ECE 558 Digital Imaging Systems
Project 3

Implementation of a Laplacian blob detector

Group Members:
1. Sumukha Manjunath (smanjun3)
2. Aditya Joshi (asjoshi6)
3. Akshay Balasubramaniyan (abalasu5)

December 11, 2022

Contents

List of Figures

Chapter 1

Introduction to Blob Detection

1.1 Objective

To implement the Laplacian Blob Detector and validate on set of images and to obtain regions of interest for further processing. These regions could signal the presence of objects or parts of objects in the image domain with application to object recognition and/or object tracking.

1.2 Introduction

Blob detection in image processing refers to modules that recognize points and/or areas in an image that differ in attributes such as brightness or color from the surrounding area. There are several reasons for researching and creating blob detectors. One major reason is to offer more information about regions that edge and corner detectors do not disclose. It is used to identify regions of interest for further analysis. These zones might indicate the existence of items or portions of things in the visual domain, which could be used for object detection and/or tracking.

Blob detection is often performed after color detection and noise reduction in order to locate the needed object in the image. A large number of irrelevant blobs of the needed color may be present in the image, and we also require blob features such as blob center, blob corners, and so on, for which we must know the precise coordinates of all the pixels in which the required blob is present.

Figure 1.1: Example of before Blob Detection

Figure 1.2: Example of after the blob detection

1.3 Outline of Algorithm

Below flowchart [Fig 1.3] represents the complete overview of the project step by step for better understanding.

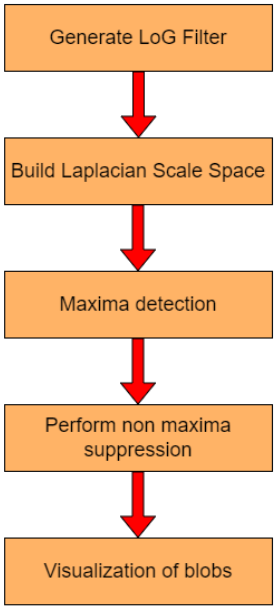


Figure 1.3: Flowchart of laplacian blob detector algorithm

1.3.1 Implementation of the Algorithm

The Laplacian of Gaussian (LoG) operation works in this way. You take an image and slightly blur it. Then you compute second order derivatives on it (or, the "laplacian"). This locates the image's boundaries and corners. These edges and corners are excellent for locating keypoints. The second order derivative, on the other hand, is particularly susceptible to noise. The blur reduces noise and stabilizes the second order derivative. The equation used to perform this operation is shown below in Figure 1.4.

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

Figure 1.4: Laplacian of Gaussian Equation

Scale-space representation is used to analyze measurement data at numerous scales, especially to enhance or suppress visual characteristics across different scale ranges. The Gaussian scale space provides a unique sort of scale-space representation, in which picture data in N dimensions is smoothed using Gaussian convolution. The majority of Gaussian scale space theory deals with continu-

ous pictures, however when implementing this theory, one must contend with the reality that most measurement data is discrete. As a result, the theoretical dilemma of how to discretize the continuous theory while keeping or well approximating the required theoretical qualities that led to the selection of the Gaussian kernel emerges. We are taking the scale space size and increasing it by 2 because the top and bottom most response will not be tested for maxima since there is not octave below that nor is there a octave above that. For building the Laplace scale, we are keeping the kernel fixed and changing the size of image. By doing this the relative size of kernel to image changes at each iteration, This relative size is maintained to what it would have been if we changed the kernel size. By doing this we are able to reduce the time of convolution.

Non Maximum Suppression is a computer vision algorithm that chooses a single entity from a group of overlapping entities (for example bounding boxes in object detection). Entities that fall below a certain probability threshold are often discarded. With the remaining entities, we choose the entity with the highest probability, output it as the prediction, and reject any remaining box with an IOU greater than or equal to any acceptable threshold with the box output in the preceding step. We apply both 2D and 3D Non-maximum suppression. Here we apply the non-maximum suppression at each scale space in 3d Laplacian response to remove redundancy of repeating coordinates, hence, the features.

After constructing the Difference of Gaussians, we may proceed to the fourth stage, which is to discover local maxima and minima in the DoG images. So now we'll find local minima and maxima for each pair of Difference of Gaussian pictures. Consider the pixel labelled X and its eight neighbors, as illustrated in Figure 1.5.

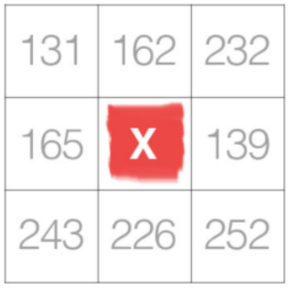


Figure 1.5: Maxima Detection

This pixel X is a "keypoint" if the pixel intensity value is more or less than the 26 checks made by the DoG keypoint detector, 8 on the current octave layer followed by 9 on the above and below octave layers respectively. This is shown in the Figure 1.6 below.

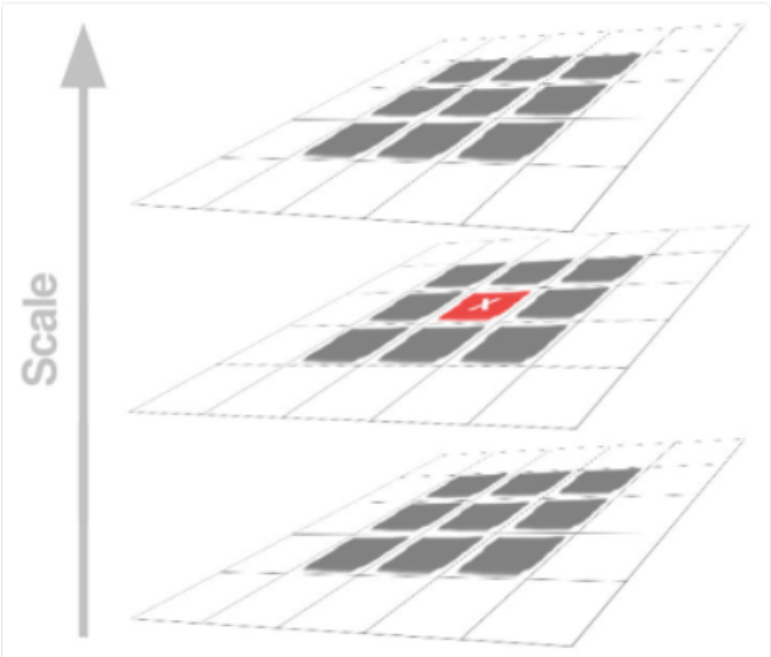


Figure 1.6: Octave Layers for comparsion

So far, 26 checks have been written. Again, a keypoint is defined as a pixel X that is larger or less than all 26 of its neighbors. Finally, we collect and label all pixels that are maxima and minima throughout all octaves as keypoints. Pruning is then used to remove keypoints with poor contrast.

The final stage of the blob detection is the visualization or displaying the resulting circles at their characteristic space. The radius of the circles to be drawn is determined by the sigma values we entered previously or by the matching scale. Once all of the maximum values have been located or described, the circle parameters are appended to a list that may be shown on the image.

1.4 Generating Laplacian of Gaussian filter

Initially we start by estimating the kernel size and location values. Based on these findings the Gaussian filter is generated and the Laplacian of Gaussian is computed using the standard formula.

```
def generate_log_apprximation(standard_deviation):
    """
    Computes the approximation of Laplacian of Gaussian using Difference of Gaussian given the standard deviation
    for the Gaussian kernel
    :param standard_deviation: The standard deviation for the Gaussian kernel/filter which is to be used for computed
                               the Difference of Gaussian
           type: float
    :return: 2D Laplacian of Gaussian kernel
    """
    # kernel_size = (standard_deviation * 4) + 1
    kernel_size = np.ceil(standard_deviation * 6)
    location_values = np.arange(int(-kernel_size//2), int(kernel_size//2 + 1))
    gaussian_1d = np.exp(-(location_values ** 2)/(2 * (standard_deviation ** 2)))
    log2d = (-1/(np.pi * (standard_deviation ** 4))) * (1 - (((location_values[np.newaxis, :] ** 2)
             + location_values[:, np.newaxis] ** 2)/(2 * (standard_deviation ** 2)))) *\
             (gaussian_1d[np.newaxis, :] * gaussian_1d[:, np.newaxis])
    return log2d
```

Figure 1.7: Code - Generation LoG filter

1.5 Filtering Image with Scale Normalized Laplacian at Current Scale

When the image is added, it is getting filtered by convolving with the scale normalized kernel. The 2D FFT or Fast Fourier Transform is implemented. To increase the accuracy or efficiency of the program, the input image is padded with zero padding prior to the convolution. The kernel is also getting padded.

```
def dft2d_fft_based(image):
    """
    Implementation of DFT2D using the 1D FFT using the separable property of DFT2D
    :param image: The image for which the DFT2D transformation needs to be computed
           type: ndarray
    :return: The frequency domain transformed image, spectrum and corresponding phase
    """
    image = np.asarray(image, np.complex)
    for i in range(image.shape[0]):
        image[i, :] = np.fft.fft(image[i, :])
    for i in range(image.shape[1]):
        image[:, i] = np.fft.fft(image[:, i])
    return image


def idft2d_fft_based(image):
    """
    Implementation of IDFT2D using the 1D FFT
    :param image: The image for which the DFT2D transformation needs to be computed
           type: complex ndarray
    :return: The frequency domain transformed image, spectrum and corresponding phase
    """
    for i in range(image.shape[0]):
        image[i, :] = np.array(1j, np.complex) * np.conjugate(image[i, :])
        image[i, :] = np.fft.fft(image[i, :])
        image[i, :] = np.array(1j, np.complex) * np.conjugate(image[i, :])
    image /= image.shape[0]
    for i in range(image.shape[1]):
        image[:, i] = np.array(1j, np.complex) * np.conjugate(image[:, i])
        image[:, i] = np.fft.fft(image[:, i])
        image[:, i] = np.array(1j, np.complex) * np.conjugate(image[:, i])
    image /= image.shape[1]
    return image
```

Figure 1.8: Code for DFT and IDFT

```python
def pad_kernel(kernel, target_size):
    """
    Zero Pads the given kernel to the target size
    :param kernel: A ndarray which needs to be padded
           type: ndarray
    :param target_size: Size to which the kernel needs to be padded
           type: tuple/list
    :return: Padded kernel (ndarray)
    """
    padded_kernel = np.zeros(target_size)
    x_position_1 = int(max(0, int((padded_kernel.shape[0] - kernel.shape[0]) / 2)))
    x_position_2 = int(min(target_size[0], ((padded_kernel.shape[0] + kernel.shape[0]) / 2)))
    y_position_1 = int(max(0, int((padded_kernel.shape[1] - kernel.shape[0]) / 2)))
    y_position_2 = int(min(target_size[1], ((padded_kernel.shape[1] + kernel.shape[0]) / 2)))
    padded_kernel[x_position_1 : x_position_2, y_position_1: y_position_2] = kernel
    return padded_kernel
```

Figure 1.9: Code for padding

The resultant after performing DTF2 multiplication, IDFT or Inverse Fast Fourier transform is implemented. The filtered image should have the same size as the original image.

1.6 Building Laplacian Scale Space

For the given image, the Laplacian of Gaussian stack or scale space is constructed using the *generate log stack* function. For the given parameters, by using the functions specified in section 1.4 and section 1.5, the scale space is created.

```python
def generate_log_stack(img, number_of_iterations, init_sigma, k):
    """
    Computes the scale space/LoG stack for the given image
    :param img: The image for which the LoG stack needs to be computed
          type: ndarray
    :param number_of_iterations: The number of levels in the LoG space
           type: int
    :param init_sigma: The standard deviation for the first level of the LoG
           type: float
    :param k: The factor by which the standard deviation is scaled in consequent levels in LoG scale space
              (sigma, k*sigma, k^2*(sigma), ..)
           type: float
    :return: Stack of LoG
    """
    stacked_logs = []
    img = img / 255
    for i in range(number_of_iterations):
        current_sigma = init_sigma * (k ** i)
        log_approx = generate_log_apprximation(current_sigma) * (current_sigma ** 2)
        padded_log_approx = pad_kernel(log_approx, img.shape)
        padded_log_approx = np.fft.ifftshift(padded_log_approx)
        img_freq = dft2d_fft_based(img)
        padded_log_approx_freq = dft2d_fft_based(padded_log_approx)
        filtered_freq = np.multiply(padded_log_approx_freq, img_freq)
        filtered = idft2d_fft_based(filtered_freq)
        filtered = np.abs(filtered)
        filtered = np.square(filtered)
        stacked_logs.append(filtered)
    return stacked_logs
```

Figure 1.10: Code for attaining the initial keypoints

1.7 Finding the initial keypoints

```python
def find_initial_keypoints(log_stack, threshold):
    """
    Computes the initial key points from the stacked LoG by comparing each value with its 3D neighborhood,
    picking the maximum value and checking if its value is greater than the given threshold
    :param log_stack: The stack of LoG responses of the image
           type: ndarray
    :param threshold: The threshold used for selecting maximas in the LoG
           type: float
    :return: A binary mask with the detected keypoints/maximas being 1
    """
    number_of_logs, height, width = log_stack.shape
    maxima_mask = np.zeros_like(log_stack)
    for i in range(1, number_of_logs - 1):
        for j in range(1, height - 1):
            for k in range(1, width - 1):
                comparison_cube = log_stack[i - 1: i + 2, j - 1: j + 2, k - 1: k + 2]
                cube_max_value_index = np.unravel_index(np.argmax(comparison_cube), comparison_cube.shape)
                if comparison_cube[cube_max_value_index] > threshold:
                    maxima_mask[i - 1 + cube_max_value_index[0], j - 1 + cube_max_value_index[1], k - 1 + cube_max_value_index[2]] = 1
    return maxima_mask
```

Figure 1.11: Code for attaining the initial keypoints

The Gaussian Laplacian functions as a blob detector, detecting blobs of varying sizes as the sigma values change. Sigma functions as a scaling factor. Gaussian kernels with low sigma provide a high value for tiny corners, but high sigma values match well with bigger corners. As a result, we locate the local maxima throughout scale and space and identify a probable keypoint. Another way of finding the accurate keypoints is by performing Difference of Gaussian.The Difference of Gaussian blurring of a picture with two distinct sigma values is calculated. In Gaussian Pyramid, this technique is repeated for each of the image's octaves. After locating this DoG, images are examined for local extrema throughout size and space. For each response in the scale space the maximas detected by comparing the Laplacian of Gaussian response of each point with its 3D neighbourhood (3 x 3 x 3 space). If the maximum value in the neighbourhood is greater than a heuristically determined threshold, the point is considered as a key point.

1.8 Non-Max Suppression

The obtained initial keypoints are subjected to non maxima suppression. Here the NMS is carried only for the initial key points obtained which results in

saving of computational time and power. Initially the local neighbourhood is used to identify the 3*3 region around the initial key points. The inbuilt *numpy* function called *np.max* is used to search in neighbour hood for maximum value. The detection NMS is performed in 2D i.e with respect to image itself and then consequently in 3D i.e w.r.t layers or stack. And only those points where the values match in the 2d and 3d neighbourhoods are maintained as the maximas and the rest are set to 0.

```
def non_maxima_suppression(initial_maxima_mask, log_stack):
    """
    Performs Non Maxima suppression in 2D neighbourhood of each maxima and then in the 3D neighborhood of each maxima
    and maintains only the key points which are a maxima in both the 2D and 3D neighborhoods
    :param initial_maxima_mask: A binary mask indicating the initially computed maxima values
           type: ndarray
    :param log_stack: Stack of the LoG responses
           type: ndarray
    :return: A binary mask indicating the filtered key points/maximas
    """
    number_of_scales = initial_maxima_mask.shape[0]
    local_maximas_2d = np.zeros_like(log_stack)
    for i in range(number_of_scales):
        maxima_locations = np.where(initial_maxima_mask[i] == 1)
        for x_i, y_i in zip(maxima_locations[0], maxima_locations[1]):
            local_neighborhood = log_stack[i, max(0, x_i - 1): x_i + 2, max(y_i - 1, 0): y_i + 2]
            local_maximas_2d[i, x_i, y_i] = np.max(local_neighborhood)

    local_maxima_3d = np.zeros_like(log_stack)
    for i in range(1, number_of_scales - 1):
        maxima_locations = np.where(initial_maxima_mask[i] == 1)
        for x_i, y_i in zip(maxima_locations[0], maxima_locations[1]):
            local_neighborhood_3d = local_maximas_2d[i - 1: i + 2, max(0, x_i - 1): x_i + 2, max(0, y_i - 1): y_i + 2]
            local_maxima_3d[i, x_i, y_i] = np.max(local_neighborhood_3d)

    final_maxima_locations = (local_maxima_3d == local_maximas_2d) * initial_maxima_mask
    return final_maxima_locations
```

Figure 1.12: Code for Non-Max Suppression

1.9 Visualization of Output

The visualization of blobs (maxima locations) after NMS is done by using cv2.circle function from inbuilt directory. The radius of blob is calculted as $r = \sigma(K^z)$ where z stands for the layer number. The pixel values are converted to (0,255) level from the normalized form.

```
def run_blob_detection(img_path, sigma=1.5, save_folder=None, number_of_iterations=8, threshold=0.02, k=1.5, viz=True):
    """
    Detects blobs for the image sorted at img_path with respect to the other parameters provided
    :param img_path: The path to the image for which the blob detection needs to be performed
           type: str
    :param sigma: The standard deviation for the first level in the scale space
           type: str
           default: 1.5
    :param save_folder: The path to the folder where the resulting visualizations need to be saved
           type: str
           default: None
    :param number_of_iterations: The number of scales/levels of the laplacian of Gaussian space
           type: int
           default: 8
    :param threshold: The threshold used to select the maxima values from the LoG response in a neighborhood fashion
           type: float
           default: 0.02
    :param k: The factor by which the standard deviation is scaled in consequent levels in LoG scale space
              (sigma, k*sigma, k^2*(sigma), ..)
           type: float
           default: 1.5
    :param viz: If the results need to be visualized from the LoG scale space, maxima detection and non maxima
                suppressed steps.
           type: bool
           default: True
    """
    img = cv2.imread(img_path, 0)
    stacked_logs = generate_log_stack(img, number_of_iterations, sigma, k)
    if viz:
        for i, log in enumerate(stacked_logs):
            plt.imshow(log, cmap="gray")
            plt.savefig(os.path.join(save_folder, "scale_%d_log.png" % i))
    initial_keypoints_mask = find_initial_keypoints(np.asarray(stacked_logs), threshold)
    initial_keypoints_locations = np.where(initial_keypoints_mask == 1)
    if viz:
        img_rgb = cv2.imread(img_path)
        for z, x, y in zip(initial_keypoints_locations[0], initial_keypoints_locations[1], initial_keypoints_locations[2]):
            radius = sigma * (K ** z)
            cv2.circle(img_rgb, (y, x), int(radius), (0, 0, 255), 1)
        cv2.imwrite(os.path.join(save_folder, "key_points_initial.png"), img_rgb)
    if viz:
        img_rgb = cv2.imread(img_path)
        filtered_keypoints = non_maxima_suppression(initial_keypoints_mask, np.asarray(stacked_logs))
        maxima_locations = np.where(filtered_keypoints == 1)
        for z, x, y in zip(maxima_locations[0], maxima_locations[1], maxima_locations[2]):
            radius = sigma * (K ** z)
            cv2.circle(img_rgb, (y, x), int(radius), (0, 0, 255), 1)
        cv2.imwrite(os.path.join(save_folder, "key_points.png"), img_rgb)
```

Figure 1.13: Code for Visualization

```python
if __name__ == "__main1__":
    """
    Runtime Analysis
    """
    image_folder = "../test_images/"
    image_paths = list(os.listdir(image_folder))
    save_folder_path = "../results"
    sigma_val = [2, 1, 2, 2]
    number_of_iter = [8, 8, 8, 8]
    maxima_threshold = [0.015, 0.02, 0.02, 0.025]
    K = 1.5
    visualize_results = True
    total_time = 0
    for i, image_path in enumerate(image_paths):
        image_name = image_path.split(".")[0]
        save_folder_path_i = os.path.join(save_folder_path, image_name)
        if not os.path.exists(save_folder_path_i):
            os.mkdir(save_folder_path_i)
        start = time.time()
        run_blob_detection(os.path.join(image_folder, image_path), sigma_val[i], save_folder_path_i, number_of_iter[i],
                           maxima_threshold[i], K, visualize_results)
        total_time += (time.time() - start)
    print("Time taken for blob detection: ", total_time / len(image_paths))


if __name__ == "__main2__":
    """
    Blob detection for single image
    """
    image_path = "../test_images/butterfly.jpg"
    save_folder_path = "../results/butterfly"
    if not os.path.exists(save_folder_path):
        os.mkdir(save_folder_path)
    sigma_val = 2
    number_of_iter = 8
    maxima_threshold = 0.01
    K = 1.5
    visualize_results = True
    start = time.time()
    run_blob_detection(image_path, sigma_val, save_folder_path, number_of_iter,
                       maxima_threshold, K, visualize_results)
    print("Time taken for blob detection: ", (time.time() - start))
```

Figure 1.14: Code for Visualization

Chapter 2

Results

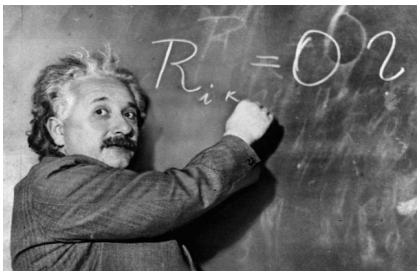
In this section the results obtained by implementing the code are summarized. Here the output will look different depending upon the threshold, sigma, no of iterations and K. The experimentation was done as follows
1. Variable threshold on butterfly image
2. Variable sigma on butterfly image
3. Variable no of iterations on butterfly image
4. Variable K on butterfly image
5. Same parameter's for different images
6. Best parameter's obtained for different images
Note :- As the selection of parameters is heuristics the results may be much more better than obtained in 6.

2.1 Test Images Used

We used in total eight images for testing as shown in below Fig 2.1:

- Butterfly Image
- Einstein Image
- Fishes Image
- Sunflowers Image
- Mr Wufs image
- Actress Image
- APJ Abdul Kalam Image
- Ronaldo Image

(a) Butterfly Image



(b) Einstein Image

(c) Fishes Image

(d) Sunflowers image

(e) Wuf's Image

(f) Actress Image

(g) APJ Abdul Kalam Image

(h) Ronaldo Image

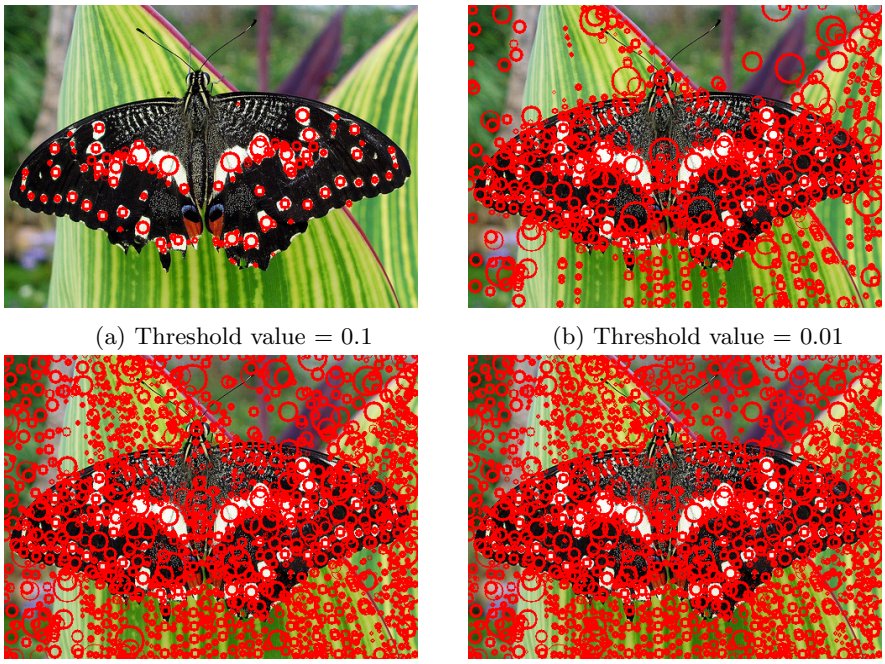
Figure 2.1: Test Images Used

2.2 Results on Test Images

2.2.1 Variable Threshold

Here the results for butterfly image [Fig 2.2] are obtained with below hyperparameter setting

- $\sigma = 1.5$
- no of iterations = 8
- threshold = 0.1, 0.01, 0.001, 0.0001
- K = 1.5



(a) Threshold value = 0.1

(b) Threshold value = 0.01

(c) Threshold value = 0.001

(d) Threshold value = 0.0001

Figure 2.2: Results for different threshold values on butterfly image

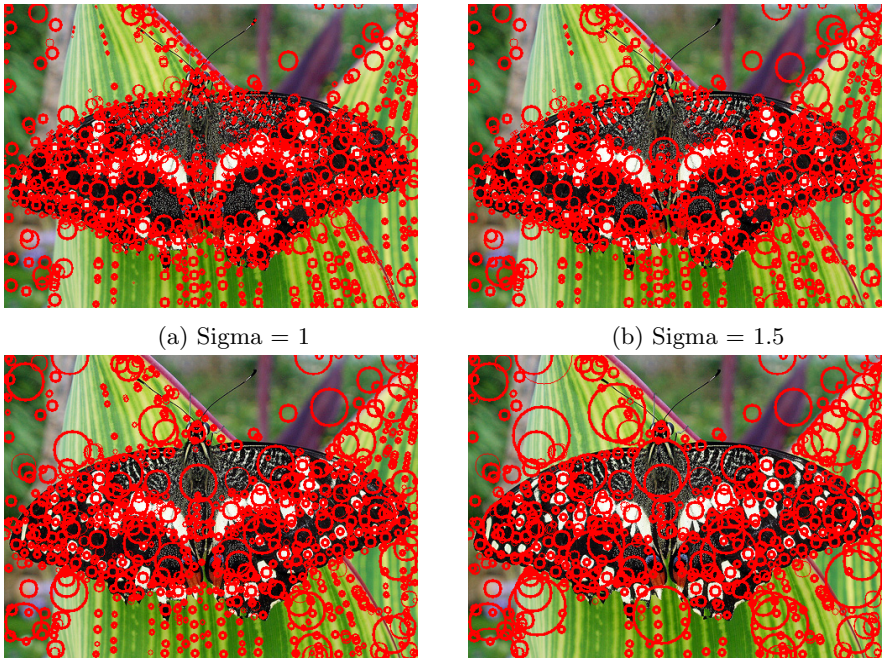
The table summarizing the runtime for each of thresholding values is as follows :

Threshold Value	Run Time (in sec)
0.1	8.487
0.01	12.555
0.001	17.746
0.0001	18.720

2.2.2 Variable sigma

Here the results for butterfly image [Fig 2.3] are obtained with below hyper parameter setting

- σ = 1, 1.5, 2, 3
- no of iterations = 8
- threshold = 0.01
- K = 1.5



(a) Sigma = 1

(b) Sigma = 1.5

(c) Sigma = 2

(d) Sigma = 3

Figure 2.3: Results for different sigma values on butterfly image

The table summarizing the runtime for each of sigma values is as follows :

Sigma Value	Run Time (in sec)
1	11.856
1.5	12.233
2	12.463
3	12.914

2.2.3 Variable no of iterations

Here the results for butterfly image [Fig 2.4] are obtained with below hyperparameter setting

- $\sigma = 1.5$
- no of iterations = 4, 6, 8, 10
- threshold = 0.01
- K = 1.5

(a) No of iterations = 4

(b) No of iterations = 6

(c) No of iterations = 8

(d) No of iterations = 10

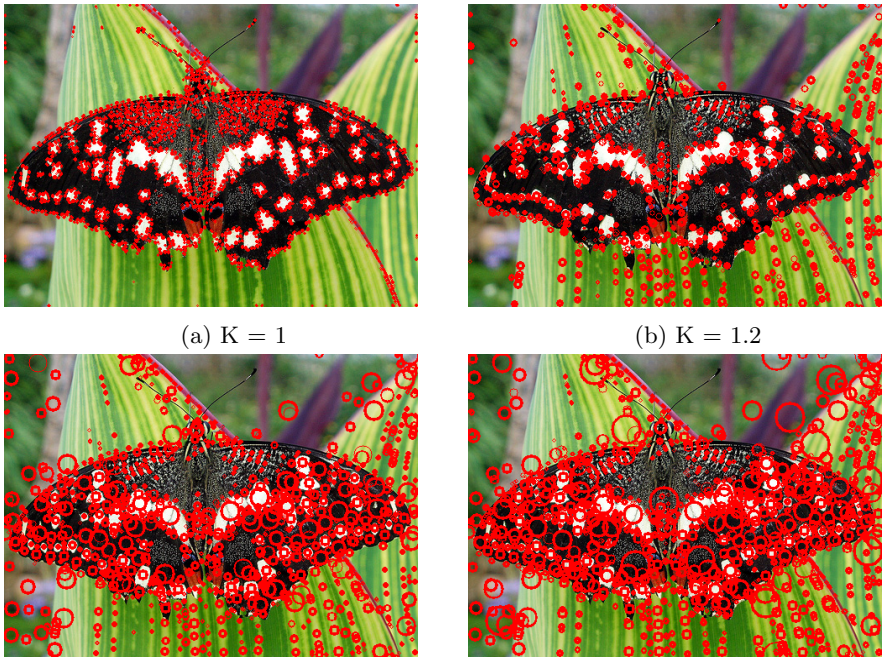
Figure 2.4: Results for different no of iterations values on butterfly image

No of iterations	Run Time (in sec)
4	4.271
6	7.812
8	12.113
10	16.538

2.2.4 Variable K

Here the results for butterfly image [Fig 2.5] are obtained with below hyperparameter setting

- $\sigma = 1.5$
- no of iterations = 8
- threshold = 0.01
- K = 1, 1.2, 1.41, 1.5



(a) K = 1 (b) K = 1.2

(c) K = 1.41 (d) K = 1.5

Figure 2.5: Results for different K values on butterfly image

The table summarizing the runtime for each of K values is as follows :

K Value	Run Time (in sec)
1	9.787
1.2	11.099
1.41	11.760
1.5	11.889

2.2.5 Same parameter's for different images

Here the results are obtained for below hyper parameter setting on remaining images [Fig 2.6] Note : SP stands for same parameters

- $\sigma = 1.5$
- no of iterations = 8
- threshold = 0.01
- K = 1.5

The runtime of image depends upon the size of image. This is the reason that each image has variation in run time. The table summarizing the runtime for each of image is as follows :

Image	Run Time (in sec)
Butterfly	12.555
Einstein	15.671
Fishes	9.518
Sunflowers	7.546
Wuf's	8.716
Actress	12.733
APJ Abdul Kalam	38.091
Ronaldo	12.863

2.2.6 Best parameter's obtained for different images

As from the previous results obtained we can analyze that selection of parameters is heuristic or parametric. So with best possible combinations of parameters below results are obtained as shown in Fig [2.7].
The parameter's for butterfly image [Fig 2.7a]

- $\sigma = 1.5$
- no of iterations = 8
- threshold = 0.01
- K = 1.5

The parameter's for Einstein image [Fig 2.7b]

- $\sigma = 1$
- no of iterations = 8
- threshold = 0.02
- K = 1.5

The parameter's for sunflowers image [Fig 2.7c]

- $\sigma = 2$
- no of iterations = 8
- threshold = 0.025
- K = 1.5

The parameter's for fishes image [Fig 2.7d]

- $\sigma = 2$
- no of iterations = 8
- threshold = 0.02
- K = 1.5

The parameter's for Mr Wuf's image [Fig 2.7e]

- $\sigma = 1.5$
- no of iterations = 8
- threshold = 0.02
- K = 1.41

The parameter's for actress image [Fig 2.7f]

- $\sigma = 1.5$
- no of iterations = 8
- threshold = 0.01
- K = 1.5

The parameter's for APJ Abdul Kalam image [Fig 2.7g]

- $\sigma = 2$
- no of iterations = 6
- threshold = 0.01
- K = 1.5

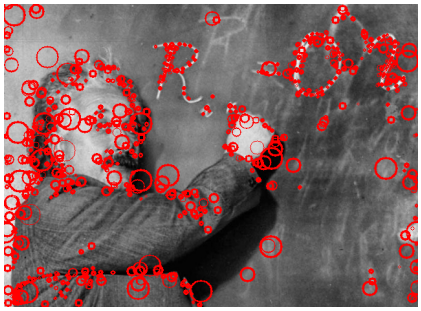
The parameter's for Ronaldo image [Fig 2.7h]

- $\sigma = 1.5$
- no of iterations = 8
- threshold = 0.01
- K = 1.5

The table summarizing the runtime for each of image is as follows :

Image	Run Time (in sec)
Butterfly	12.555
Einstein	13.880
Fishes	8.385
Sunflowers	6.073
Wuf's	7.303
Actress	11.809
APJ Abdul Kalam	24.842
Ronaldo	8.716

(a) Butterfly Image result(SP)



(b) Einstein Image result(SP)

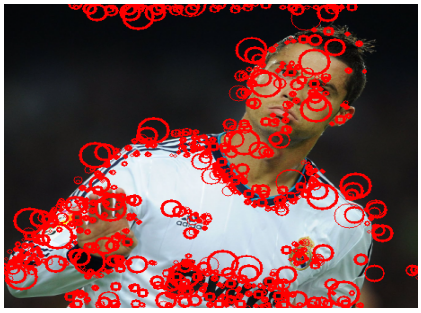
(c) Fishes Image result(SP)

(d) Sunflowers image result(SP)

(e) Mr.Wufs Image result(SP)

(f) Actress Image result(SP)

(g) APJ Abdul Kalam Image result(SP)

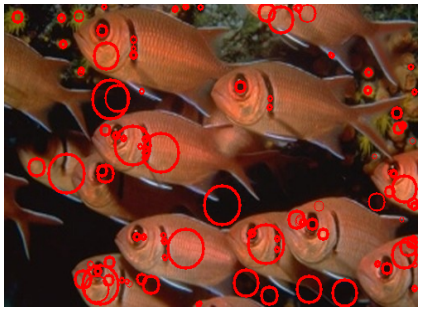


(h) Ronaldo Image result(SP)

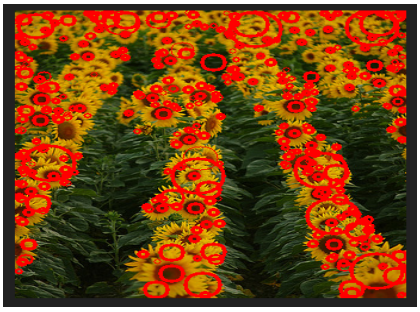
Figure 2.6: Result for same parameter's on different images

(a) Butterfly Image result

(b) Einstein Image result



(c) Fishes Image result



(d) Sunflowers image result

(e) Wuf Image result

(f) Actress Image result

(g) APJ Abdul Image result

(h) Ronaldo Image result

Figure 2.7: Result for best parameters on different images

Chapter 3

Conclusion and Future Scope

3.1 Conclusion

- In total eight images (four - instructor provided and four- random web searched) were used to validate the Laplacian blob detector algorithm. The no of blobs detected in a given image are dependent upon four hyper-parameters i.e threshold value, iterations, K and sigma. The empirical approach (trial and error) is required to be used for tuning of hyper-parameters. Smaller the threshold value longer the run time.
- The approach stands unique in terms of implementation as compared to conventional since the maxima's (key-points) are detected and then the Non maxima suppression (NMS) is implemented. This lead to decrease in computational time and power since in this method the NMS is performed only in the neighbourhood of maxima instead of executing at each pixel level (as in case of traditional approach).

3.2 Future Scope

Scale space computation can be implemented using two 1-D convolution to increase the computational speed. The current approach to blob detection is heuristic and empirical based in terms of tuning of hyper-parameters. In the course of time it would be interesting to discuss and focus on generalized framework for obtaining confidence bounds for hyper-parameters. Once prospective keypoint locations are identified, they can be improved to get more precise results. Because Difference of Gaussian has a stronger reaction to edges, edges must also be eliminated. A concept similar to the Harris corner detector can be employed for this. The principal curvature was calculated using a 2x2 Hessian matrix (H). We know from the Harris corner detector that one eigenvalue

is greater than the other for edges. So we can utilize a simple function here. Based on this if the ratio is greater than the threshold then it is rejected or discarded. In the current implementation we are first performing NMS in 2d space by considering a fixed 3x3 neighbourhood for initial keypoints, in the future we want to explore considering the neighborhood based on the scale or radius of initial detected keypoints.This way it reduces the overlapping blobs which are in closed proximity but which might not originally be in the 3x3 neighbourhood.

Chapter 4

References

4.1 List of References

[1] Gonzalez and Woods, Digital Image Processing (3rd Edition). Prentice Hall, August 2007.

[2] Lowe, D. "Distinctive image features from scale-invariant keypoints" International Journal of Computer Vision, 60, 2 (2004), pp. 91-110

[3] Course work slides ECE 558 Digital Imaging System

[4] https://medium.com/analytics-vidhya/a-beginners-guide-to-computer-vision-part-2-edge-detection-4f10777d5483 (For Laplacian of Gaussian Formula)

[5]J.R Parker, Algorithm for image processing and computer vision, 2nd Edition

4.2 Appendix

Contribution by each of teammates.
Each of team member had individually as well as in group went through required literature and know about algorithm. A good coordinated efforts were contributed by each of member and having group meetings helped to exchange and refine the ideas for the project.

Unity Id	Associated task
smanjun3	Maxima computation and suppression+ Report + Results
asjoshi6	Scale Space computation - experimenting with spatial and frequency domain computations + Report + Results
abalasu5	LoG generation + parameter tuning for different examples + Report + Results