ECE 558

Project 03 – Final

Laplacian Blob Detector

Group Members :

Hritwik Shukla (*hshukla*)

Sushant Kolhe (*shkolhe*)

Aaron Mathew (*asmathew*)

In the area of computer vision, blob detection refers to visual modules that are aimed at detecting points and/or regions in the image that differ in properties like brightness or color compared to the surrounding. A blob is a region of an image in which some properties are constant or approximately constant.

In this project we aim to create a Laplacian Blob Detector to detect features from images which are invariant to affine transformations. Also since we are trying to find a tradeoff between the speed of the algorithm and the accuracy. Towards this we choose to use the approximation of the compute extensive Laplacian of Gaussian (LoG) kernel which is the Difference of Gaussian (DoG) kernel. This approximation is shown by the image (a) LoG vs DoG.

$$L = \sigma^2 \left(G_{xx}(x, y, \sigma) + G_{yy}(x, y, \sigma) \right)$$

(Laplacian)

$$DoG = G(x, y, k\sigma) - G(x, y, \sigma)$$

(Difference of Gaussians)

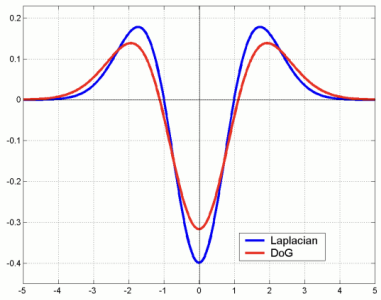
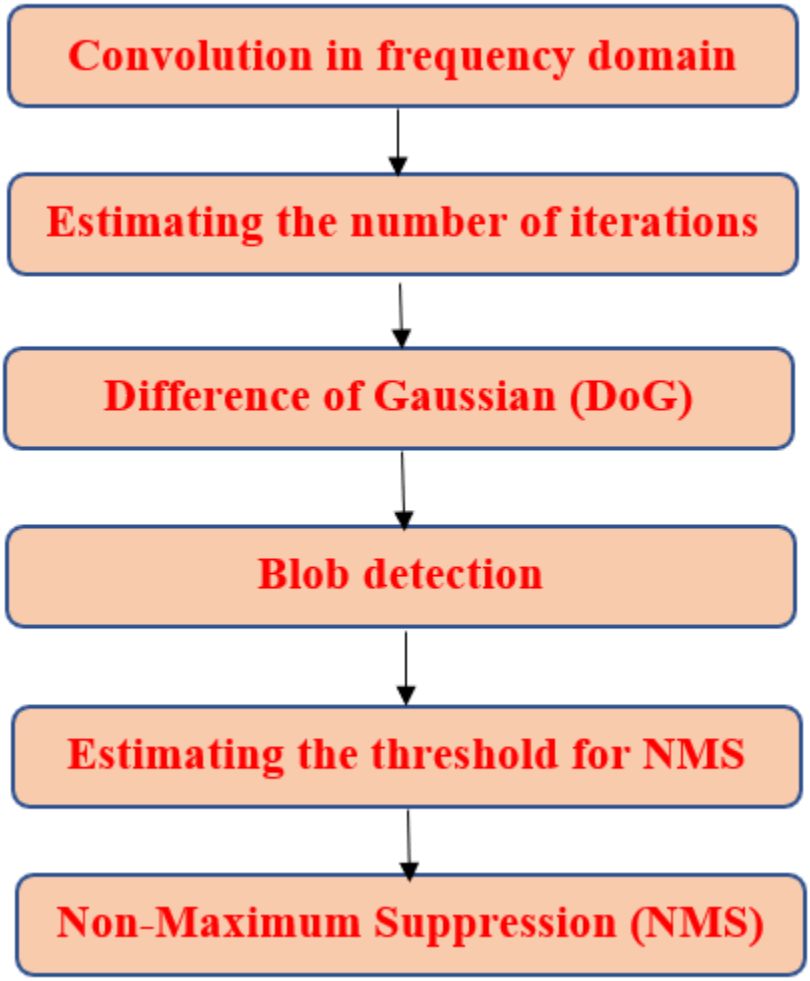


Image (a). Approximation of the Laplacian of Gaussian (LoG) kernel by using the Difference of Gaussian (DoG) kernel.

Algorithm:



Steps Outline:

Step 0: Library requirements and debug functions for the project

- For this project we require the following python libraries:
    1. numpy
    2. cv2
    3. matplotlib
    4. pylab
    5. scipy

```
import math
import time
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
from pylab import *
from scipy import ndimage
from scipy.ndimage import filters
from scipy import spatial
```

- Display convolution images to visualize the scale-space

```
#Display images
def display(convolved_images):
    for h in range(len(convolved_images)):
        plt.imshow(convolved_images[h],cmap='gray')
        plt.show()
```

Step 1: Difference of Gaussian (DoG)

The function takes the gaussian smoothened octave. The adjacent images are subtracted to obtain the final DoG octave.

```
def D_o_G(gaussian_smoothened_pyramid):
    difference_of_gaussian_images = []
    length_of_pyramid = len(gaussian_smoothened_pyramid)
    for index_in_dog in range(length_of_pyramid-1):
        difference_between_2_layers =
np.abs(np.subtract(gaussian_smoothened_pyramid[index_in_dog],gaussian_smoothened_pyramid[index_in_dog+1
]))
        difference_of_gaussian_images.append(difference_between_2_layers)
    return difference_of_gaussian_images
```

Step 2: Convolution

In convolution, we first pad the kernel to make it the same size as the reference image.
Then we perform FFT along the X and Y axis, for the kernel and the image.
The next step is to center shift the image and then multiply them together.
After that we perform IFFT to get the original image back.

```python
#DFT2 function defination
def DFT2(f):
    sz1 = f.shape
    im_eq = f
    img = im_eq
    f_1 = np.zeros((sz1[0],sz1[1]), dtype="complex_")
    f_2 = np.zeros((sz1[0],sz1[1]), dtype="complex_")
    for i in range(sz1[0]):
        f_1[i,:] = np.fft.fft(img[i,:])
    for j in range(sz1[1]):
        f_2[:,j] = np.fft.fft(f_1[:,j])
    F = f_2
    return F

#IDFT2 function defination
def IDFT2(product):
    sz_F = product.shape
    #Again re-shift the origin to top left corner of the image
    corner_shifted_product = np.fft.ifftshift(product)
    f_3 = np.zeros((sz_F[0],sz_F[1]), dtype="complex_")
    f_4 = np.zeros((sz_F[0],sz_F[1]), dtype="complex_")
    #Finding 2D inverse of F using 1D inbuilt function
    for i in range(sz_F[1]):
        f_3[:,i] = np.fft.ifft(corner_shifted_product[:,i])
    for j in range(sz_F[0]):
        f_4[j,:] = np.fft.ifft(f_3[j,:])
    g = f_4
    return g

def conv2(f,kernal):

    #Calling the DFT2 function for input image and passing normalized image
    F_image = DFT2(f/255)
    #Spectrum Shifted to centre
    centre_shifted_dft2_img = np.fft.fftshift(F_image)
    kernal_original = kernal

    #Padding done on kernal to make size of kernaL equal to size of input image
    kernal_resized_to_input_image = np.zeros((img_original.shape[0],img_original.shape[1]))

    rows_input_image = f.shape[0]
    cols_input_image = f.shape[1]

    row_lower_limit = int(np.floor((rows_input_image-kernal.shape[0])/2))
    row_upper_limit = row_lower_limit+kernal.shape[0]

    col_lower_limit = int(np.floor((cols_input_image-kernal.shape[1])/2))
    col_upper_limit = col_lower_limit + kernal.shape[1]

    kernal_resized_to_input_image[row_lower_limit:row_upper_limit,col_lower_limit: col_upper_limit] =
kernal

    #Calling the DFT2 function for kernal
    F_kernal = DFT2(kernal_resized_to_input_image)
    #Spectrum shifted to centre
    centre_shifted_dft2_kernal = np.fft.fftshift(F_kernal)

    #Convolution in frequency domain
    Product_img = F_image*F_kernal

    #Inverse fourier transform of product to get convoluted image
    inverse_fft_product = IDFT2(Product_img)

    #Centre shifted prodcut of convolution in frequency domain
    centre_shift_inverse_transpose_product = np.fft.fftshift(inverse_fft_product)
    return np.abs(centre_shift_inverse_transpose_product.real)
```

Step 3: Estimating the number of iters and octave:

```python
def get_kernal(size,sigma):
    ker = cv.getGaussianKernel(int(size), sigma)
    ker = np.outer(ker,ker)
    return ker

def get_kernal_octave(sigma_initial,k,size_of_octave,scale_for_sigma_to_kernel_size):
    # we create an empty array to store various sigma values
    gaussian_kernal_with_diff_sigma = []
    sigma_values = []
    for i in range (size_of_octave):
        sigma_next = (k**i)*sigma_initial
        sigma_values.append(sigma_next)
    for kernal_index_in_octave in range(size_of_octave):
        kernel_size = np.ceil(scale_for_sigma_to_kernel_size*sigma_values[kernal_index_in_octave])
        kernal = get_kernal(kernel_size,sigma_values[kernal_index_in_octave])
        gaussian_kernal_with_diff_sigma.append(kernal)
    return gaussian_kernal_with_diff_sigma
```

Step 4: detect_blob()

The octaves of the size 9x3x3 are iterated at the same time and the maximum value is from those masks is obtained.

This max value is compared to the threshold value, if its greater then the coordinates are saved in an array.

```python
def detect_blob(dog_images, sigma, k):
    #to store co ordinates
    co_ordinates = []
    (h,w) = dog_images[0].shape
    a = 0
    #Set the ranges of i and j accordingly to make the shape of sliced matrix equal to 3x3,5x5,7x7, etc
    for i in range(3,h-3):
        for j in range(3,w-3):
            #7*7*10 slice
            sliced_matrix_in_octave = dog_images[:,i-3:i+4,j-3:j+4]
            max_value_in_sliced_matrix = np.amax(sliced_matrix_in_octave)
            #Set Threshold Value to detect any peak in the image and ignore the rest
            if max_value_in_sliced_matrix >= 0.009:
                z,x,y=np.unravel_index(sliced_matrix_in_octave.argmax(),sliced_matrix_in_octave.shape)
                #finding co-ordinates - x, y, radius
                co_ordinates.append((i+x-1,j+y-1,k**(z)*sigma))
            else:
                a=1
    return co_ordinates
```

Step 5: Non_max_suppression

```
def non_max_suppression(blobs_array, overlap_threshold):
    print("\nWorking on Non-Maximum Suppression...Please wait !!")
    blobs_array_after_NMS = []
    max_radius_of_blob = blobs_array[:, -1].max()
    distance = sqrt(2) * max_radius_of_blob
    tree = spatial.cKDTree(blobs_array [:, :-1])
    #Make pair of keypoints whose distance from each other is at max equal to 'distance'
    pairs = np.array(list(tree.query_pairs(distance)))
    if len(pairs) == 0:
        return blobs_array
    else:
        for (i, j) in pairs:
            blob1, blob2 = blobs_array[i], blobs_array[j]
            if overlap_ratio(blob1, blob2) > overlap_threshold:
                if blob1[-1] > blob2[-1]:
                    blob2[-1] = 0
                else:
                    blob1[-1] = 0
    for blobs in blobs_array:
        if blobs[-1]>1:
            blobs_array_after_NMS.append(blobs)
    blobs_array_after_NMS = np.asarray(blobs_array_after_NMS)
    return blobs_array_after_NMS
```

The purpose of this function is to implement non-maximum suppression where we pass an array of blobs as the first argument (blobs_array) that has the coordinates and radius of the blobs. The second argument is the overlapping threshold to check and eliminate the blobs which are overlapping with each other with an overlapping ratio greater than a particular threshold. Overlapping ratio is calculated in the overlap_ratio() function. We have to find pairs of blobs to check and eliminate the blobs whose overlapping ratio is greater than the threshold value. Now we found out the pair of blobs are at a maximum distance of [sqrt(2)*max_radius_of_blob]. This way we can eliminate many blobs which gives us refined output.

Step 6: overlap_ratio()

```
#Function that returns the overlap ratio between two blobs
def overlap_ratio(blob1,blob2):
    radius_blob1 = blob1[-1]
    radius_blob2 = blob2[-1]
    distance_between_blobs = sqrt(((blob2[0]-blob1[0])**2)+((blob2[1]-blob1[1])**2))
    if (distance_between_blobs > (radius_blob1 + radius_blob2)):
        return 0
    elif distance_between_blobs <= abs(radius_blob1 - radius_blob2):
        return 1
    else:
        d1 = np.abs((((radius_blob1)**2)-((radius_blob2)**2)+
((distance_between_blobs)**2))/(2*distance_between_blobs))
        d2 = np.abs((((radius_blob2)**2)-((radius_blob1)**2)+
((distance_between_blobs)**2))/(2*distance_between_blobs))

        theta_1 = 2* (math.acos ((round((d1/radius_blob1),4))))
        theta_2 = 2* (math.acos ((round((d2/radius_blob2),4))))

        base_of_triangle = 2*radius_blob1*sin(theta_1/2)

        Area_S1 = (theta_1/(2*np.pi))*(np.pi*(radius_blob1**2))-((1/2)*(base_of_triangle)*(d1))
        Area_S2 = (theta_2/(2*np.pi))*(np.pi*(radius_blob2**2))-((1/2)*(base_of_triangle)*(d2))
        Area_of_overlap = Area_S1 + Area_S2

        ratio_of_overlap = Area_of_overlap/(np.pi*((min(radius_blob1,radius_blob2))**2))
        return ratio_of_overlap
```

Here, we get 2 overlapping blobs as an input. Blob1 and blob2 are the array containing x-coordinate, y-coordinate and radius of the blob. No we calculate the overlapping area of both the circular blobs and divide it by the area of the smaller blob. By doing this, we get the overlapping ratio.

Step 7: display_images()

```
#Display images
def display(convolved_images):
    for h in range(len(convolved_images)):
        plt.imshow(convolved_images[h],cmap='gray')
        plt.show()
```

This function is used to visualize different scale space images

Hyper-parameters:

1. Initial sigma value for generating gaussian kernel [variable name : 'sigma_initial']
2. Threshold value for blob detection [variable name : 'max_value_in_sliced_matrix']
3. Distance value to generate pair of blobs in Non-Maximum Suppression function [variable name : 'distance']
4. Size of the sliced matrix [in the detect_blob() function] from which we extract the maximum intensity value to compare with a threshold value for blob detection.
5. Scale Factor for calculating the radius of the blob. [variable name : 'k']
6. Threshold value to check overlap of blobs used in Non-maximum Suppression. [variable name : 'overlap_threshold']
7. Scale parameter to generate kernels with different dimensions in order to compensate for different sigma values [variable name : 'scale_for_sigma_to_kernel_size']
8. Number of smoothened images in octave [variable name = 'size_of_octave']

In order to validate our code as per the problem statement mentioned, we choose the following **performance parameters** on which the effectiveness of the code depends:

1. Effectively identify blobs in the image.
2. Number of blobs detected before applying non-maximum suppression.
3. Number of blobs left after applying non-maximum suppression.
4. Time elapsed in running non-maximum suppression.
5. Time elapsed in plotting the blobs.

Relationship between the hyper-parameters and performance parameters:
As the Size of the sliced matrix [in the detect_blob() function] decreases, the number of blobs that are detected increases rapidly, and hence the time taken to execute the code increases (cost of computation is high)

As the 'overlap_threshold' decreases, the number of blobs decreases and thus the time taken to execute the code decreases.

As the distance value in the non-maximum suppression function increases, the number of blobs detected increases, and therefore the time to execute the code increases. So the cost of computation increases.

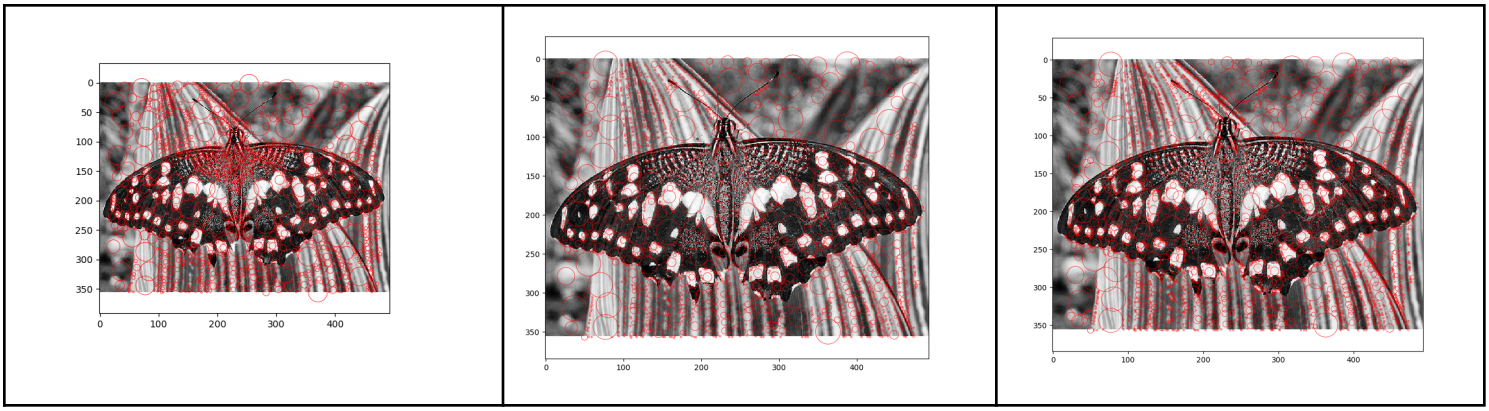
As the threshold value for blob detection [variable name : 'max_value_in_sliced_matrix'] decreases, the number of blobs detected increases and so does the time to execute the code increases.

Hyper-parameters Tuning
Now we worked on tuning these hyper-parameters corresponding to each image to get the desired results. The detection results corresponding to each image will depend on these hyper-parameters. This step in the project was the most critical as fine tuning of these parameters gives us good results, whereas if any of

these parameters are not tuned to perfection, we might end up getting bad results (also at the cost of high computation time)

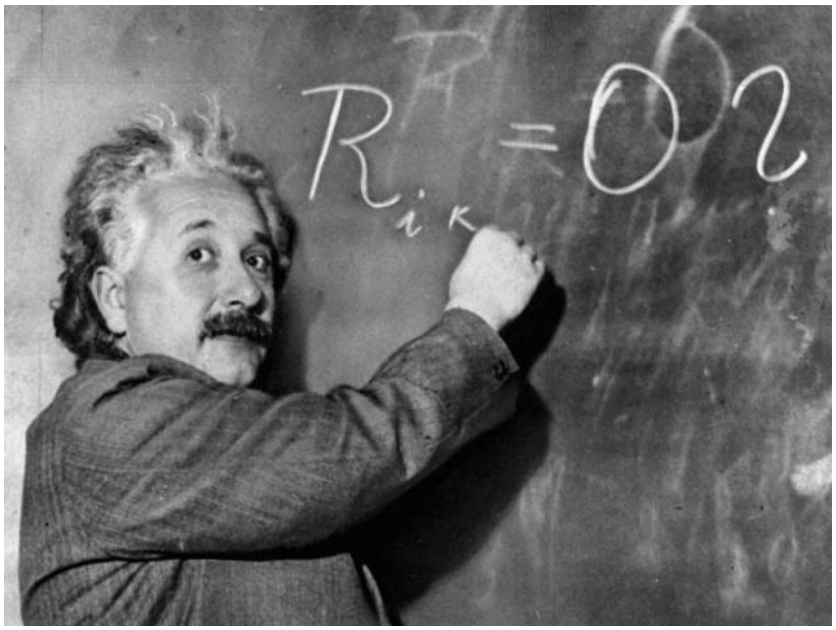
Results:

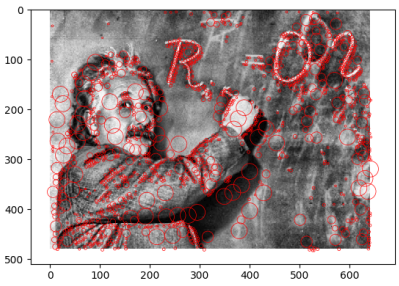
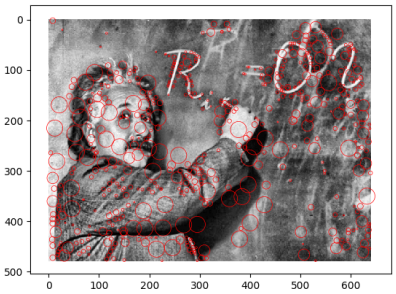
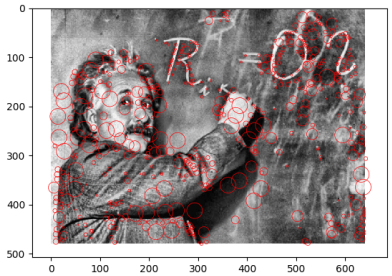
1. Test image : ‘Butterfly.jpg’



Hyper-parameters • distance = sqrt(2) * radius of the biggest blob • size of sliced matrix = 5x5 • max_value_in_sliced_matrix = 0.01 • sigma_initial = 1 • K = sqrt(2) • Overlap_threshold = 0.1 • size_of_octave = 10 • scale_for_sigma_to_kernel_size = 4 **Performance Parameters:** • Blobs before NMS : 64245 • Blobs after NMS : 2046 • Time to run NMS : 277.79 sec • Time to Plot keypoints : 287 sec	**Hyper-parameters** • distance = sqrt(2) * radius of the biggest blob • size of sliced matrix = 9x9 • max_value_in_sliced_matrix = 0.01 • sigma_initial = 1 • K = sqrt(2) • Overlap_threshold = 0.1 • size_of_octave = 10 • scale_for_sigma_to_kernel_size = 6 **Performance Parameters:** • Blobs before NMS : 34157 • Blobs after NMS : 2960 • Time to run NMS : 95 sec • Time to Plot keypoints : 10.656 sec	**Hyper-parameters** • distance = sqrt(2) * radius of the biggest blob • size of sliced matrix = 9x9 • max_value_in_sliced_matrix = 0.01 • sigma_initial = 1 • K = sqrt(2) • Overlap_threshold = 0.3 • size_of_octave = 10 • scale_for_sigma_to_kernel_size = 6 **Performance Parameters:** • Blobs before NMS : 34157 • Blobs after NMS : 4214 • Time to run NMS : 93.19 sec • Time to Plot keypoints : 14.75 sec

2. Test image : ‘einstein.jpg’



<table>
<tr>
<td>  </td>
<td>  </td>
<td>  </td>
</tr>
<tr>
<td>

Hyper-parameters

- distance = sqrt(2) * radius of the biggest blob
- size of sliced matrix = 11 x 11
- max_value_in_sliced_matrix = 0.05
- sigma_initial = 1
- K = sqrt(2)
- Overlap_threshold = 0.3
- size_of_octave = 10
- scale_for_sigma_to_kernel_size = 9

Performance Parameters:

- Blobs before NMS : 38663
- Blobs after NMS : 1450
- Time to run NMS :76.15 sec
- Time to Plot keypoints : 79.78 sec

</td>
<td>

Hyper-parameters

- distance = sqrt(2) * radius of the biggest blob
- size of sliced matrix = 7 x 7
- max_value_in_sliced_matrix = 0.05
- sigma_initial = 1
- K = sqrt(2)
- Overlap_threshold = 0.2
- size_of_octave = 10
- scale_for_sigma_to_kernel_size = 6

Performance Parameters:

- Blobs before NMS : 45545
- Blobs after NMS : 575
- Time to run NMS : 99.89 sec
- Time to Plot keypoints : 102.375 sec

</td>
<td>

Hyper-parameters

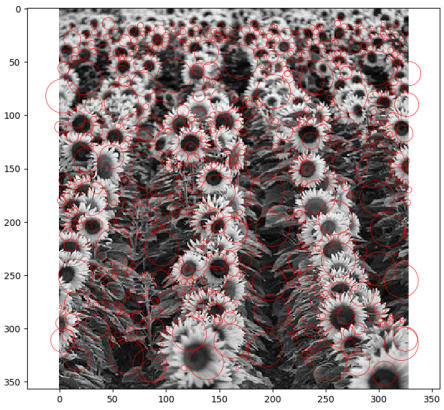
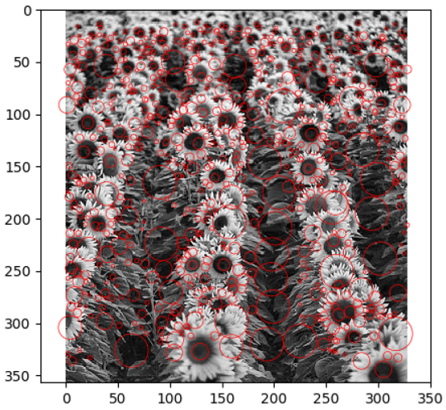
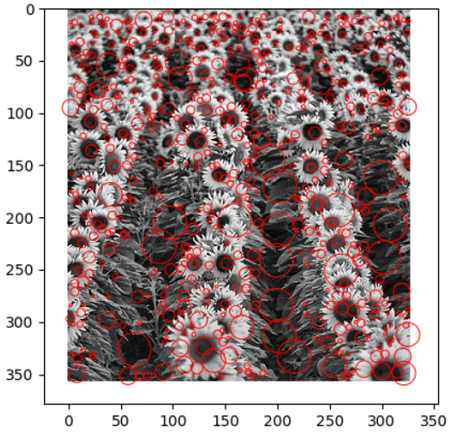
- distance = sqrt(2) * radius of the biggest blob
- size of sliced matrix = 11x11
- max_value_in_sliced_matrix = 0.053
- sigma_initial = 1
- K = sqrt(2)
- Overlap_threshold = 0.3
- size_of_octave = 10
- scale_for_sigma_to_kernel_size = 6

Performance Parameters:

- Blobs before NMS : 29354
- Blobs after NMS : 540
- Time to run NMS : 48.078 sec
- Time to Plot keypoints : 48.53 sec

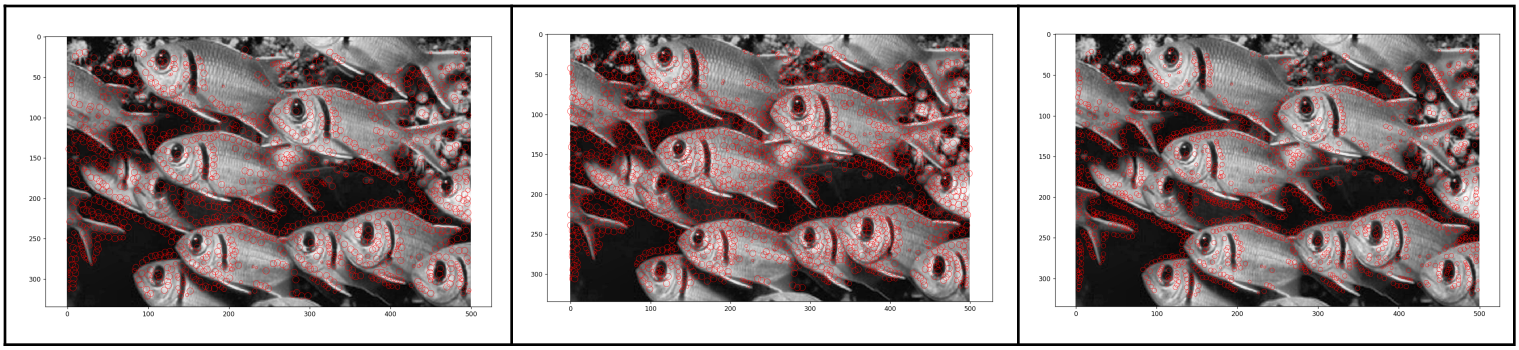
</td>
</tr>
</table>

3. Test image : ‘sunflowers.jpg’

		
Hyper-parameters • distance = sqrt(2) * radius of the biggest blob • size of sliced matrix = 5 x 5 • max_value_in_sliced_matrix = 0.012 • sigma_initial = 1 • K = sqrt(2) • Overlap_threshold = 0.15 • size_of_octave = 10 • scale_for_sigma_to_kernel_size = 3 **Performance Parameters:** • Blobs before NMS : 27821	**Hyper-parameters** • distance = sqrt(2) * radius of the biggest blob • size of sliced matrix = 5 x 5 • max_value_in_sliced_matrix = 0.012 • sigma_initial = 1 • K = sqrt(2) • Overlap_threshold = 0.15 • size_of_octave = 10 • scale_for_sigma_to_kernel_size = 6 **Performance Parameters:** • Blobs before NMS : 28339	**Hyper-parameters** • distance = sqrt(2) * radius of the biggest blob • size of sliced matrix = 7x7 • max_value_in_sliced_matrix = 0.0012 • sigma_initial = 1 • K = sqrt(2) • Overlap_threshold = 0.2 • size_of_octave = 10 • scale_for_sigma_to_kernel_size = 6 **Performance Parameters:** • Blobs before NMS : 20955

• Blobs after NMS : 651 • Time to run NMS :72.57 sec • Time to Plot keypoints : 73.2656 sec	• Blobs after NMS : 713 • Time to run NMS : 75.20 sec • Time to Plot keypoints : 75.984 sec	• Blobs after NMS : 558 • Time to run NMS : 44.48 sec • Time to Plot keypoints : 44.078 sec

4. Test image : ‘fishes.jpg’



Hyper-parameters • distance = sqrt(2) * radius of the biggest blob • size of sliced matrix = 7 x 7 • max_value_in_sliced_matrix = 0.009 • sigma_initial = 1 • K = sqrt(2) • Overlap_threshold = 0.35 • size_of_octave = 6 • scale_for_sigma_to_kernel_size = 20 **Performance Parameters:** • Blobs before NMS : 40830 • Blobs after NMS : 1607 • Time to run NMS : 9.09 sec • Time to Plot keypoints : 10.15 sec	**Hyper-parameters** • distance = sqrt(2) * radius of the biggest blob • size of sliced matrix = 3 x 3 • max_value_in_sliced_matrix = 0.01 • sigma_initial = 1 • K = sqrt(2) • Overlap_threshold = 0.35 • size_of_octave = 6 • scale_for_sigma_to_kernel_size = 20 **Performance Parameters:** • Blobs before NMS : 68784 • Blobs after NMS : 2369 • Time to run NMS : 20.28 sec • Time to Plot keypoints : 21.78 sec	**Hyper-parameters** • distance = sqrt(2) * radius of the biggest blob • size of sliced matrix = 7x7 • max_value_in_sliced_matrix = 0.009 • sigma_initial = 1 • K = sqrt(2) • Overlap_threshold = 0.35 • size_of_octave = 5 • scale_for_sigma_to_kernel_size = 6 **Performance Parameters:** • Blobs before NMS : 35361 • Blobs after NMS : 1969 • Time to run NMS : 4.3281 sec • Time to Plot keypoints : 6.625 sec

Other test results in the zip folder.

Conclusion-

The program gives a detailed description of how to implement Blob detection and Non-Maximum suppression on a given image. Few takeaways are the comparison between smoothening with convolution and using FFT. While regular convolution took O(N^4) computations, using FFT drastically reduced it to O($N^2 Log^2$N). Using Difference of Gaussian instead of Laplacian of Gaussian also saved us computations yielding better results in time. Furthermore, we observed the threshold values, sigma values and kernel size play an important role in accurately detecting blobs. The number of blobs depended on the size of the initial kernel and lesser blobs were detected with increasing kernel size. But using higher values made the computations very slow, so we used trial and error to find the right balance. The non-maximum suppression algorithm was used to remove redundant circles, and overlapping circles from where the blob was detected.

Future work and further improvements: We plan to make use of the Gaussian Pyramid Layers to subsample the image and further create octaves to make it even more precise. Using other "divide and conquer" algorithms to further improve on time.

Difficulties faced:

1. Reusing code for the convolution- the code was extremely slow and computation-wise heavy.
2. Using LoG , where we had to convolute the images twice. So instead we went ahead with using DoG. We reused the project 2 code to have a similar background.
3. Detecting Blobs and implementation of NMS. Finding the right threshold and using the right sigma values.
4. We had to modify the code for the NMS such so that the circles were not aliasing

Individual contributions:

Aaron Mathew - Worked on refining old codes, especially the convolution code, and converting it into convolution using FFT. Implemented peer programming with other teammates to implement Non maximum suppression.

Hritwik Shukla - Implemented the initial code for DoG, compared and researched the difference between LoG and DoG. Researched about parameters tuning for Blob detection to get optimal results.

Sushant Kolhe - Implemented Non-maximum suppression and tuned the code to refine it even more to get better results. Applied smart inbuilt functions like ckDTree to perform nonmaximum suppression over blobs within a specific distance to reduce compile time.

References:

1. https://projectsflix.com/opencv/laplacian-blob-detector-using-python/
2. https://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf