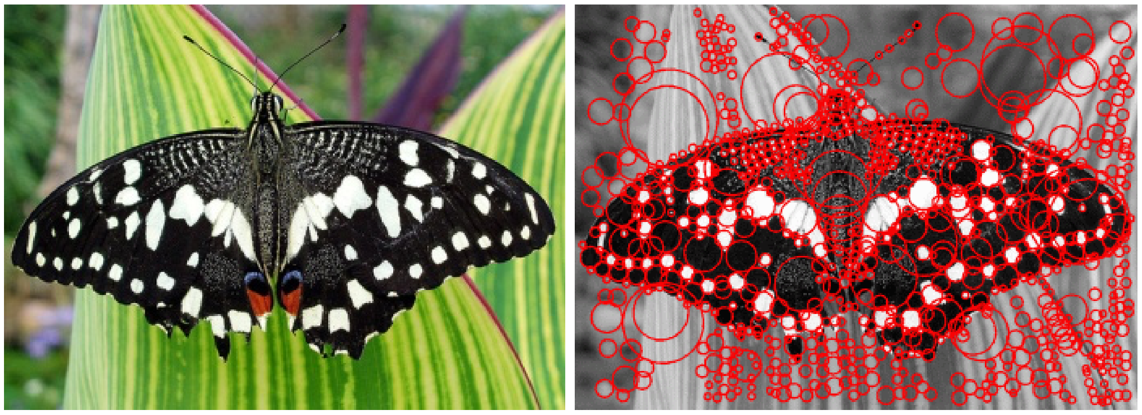
ECE 558 Project 3 - Laplacian Blob Detector (Extra Credits)

Derik Muñoz Solis, Khoa Do

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) Resulting Image

Figure 1: Example Results of Blob Detection.

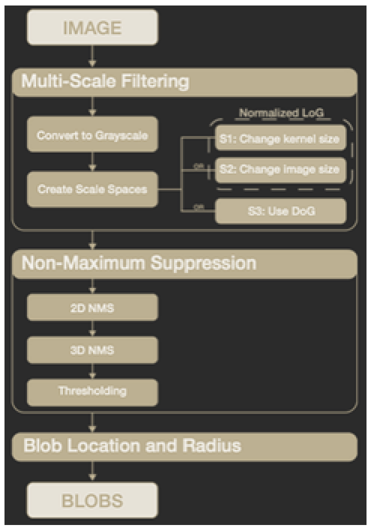


Figure 2: Different Approaches to the Implementation of Blob Detector [1].

II Algorithm

The general algorithm for this blob detector 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 of Laplacian response for current level of scale space
  - Increasing scale by a factor k
- Performing non-maximum suppression
- Displaying resulting circles at their characteristic scales

II.1 Generating a Laplacian of Gaussian Filter

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

The equation for the derivation of the LoG filter is given in equation 1. The implementation for this is shown in the Python code below

```python
253 """ LoG filter generation function """
254 def log_kernel(scaleSigma):
255 
256 
257     n = np.ceil(scaleSigma*6)
258     # generate a grid
259     x,y = np.ogrid[(-n//2):(n//2+1) , (-n//2):(n//2+1)]
260 
261     xW = np.exp(-(x*x/(2.*scaleSigma**2)))
262     yW = np.exp(-(y*y/(2.*scaleSigma**2)))
263         # simply with get to standard LoG filter formula
264     w = (-(2*scaleSigma**2) + (x*x + y*y) ) * (xW*yW) * (1/(2*np.pi*scaleSigma**4))
265     # return Gaussian kernel to log_scale_space
266     return w
```

Code 1: Generating LoG Kernel.

A grid is first generated (line 259) and then being filled with calculated values (line 261-262). The final kernel is put together in line 264. 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.

```python
277 """ Laplacian scale space function """
278 def log_scale_space(img, k, sigma, numLayers):
279         # store list of LoG images
280     logImg = []
281     for i in range(numLayers):
282         scaleSigma = sigma * np.power(k, i)
283                 # call log_kernal function to generate LoG filters
284         logFilter = log_kernel(scaleSigma)
285                 # convolution in freq. domain
286         filteredImg = conv2_fft(img, logFilter)
287                 # square fitlered image
288         squareImg = np.square(filteredImg)
289         logImg.append(squareImg)
290     scaleSpace = np.array([i for i in logImg])
291         # call NMS to perform non-max supression
292     newSpace = NMS(scaleSpace)
293         # return new LoG space scale after performing NMS to main function
294     return newSpace
```

Code 2: Scale Space Function.

The code snippet above 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).

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

Code 3: Constants.

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

II.2 Building a 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 Code 2).

II.2.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.

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

    # assume padding width and stride = 1
    padW = 1

    """ convolution algorithm for grayscale """
        # call zero_pad to do pad image with zero padding
    padded_f = zero_pad_img(f,padW)
    imgR, imgC = padded_f.shape
        # pad the kernel
    padded_w = pad_kernel(w, imgR, imgC)
    # transform padded image to freg. domain
    padded_f_fft = DFT2(padded_f)
        # transform padded kernel to freg. domain
    padded_w_fft = DFT2(padded_w)
    # convolution in freg. domain
    mul_fft = np.multiply(padded_f_fft, padded_w_fft)
        # invert freg. domain to spatial domain and calculate the magnitude
        # imaginary parts will be automatically discarded if np.abs is not used
    output = np.abs(np.conj(DFT2(np.conj(mul_fft))) / (imgR*imgC))


    """ 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 convolution result to log_scale_space
    return newOutput
```

Code 4: 2-D FFT Convolution Function.

Prior to the multiplication between the image and the kernel (line 230) 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.

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

    # create image same size as the original image, filled with 0's
    fw_2D = np.zeros(fw.shape, dtype=complex)

    """ 2-D FFT algorithm """
    for i in range(fw.shape[0]):
       # do 1-D FFT on rows of original image
                fw_2D[i, :] = np.fft.fft(fw[i, :])
        # do 1-D FFT on columns of image after being 1-D FFT-ed
    for i in range(fw.shape[1]):
        fw_2D[:, i] = np.fft.fft(fw_2D[:, i])
        # return transformed image to conv2_fft
    return fw_2D
```

Code 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 233, Code 4) 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 233 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 237-240) to make sure that its size is the same as the original image's size.

II.2.b Saving Square of Laplacian Response for Current Level of Scale Space.

The convolved response (filtered image) is squared (line 288, Code 2) 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.

II.2.c Increasing Scale by a Factor k

As mentioned earlier in section II.1, the kernel size is changed (or scaled) by a factor k. Refer to section II.1 for details.

II.3 Performing Non-Maximum Suppression

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

```
""" 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(1, octR):
            for k in range(1, octC):
                nms2D[i, j-1, k-1] = np.amax(octaveImg[j-1:j+2 , k-1: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 new scale space after NMS to log_scale_space
    return nms3D
```

Code 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 to 0 to ensure that there are no other features extracted from that region, thus avoiding overlapping blobs when plotting the circles.

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

    # list of extremas' coordinates
    exLocation = []

    w,l = logImg[0].shape

    for i in range(w):
        for j in range(l):
                        # 10x3x3 slice (can do with different 10xMxN slice)
            slicedImg = logImg[:, i:i+2 , j:j+2]
                        # find maximum
            extrema = np.max(slicedImg)
            if extrema >= 0.025:# threshold manually changed
                                # find locations of max values, convert linear indexes back to row-col indexes of the sliced img
                z,x,y = np.unravel_index(slicedImg.argmax(), slicedImg.shape)
                                # convert back to locations on the original image corresponding to that sigma
                exLocation.append((i+x, j+y, np.power(k,z)*sigma))
    # return extremas' location to main function
    return exLocation
```

Code 7: Thresholding.

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

II.4 Displaying Resulting Circles at Their Characteristic Scales.

The radius of the circles to be plotted is chosen based on the sigma value for at the corresponding scale ($\sigma = k * r$). 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):


    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*1.414, color='red', linewidth=0.5, fill=False)
        ax.add_patch(c)

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

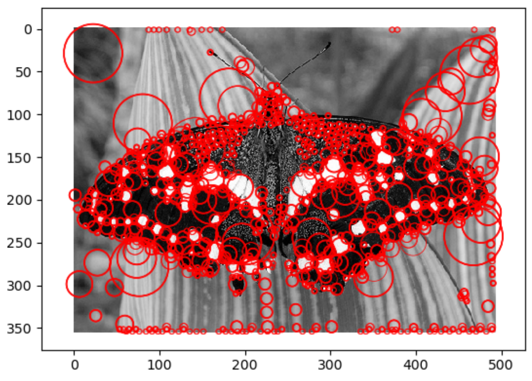
Code 8: Plotting Blobs.

III Results

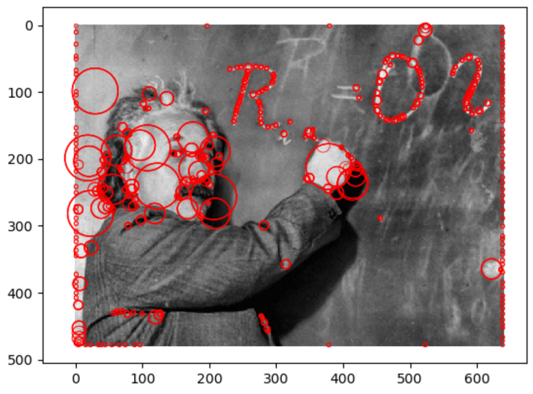
The program takes in any image as grayscale and scales its intensity to between 0 and 1. This also saves some computation power, thus performing better than the grayscale image itself. For testing, we use the flowing setup:

- $k = 1.5$
- $\sigma = 2$
- $numlayers = 7$
- Threshold value = 0.025
- 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

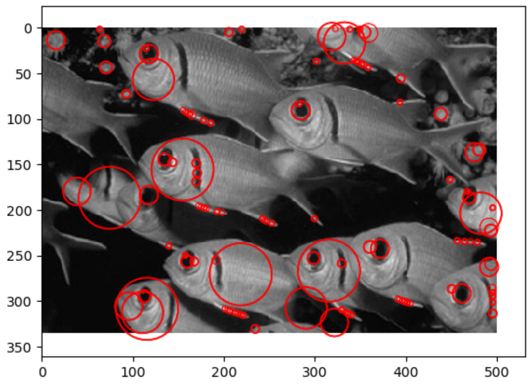
Four images provided by Dr. Wu and four images of our choice were used for testing. Upon completion, the following results are obtained respectively:



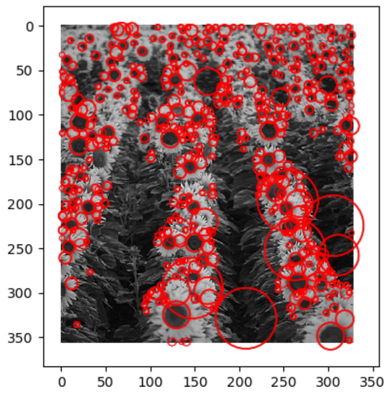
(a) Result 1 – Runtime: 12.78 Seconds.



(b) Result 2 – Runtime: 22.78 Seconds.

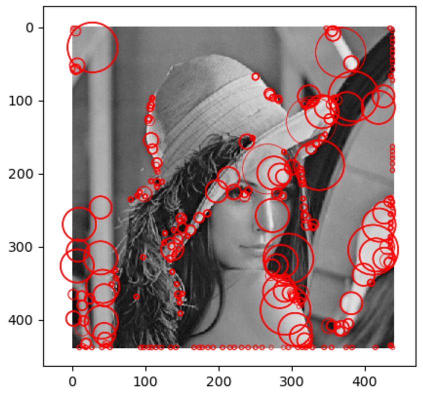


(c) Result 3 – Runtime: 10.89 Seconds.

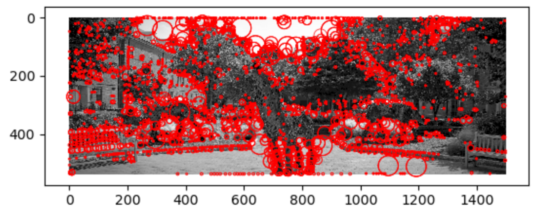


(d) Result 4 – Runtime: 9.38 Seconds.

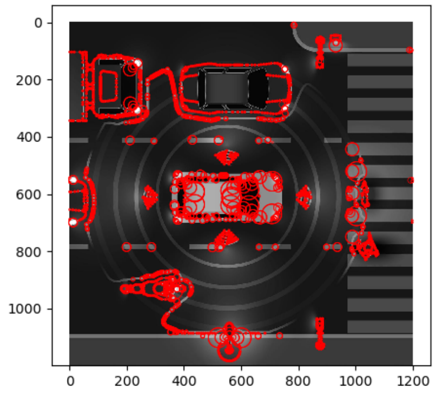
Figure 3: Results from Provided Images and Their Runtimes.



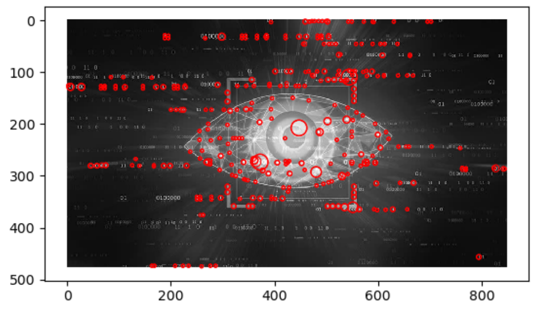
(a) Result 5 – Runtime: 13.53 Seconds.



(b) Result 6 – Runtime: 61.96 Seconds.



(c) Result 7 – Runtime: 104.73 Seconds.



(d) Result 8 – Runtime: 30.76 Seconds.

Figure 4: Results from Custom Images and Their Runtimes.

The 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 & Overcoming Them

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 in the rage from 0 to .99.

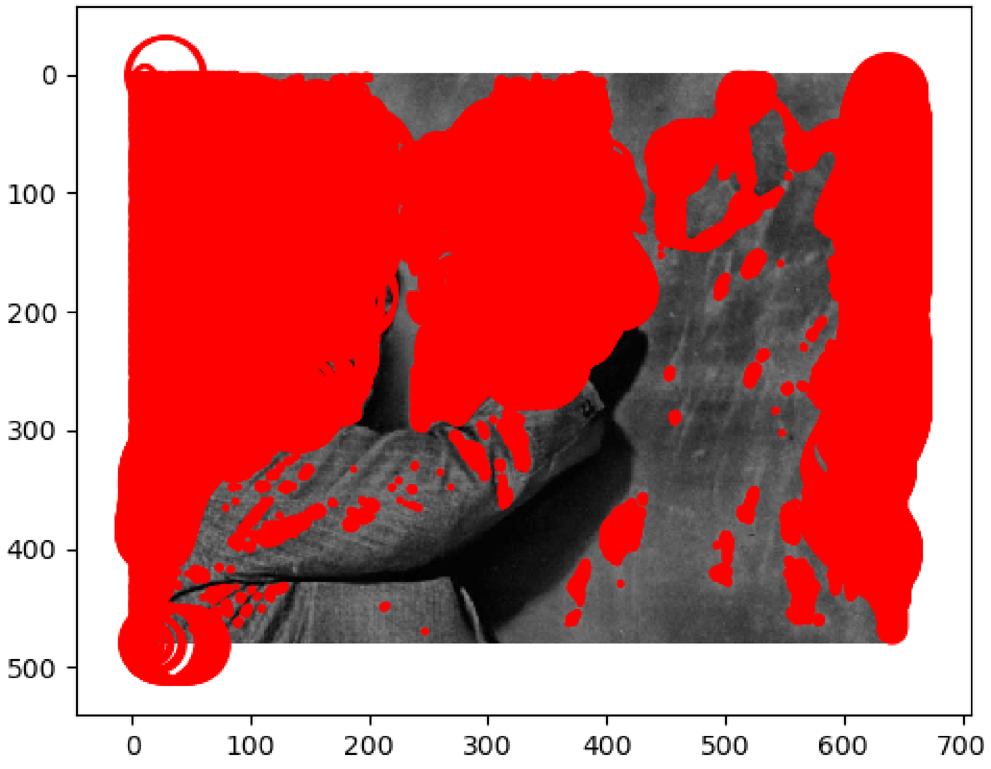
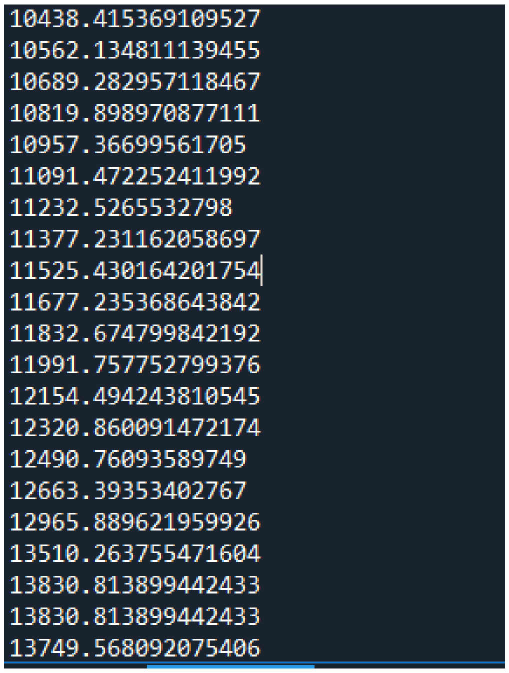


Figure 5: Result from Setting Wrong Threshold Value.



```
10438.415369109527
10562.134811139455
10689.282957118467
10819.898970877111
10957.36699561705
11091.472252411992
11232.5265532798
11377.231162058697
11525.430164201754
11677.235368643842
11832.674799842192
11991.757752799376
12154.494243810545
12320.860091472174
12490.76093589749
12663.39353402767
12965.889621959926
13510.263755471604
13830.813899442433
13830.813899442433
13749.568092075406
```

Figure 6: Example of Extrema Values for a Certain Setup.

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.

References

[1] 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