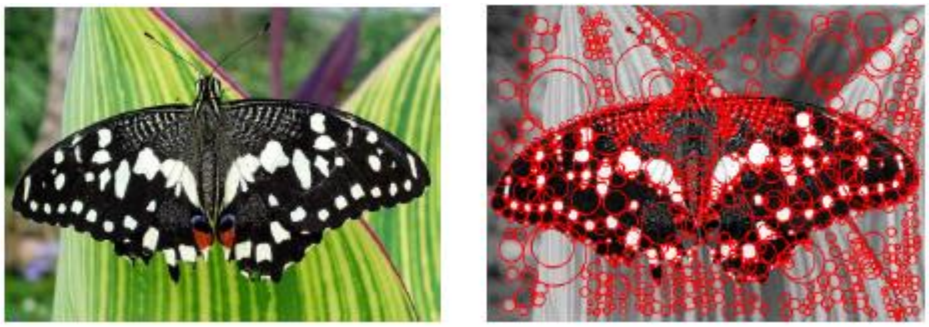
ECE 558 Project 3 – Laplacian Blob Detector (Extra Credits)

Derik Muñoz Solis
dfmunoz@ncsu.edu

Khoa Do
kddo@ncsu.edu

I. Introduction

The objective of this project is to implement a Laplacian blob detector, which generates features that are invariant to scaling for feature-tracking application in computer vision. Our approach to the project is to convolve the input image with the "blob filter" at different scales, in which the input image stays constant and the blob filter (e.g., Gaussian kernel) increases in size by a factor *k*.



(a) Input Image (b) Result

Figure 1. Example Result of Blob Detection.

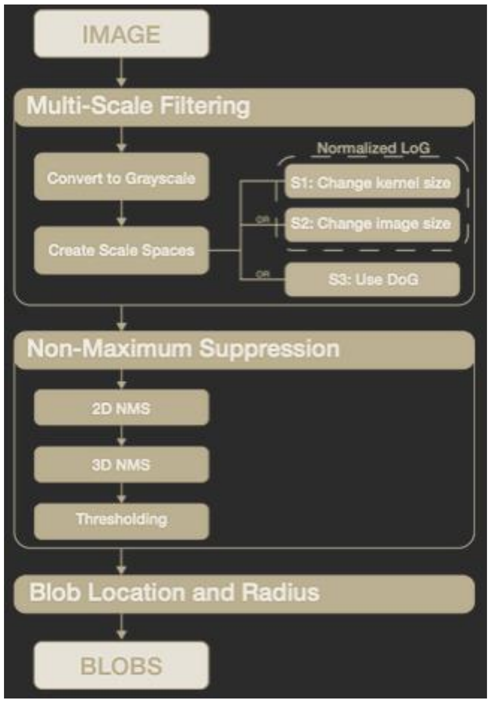


Figure 2. Different Approaches to the Implementation of Blob Detector [1].

II. Algorithm

The algorithm's outline of the bob detector in this project is as follows:

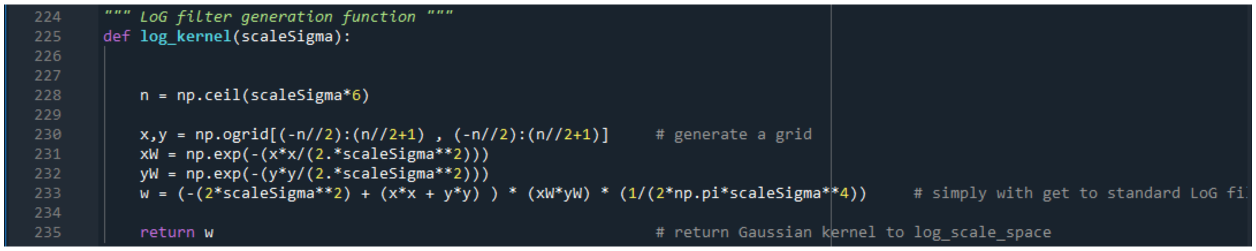
- Generating a Laplacian of Gaussian filter
- Building a Laplacian scale space starting with some initial scale and going for *n* iterations
    - Filtering image with scale-normalized Laplacian at current scale
    - Saving square Laplacian response for current level of scale space
    - Increasing scale by a factor *k*
- Performing non-maximum suppression
- Displaying resulting circles at their characteristic scale

1. Generating Laplacian of Gaussian Filter

The given equation for the Laplacian of Gaussian filter is as follows:

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

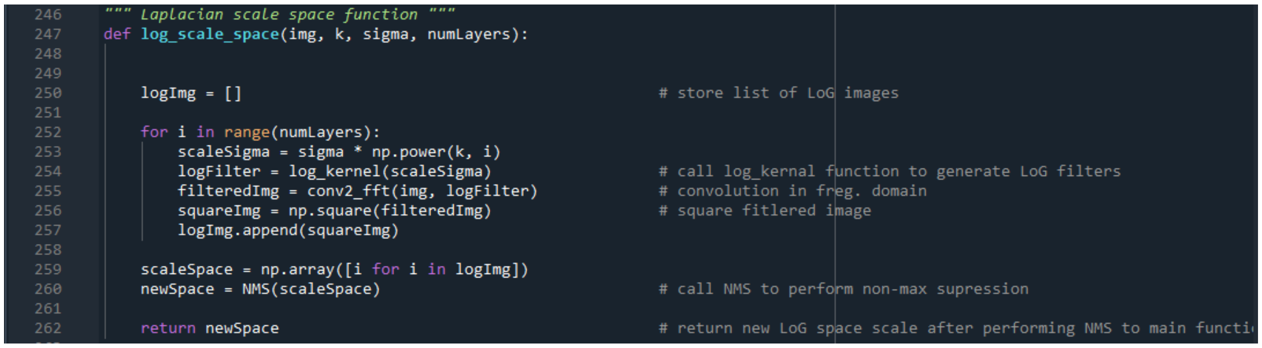
and is translated into Python code shown in figure 3.



```
224  """ LoG filter generation function """
225  def log_kernel(scaleSigma):
226  
227  
228      n = np.ceil(scaleSigma*6)
229  
230      x,y = np.ogrid[(-n//2):(n//2+1) , (-n//2):(n//2+1)]     # generate a grid
231      xW = np.exp(-(x*x/(2.*scaleSigma**2)))
232      yW = np.exp(-(y*y/(2.*scaleSigma**2)))
233      w = (-(2*scaleSigma**2) + (x*x + y*y) ) * (xW*yW) * (1/(2*np.pi*scaleSigma**4))     # simply with get to standard LoG fi
234  
235      return w                                                # return Gaussian kernel to log_scale_space
```

Figure 3. Code Snip 1 – Generating LoG Kernel.

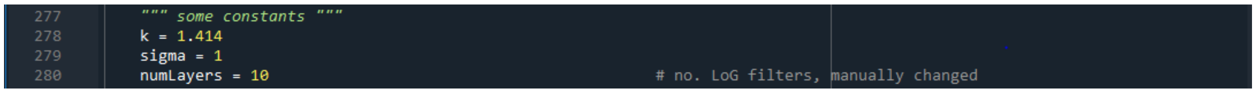
A grid is first generated (line 230) and then being filled with calculated values (line 231-232). The final kernel is put together in line 233. It is an expansion of equation 1. This kernel will increase in size and has increasing scaled values inside the grid as *scaleSigma* changed by the factor *k*.



```
""" Laplacian scale space function """
def log_scale_space(img, k, sigma, numLayers):


    logImg = []                                             # store list of LoG images

    for i in range(numLayers):
        scaleSigma = sigma * np.power(k, i)
        logFilter = log_kernel(scaleSigma)                  # call log_kernal function to generate LoG filters
        filteredImg = conv2_fft(img, logFilter)             # convolution in freg. domain
        squareImg = np.square(filteredImg)                  # square fitlered image
        logImg.append(squareImg)

    scaleSpace = np.array([i for i in logImg])
    newSpace = NMS(scaleSpace)                              # call NMS to perform non-max supression

    return newSpace                                         # return new LoG space scale after performing NMS to main functi
```

Figure 4. Code Snip 2 – Scale Space Function.

Figure 4 shows how *scaleSigma* is computed. It takes in the initial *sigma* multiplied by k^i where *sigma* and *i* are starting constants of choice defined in the main function and *i* is the number of layers of the LoG filter (e.g., *n iterations* as stated in the second bullet of the *Algorithm*).



```
    """ some constants """
    k = 1.414
    sigma = 1
    numLayers = 10                              # no. LoG filters, manually changed
```

Figure 5. Code Snip 3 – Constants.

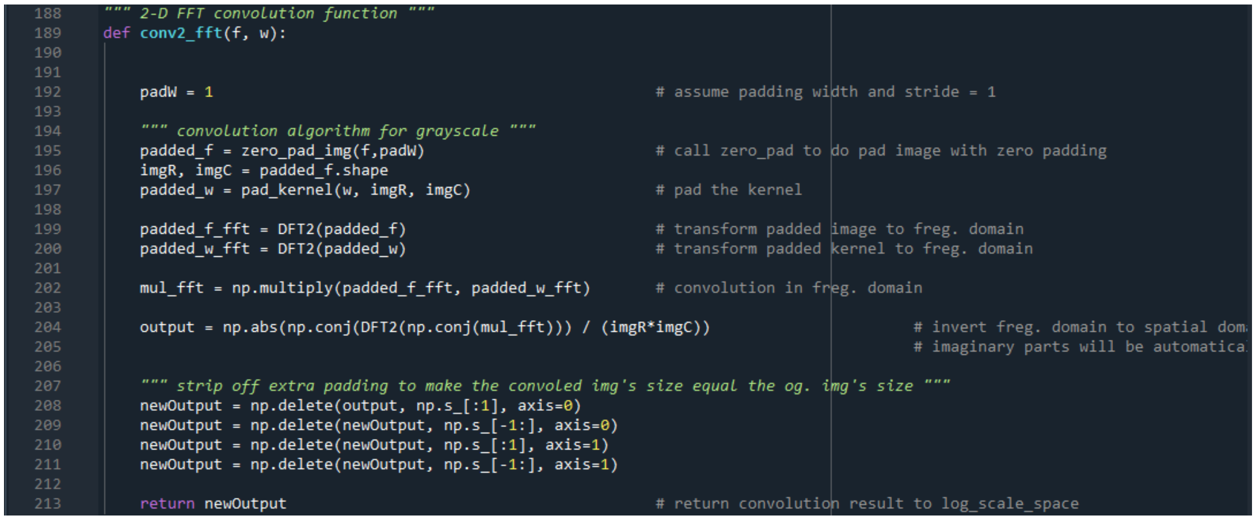
For testing images that will be included later in this report, we assign *k*, *sigma*, and *i* to be 1.414, 1, and 10 respectively.

2. Building Laplacian Scale Space

Like project 2, we create a scale space of images in this project. However, this scale space contains a list of squared resulting images from convolving the input image with each of the scaled Gaussian Kernel. As mentioned earlier, the input image remains the same and only the kernel changes (e.g., being upscaled) in each convolution. At the same time, non-maximum suppression (NMS) is performed on each of the convolved images. NMS function then returns resulting images to the scale space function. To be exact, our scale space is a list of post-NMS squared convolved images (see figure 4).

a) *Filtering Image with Scale-Normalized Laplacian at Current Scale*

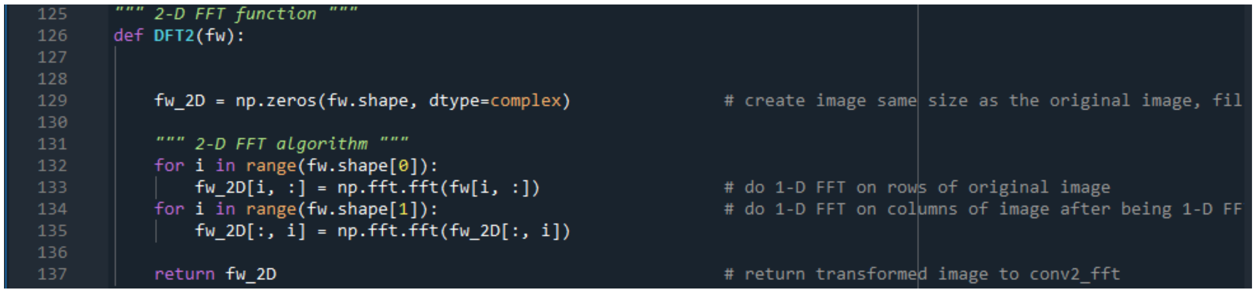
The input image is filtered by convolving with the scale-normalized kernel. A 2-D fast Fourier transform (FFT) convolution is implemented based on the convolution function in spatial domain from project 1 and the FFT function from project 2. It is 3-5 times faster compared to the convolution taking place in the spatial domain.



```
""" 2-D FFT convolution function """
def conv2_fft(f, w):


    padW = 1                                    # assume padding width and stride = 1

    """ convolution algorithm for grayscale """
    padded_f = zero_pad_img(f,padW)             # call zero_pad to do pad image with zero padding
    imgR, imgC = padded_f.shape
    padded_w = pad_kernel(w, imgR, imgC)        # pad the kernel

    padded_f_fft = DFT2(padded_f)               # transform padded image to freg. domain
    padded_w_fft = DFT2(padded_w)               # transform padded kernel to freg. domain

    mul_fft = np.multiply(padded_f_fft, padded_w_fft)       # convolution in freg. domain

    output = np.abs(np.conj(DFT2(np.conj(mul_fft))) / (imgR*imgC))              # invert freg. domain to spatial dom
                                                                                # imaginary parts will be automatica

    """ strip off extra padding to make the convoled img's size equal the og. img's size """
    newOutput = np.delete(output, np.s_[:1], axis=0)
    newOutput = np.delete(newOutput, np.s_[-1:], axis=0)
    newOutput = np.delete(newOutput, np.s_[:1], axis=1)
    newOutput = np.delete(newOutput, np.s_[-1:], axis=1)

    return newOutput                            # return convolution result to log_scale_space
```

Figure 6. Code Snip 4 – 2-D FFT Convolution Function.

Prior to the multiplication between the image and the kernel (line 202) which is the convolution in the frequency domain, the input image is padded using zero-padding and the kernel is padded based on the padded image's dimension. The zero-padding function from project 1 and the DFT2 function from project 2 are reused.



```python
""" 2-D FFT function """
def DFT2(fw):


    fw_2D = np.zeros(fw.shape, dtype=complex)               # create image same size as the original image, fil

    """ 2-D FFT algorithm """
    for i in range(fw.shape[0]):
        fw_2D[i, :] = np.fft.fft(fw[i, :])                  # do 1-D FFT on rows of original image
    for i in range(fw.shape[1]):                            # do 1-D FFT on columns of image after being 1-D FF
        fw_2D[:, i] = np.fft.fft(fw_2D[:, i])

    return fw_2D                                            # return transformed image to conv2_fft
```

Figure 7. Code Snip 5 – 2-D FFT Function.

Instead of performing the inverse fast Fourier transform (IDFT2 function from project 2) after the DTF2's multiplication, we apply the conjugate technique (line 204, figure 6) which is equivalent to the IDFT2 function [2]. This technique saves extra computation power, thus performing faster than the traditional IDFT2 (~10 seconds faster). The absolute value is taken later in line 204 to compute the magnitude of the complex values (A+Bi) resulted from the *DFT2* and the conjugates. It would also work if we were going to discard the imaginary parts and only take the real parts for computation since the imaginary coefficients (B) are significant small comparing to A's. Although this is faster than computing the magnitude, there is not a significant improvement in execution speed. Moreover, computing the magnitude guarantees a more accurate bob detector. The filtered image then goes through the final process (line 208-211, figure 6) to make sure that its size is the same as the original image's size.

b) ***Saving Square of Laplacian Response for Current level of Scale Space***

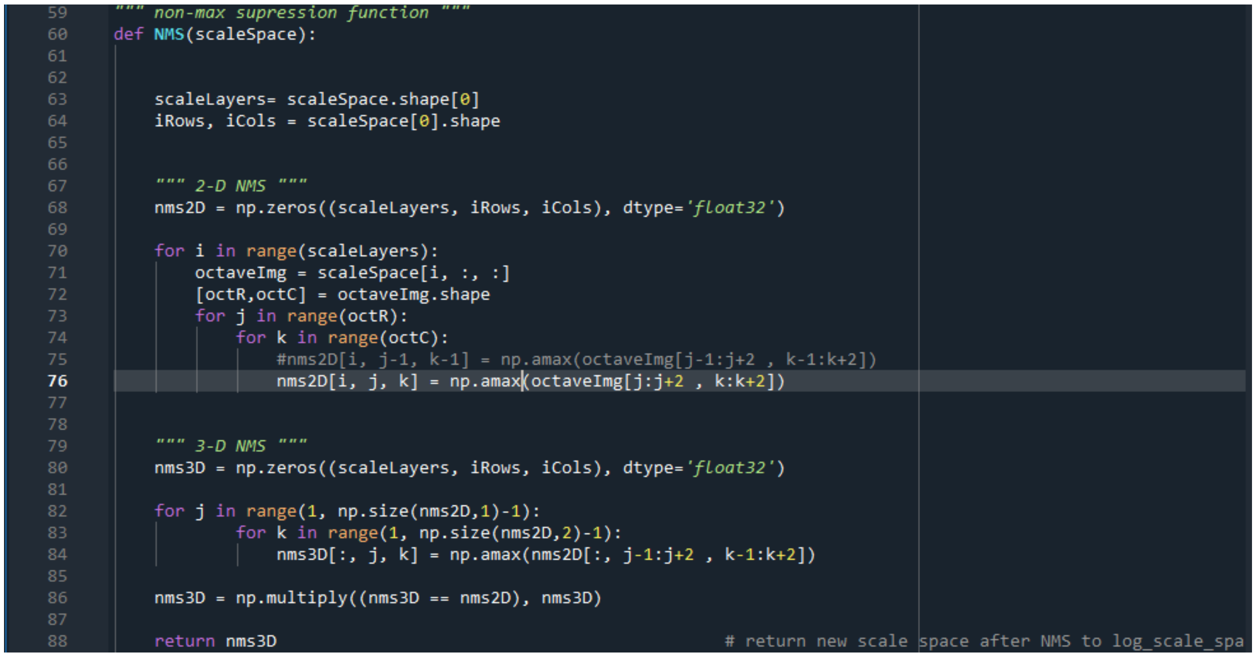
The convolved response (filtered image) is squared (line 256, figure 4) to ensure that maxima (peaks) is obtained instead of minima. The minima is resulted from convolving the image with the filter(s) of same length.

c) ***Increasing Scale by a Factor k***

As mentioned earlier in *1. Generating Laplacian of Gaussian Filter* (*), the kernel size is changed (or scaled) by a factor *k*. Referring to (*) for details.

3. Performing Non-Maximum Suppression

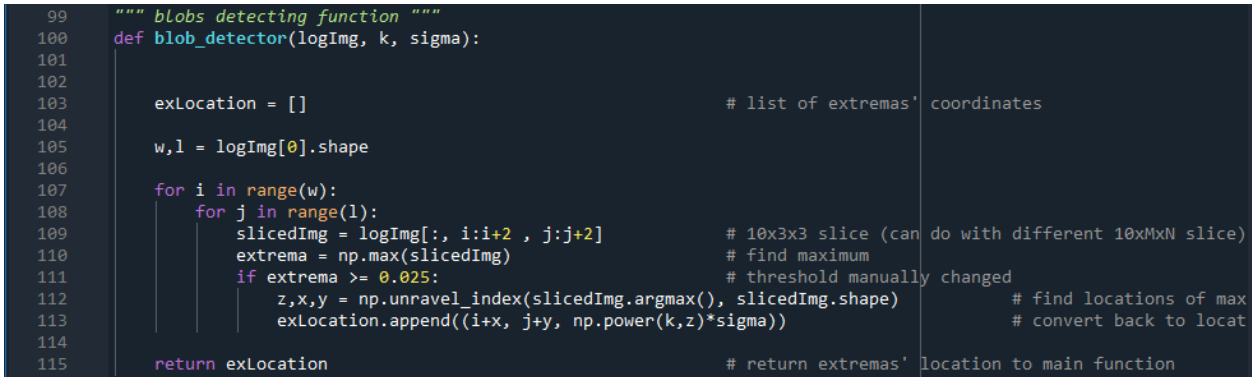
Non-maximum suppression (NMS) is performed in 2-D and then 3-D.



```python
""" non-max supression function """
def NMS(scaleSpace):


    scaleLayers= scaleSpace.shape[0]
    iRows, iCols = scaleSpace[0].shape


    """ 2-D NMS """
    nms2D = np.zeros((scaleLayers, iRows, iCols), dtype='float32')

    for i in range(scaleLayers):
        octaveImg = scaleSpace[i, :, :]
        [octR,octC] = octaveImg.shape
        for j in range(octR):
            for k in range(octC):
                #nms2D[i, j-1, k-1] = np.amax(octaveImg[j-1:j+2 , k-1:k+2])
                nms2D[i, j, k] = np.amax(octaveImg[j:j+2 , k:k+2])


    """ 3-D NMS """
    nms3D = np.zeros((scaleLayers, iRows, iCols), dtype='float32')

    for j in range(1, np.size(nms2D,1)-1):
            for k in range(1, np.size(nms2D,2)-1):
                nms3D[:, j, k] = np.amax(nms2D[:, j-1:j+2 , k-1:k+2])

    nms3D = np.multiply((nms3D == nms2D), nms3D)

    return nms3D                                        # return new scale space after NMS to log_scale_spa
```

Figure 8. Code Snip 6 – NMS Function.

The 2-D NMS takes eight neighbors of the pixel of interest and extracts the maximum value out of the total nine pixels per layer (or scale); then, the 3-D NMS does this across the scale space (all layers). Maximum values of pixels in the same region across the scale space are retained for the original values and their locations are extracted. The rest is set 0 to ensure that there are no other features extracted from that region, thus avoiding overlapping when plotting the circles.



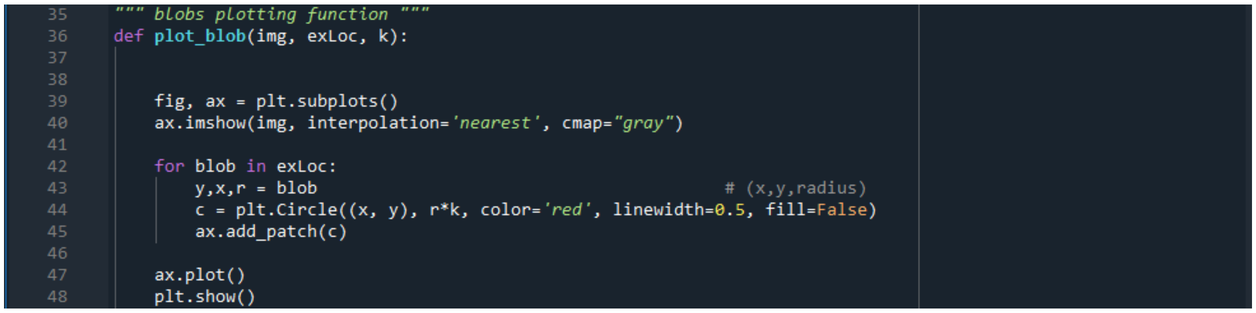
```python
""" blobs detecting function """
def blob_detector(logImg, k, sigma):


    exLocation = []                                     # list of extremas' coordinates

    w,l = logImg[0].shape

    for i in range(w):
        for j in range(l):
            slicedImg = logImg[:, i:i+2 , j:j+2]        # 10x3x3 slice (can do with different 10xMxN slice)
            extrema = np.max(slicedImg)                 # find maximum
            if extrema >= 0.025:                        # threshold manually changed
                z,x,y = np.unravel_index(slicedImg.argmax(), slicedImg.shape)          # find locations of max
                exLocation.append((i+x, j+y, np.power(k,z)*sigma))                     # convert back to locat

    return exLocation                                   # return extremas' location to main function
```

Figure 9. Code Snip 7 – Thresholding.

Next, we threshold the peaks that have values greater than or equal to 0.25 to be displayed.

4. Displaying resulting circles at their characteristic scale

The radius of the circles to be plotted is chosen based on the sigma value for at the corresponding scale (σ = k*r). The circle parameters x, y, and radius are then appended to a list which can be plotted on the figure once all the maxima have been characterized.



```python
""" blobs plotting function """
def plot_blob(img, exLoc, k):


    fig, ax = plt.subplots()
    ax.imshow(img, interpolation='nearest', cmap="gray")

    for blob in exLoc:
        y,x,r = blob                                    # (x,y,radius)
        c = plt.Circle((x, y), r*k, color='red', linewidth=0.5, fill=False)
        ax.add_patch(c)

    ax.plot()
    plt.show()
```

Figure 10. Code Snip 8 – Plotting Blobs.

III. Results

The program takes in any image as grayscale and scales its intensity to between 0 and 1. This also save some computation power and thus performing better than the grayscale image itself. For testing, we use the flowing setup:

- *k* = 1.5
- *sigma* = 2
- *numLayers* = 7
- Gray image(s) with scaled intensity between 0 and 1
- Running the program on Spyder IDE
- Adding a timer to the program to measure its performance to be considered for extra credits

Four images provided by Dr. Wu and four images of our choice are used for testing. Upon completion, the following results are obtained respectively:

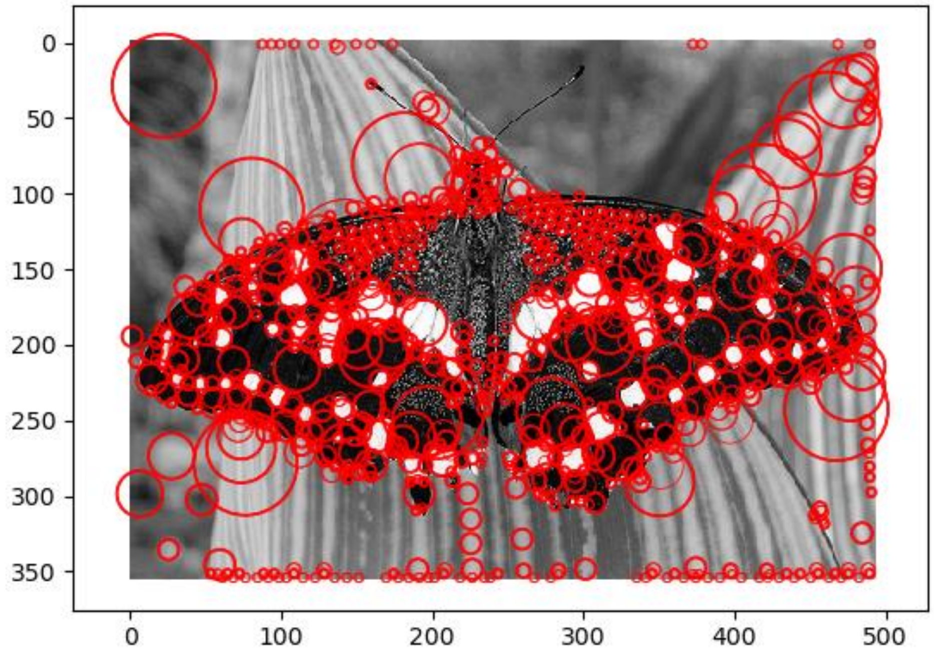


Figure 11. Result 1 – Runtime: 12.78 Seconds.

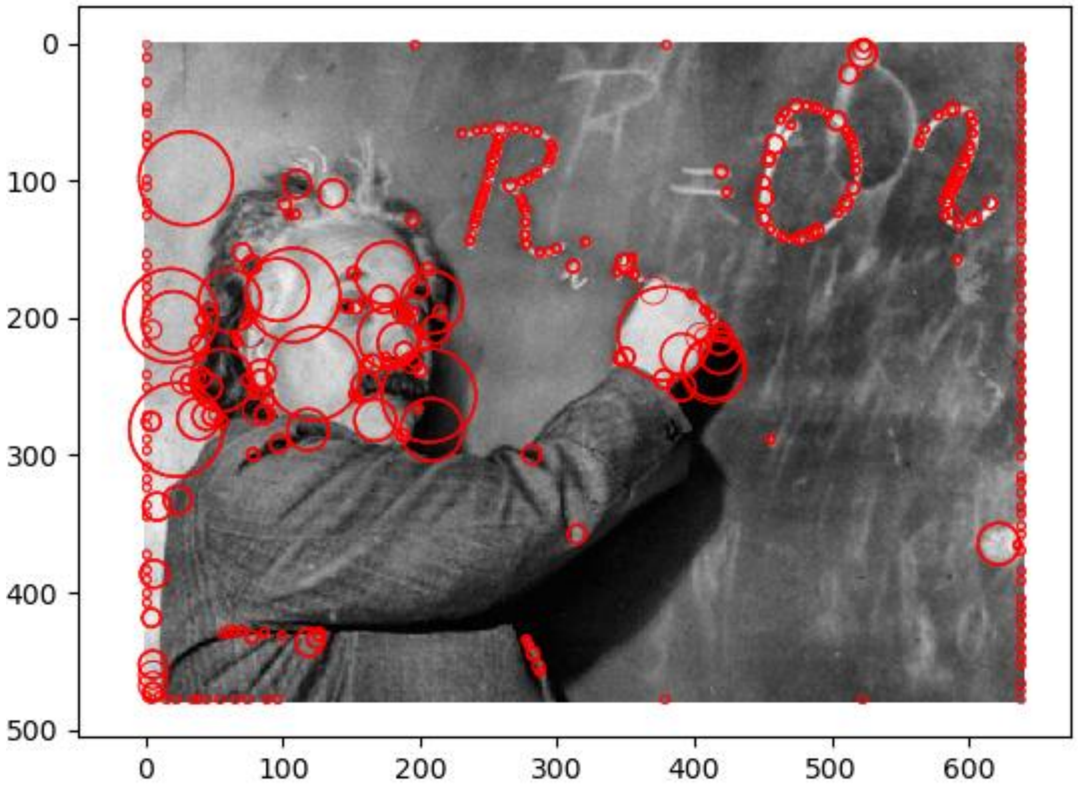


Figure 12. Result 2 – Runtime: 22.78 Seconds.

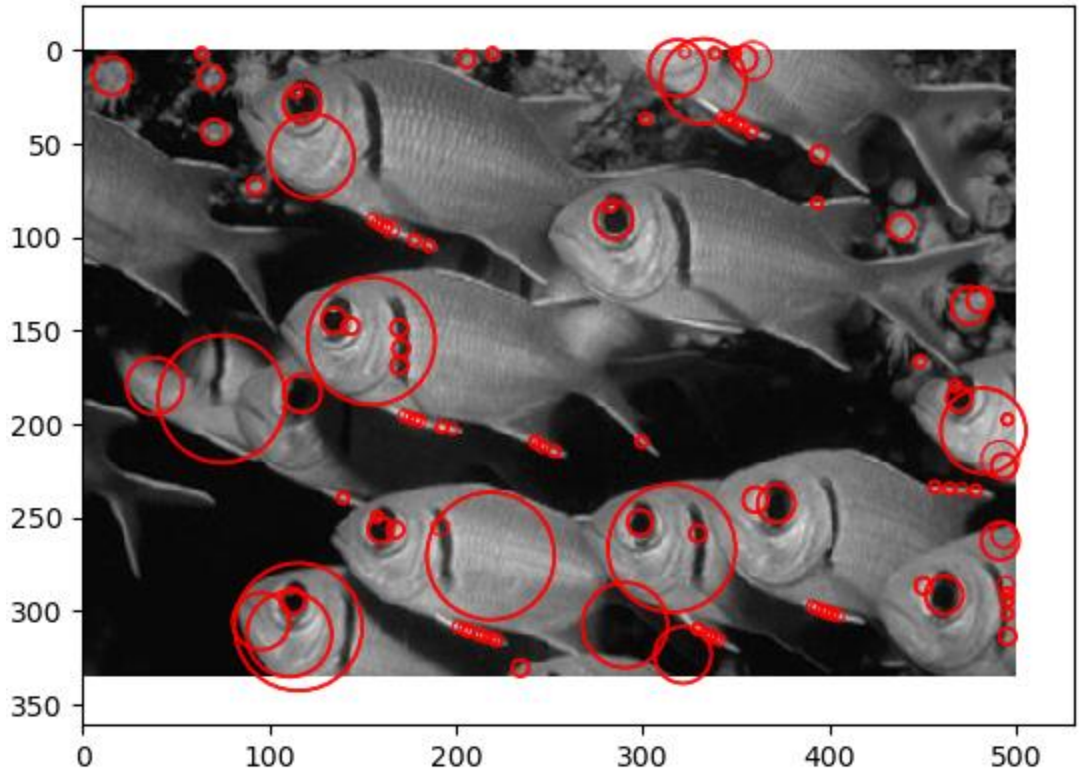


Figure 13. Result 3 – Runtime: 10.89 Seconds.

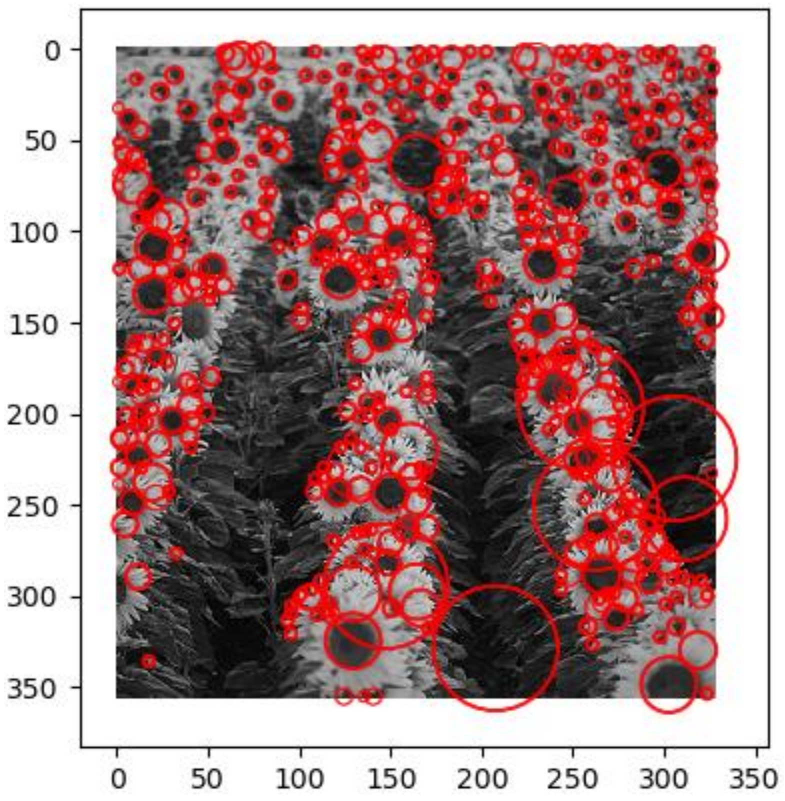


Figure 14. Result 4 – Runtime: 9.38 Seconds.

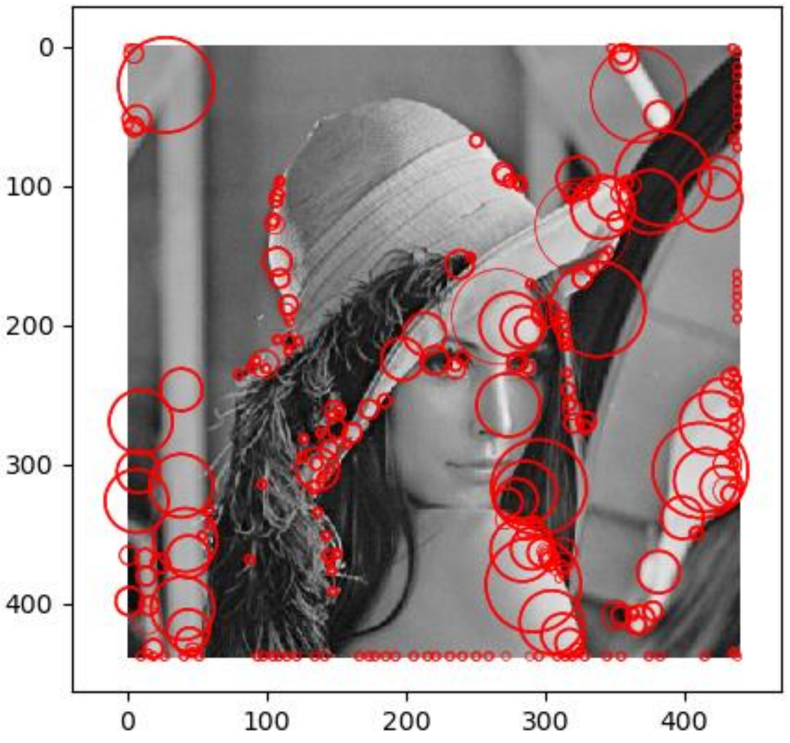


Figure 15. Result 5 – Runtime: 13.53 Seconds.

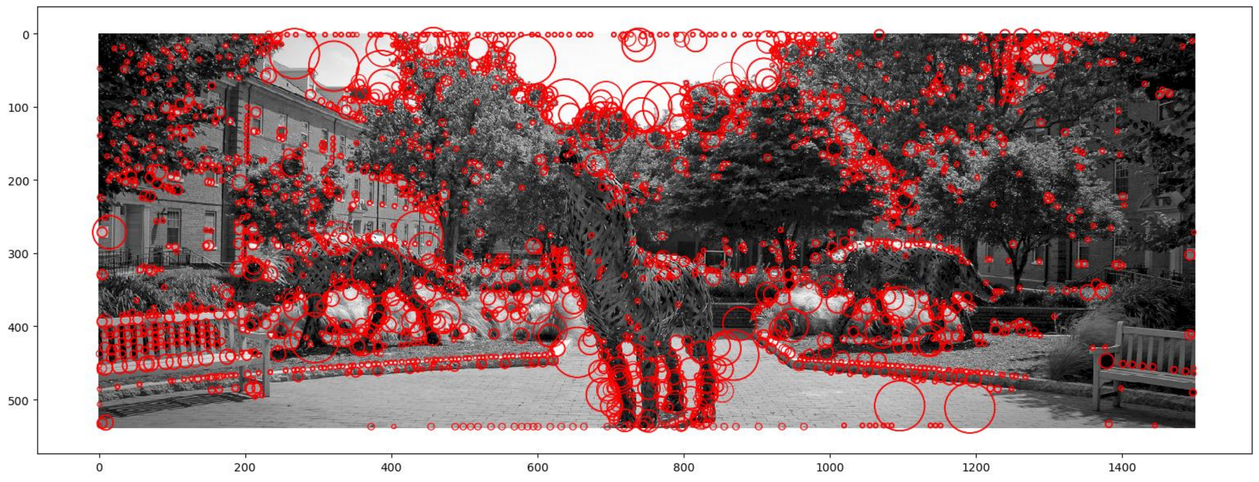


Figure 16. Result 6 – Runtime: 61.96 Seconds.

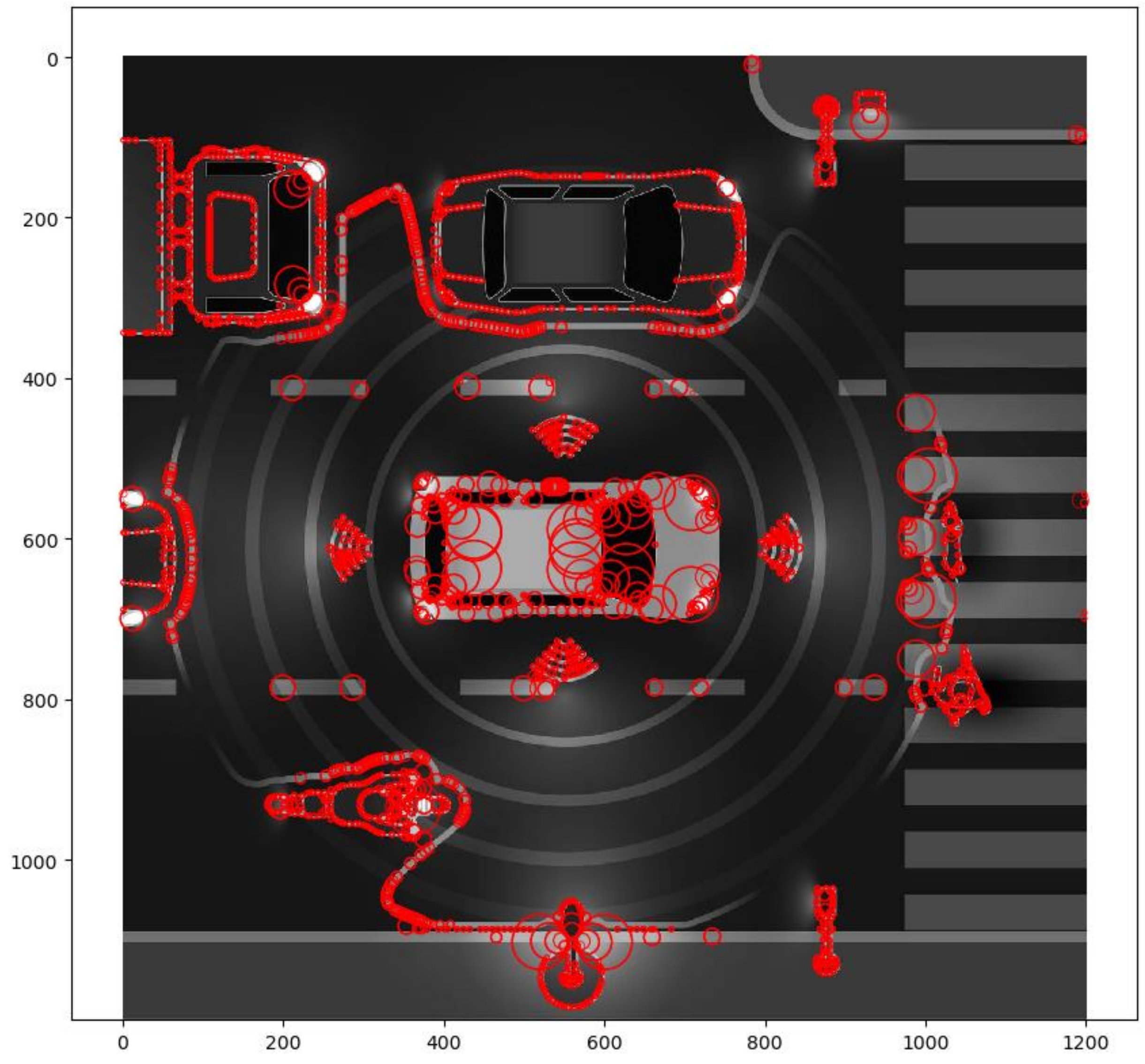


Figure 17. Result 7 – Runtime: 104.73 Seconds.

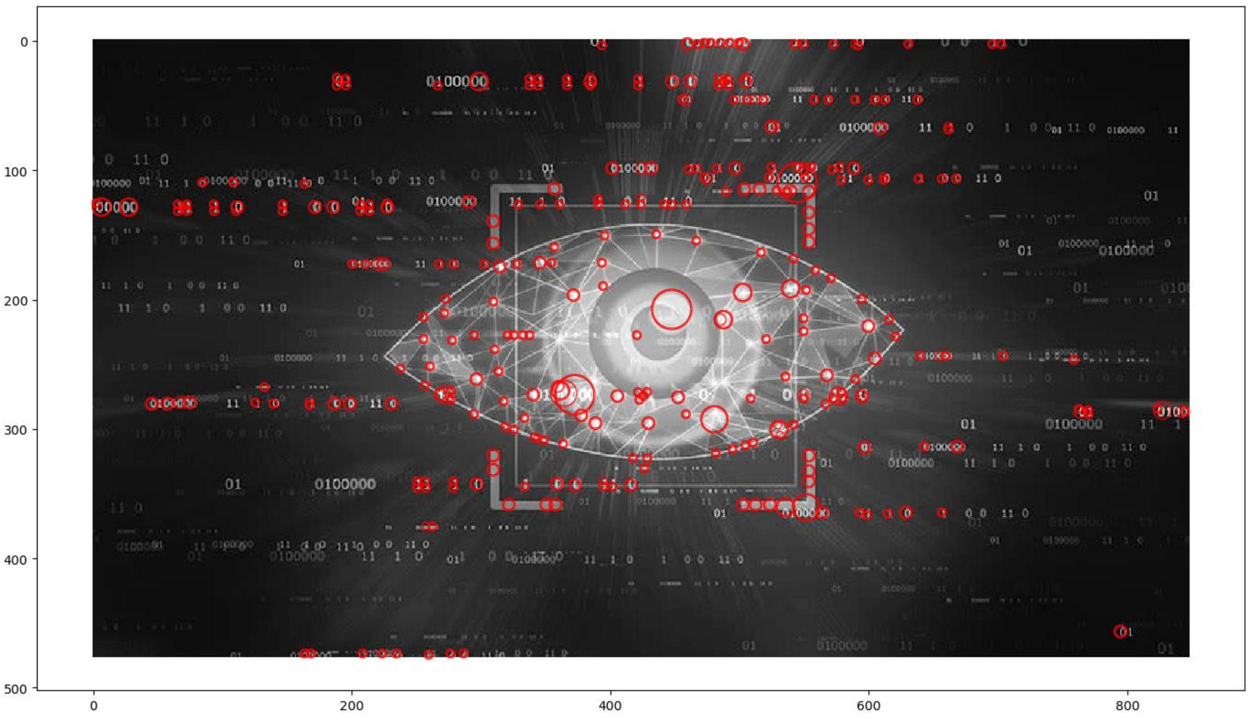


Figure 18. Result 8 – Runtime: 30.76 Seconds.

Th following table sums up the results obtained above:

Result	**Size (pixels)**	**Runtime (s)**
1	493x356	12.78
2	640x480	22.78
3	500x335	10.89
4	328x357	9.38
5	440x440	13.53
6	1500x539	61.96
7	1200x1200	104.73
8	848x477	30.76
Total	3,613,900	266.81
Performance	13,544.84465 pixels/s	

Table 1. Test Summary & Performance.

As the image size increases, the time to finish running the program also increases. Measuring the performance of our program by number of pixels it processes per second, we observe that it is almost 13,545 pixels processed every second.

IV. Obstacles & Overcome

The biggest two obstacles we encountered during the project was implementing the non-maximum suppression function and eliminating overlapping blobs finding about the threshold value. To implement the NMS function, we referred to the Scipy source code of the blob detection [3]. For eliminating overlapping blobs, it was a simple trick to print out value of *extrema*. Depending on the code setup (whether scaling the intensity to between 0 and 1, using convolution in spatial domain or frequency domain, using conjugates etc), the *extrema*'s value will vary significantly from a couple of hundred to thousand or even as small as 0.003. For our program, the threshold value is about 0.xx.

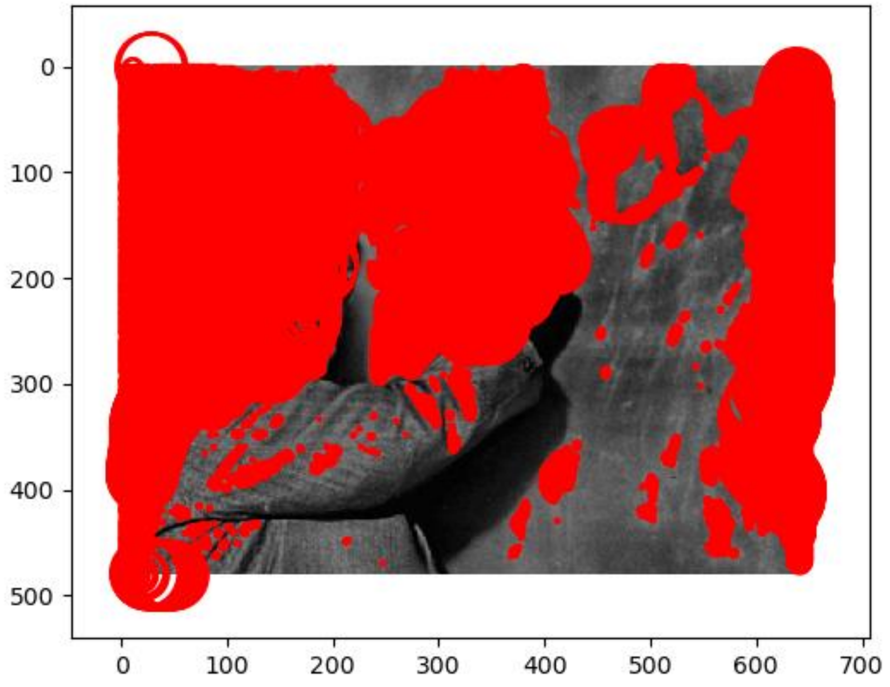
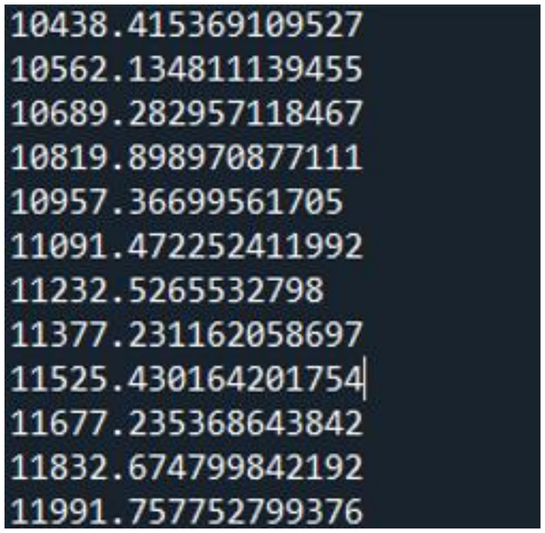


Figure 19. Result from Setting Wrong Threshold Value.



```
10438.415369109527
10562.134811139455
10689.282957118467
10819.898970877111
10957.36699561705
11091.472252411992
11232.5265532798
11377.231162058697
11525.430164201754
11677.235368643842
11832.674799842192
11991.757752799376
```

Figure 20. Example of Extrema Values for a Certain setup.

V. Conclusion & Future Improvement

Overall, we successfully completed the project and achieved an impressive fast Laplacian blob detector. We believe the performance of our blob detector is competitive and are excited to this will play out with other students in the class. There is still also room for improvement in in the future such as having an even cleaner and faster code or having a GUI for the user to upload his/her image(s) of choice and "hit run" to detect for blobs.

Reference

[1] htt D. Recchia, “Scale Invariant Blob Detection,” Recchia's Portfolio. [Online]. Available: https://www.drecchia.ca/scale-invariant-blob-detection. [Accessed: 30-Nov-2021].

[2] “DSP Tricks: Computing Inverse FFTs Using the Forward FFT,” Embedded.com, 16-Nov-2010. [Online]. Available: https://www.embedded.com/dsp-tricks-computing-inverse-ffts-using-the-forward-fft/. [Accessed: 30-Nov-2021].

[3] https://github.com/scikit-image/scikit-image/blob/main/skimage/feature/blob.py#L401-L564