_image = scipy.signal.convolve2d(img_gray, kernel_log,
        mode='same', boundary='symm')
log_images.append(log_image)

        # create a thresholded image and add them to a list
        log_image_thresholded = np.zeros_like(log_images[i])
        log_image_thresholded[log_images[i] >= max(log_images[i].flatten())
            -(threshold * max(log_images[i].flatten()))] = 1
        log_images_thresholded.append(log_image_thresholded)

    log_images_np = np.array(log_images)

    return log_images_np, kernels, np.array(log_images_thresholded), sigma_list
```

G Code for detecting the blobs in an image

```
def detect_blob_v2(log_images_np, k = np.sqrt(2), sigma = 1, percentage_high = 0.5):
    co_ordinates = []

    # create a list of coordinates that are sub-divided into sub-list ordered
        by sigma value in ascending order
    co_ordinates = [[] for i in range(log_images_np.shape[0])]

    for z in range(log_images_np.shape[0]):

        size = int(np.ceil(sigma * k**z)) * 5

        padd_y = img.shape[0]%size
        padd_x = img.shape[1]%size
```

```
        log_images_np_padded = np.pad(log_images_np[z], ((0,padd_x),(0,padd_y)),
            constant_values=0)

        (h,w) = log_images_np_padded.shape

        # calculate relative threshold for every sigma scale level
        threshold = max(log_images_np[z].flatten())-(percentage_high *
            max(log_images_np[z].flatten()))

        # split the image into chunks
        for i in range(int(size//2),int(h-size//2), size):
            for j in range(int(size//2), int(w-size//2), size):

                # search for the maximum in the chunk and assign the
                # coordinates to be the one of the maximum pixel
                slice_img = log_images_np[z,i-size//2:i+size//2,j-size//2:j+size//2]
                x_max, y_max = np.unravel_index(np.argmax(slice_img), slice_img.shape)

                # if the max is above some threshold append it to the list
                if slice_img[x_max, y_max] > threshold:
                    co_ordinates[z].append((i+x_max-size//2,
                        j+y_max-size//2,(k**z)*sigma))

    return co_ordinates
```

H Code for non maximum suppression

```
def non_max_suppression(coordinates, threshold = 1):

    print("before", len(coordinates))

    curr_coordinates = coordinates[::-1]

    tmp_lenght = len(curr_coordinates)

    i = 0

    while(i < tmp_lenght):
        j = i+1
        while j < tmp_lenght:

            x1, y1, r1 = curr_coordinates[i]
            x2, y2, r2 = curr_coordinates[j]
```

```
            distance = np.sqrt((x1-x2)**2 + (y1-y2)**2)

            if distance < threshold * (r1+r2):
                curr_coordinates.pop(j)
                tmp_lenght -= 1
                j -= 1

            j += 1

        i += 1

    print("after", len(curr_coordinates))

    return curr_coordinates[::-1]
```