ECE558 Final Project

Nathan Starliper

12/5/2018

1 Introduction

The purpose of this project is to implement scale invariant blob detection [1] using a Laplacian of Gaussian (LoG) filter at varying scales. We then apply this LoG filter to various images and display the blobs as circles on the image. The Laplacian of Gaussian function is used to detect blobs as it is able to detect a superposition of two egdes (which create a blob) if the size between those two edges is proportional to the standard deviation of the gaussian kernel used. In this way we can detect blobs at varying sizes using various scales of our LoG filter. A visual of the LoG filter is provided in figure 1.

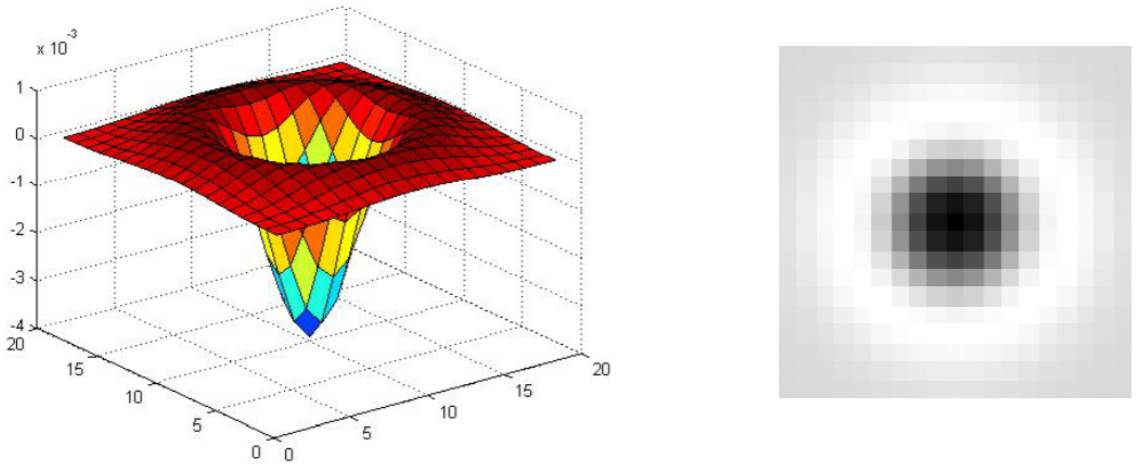

Figure 1: Laplacian of Gaussian

2 Algorithmic Impelmentation

Two different implementations of blob detection were implemented and tested. One implementation increases the blob detection scale through increasing the filter size while the other and faster implementation utilizes re-scaling the image and keeping the filter size constant. These two implementations are compared by computational requirements (run-time).

The faster implementation method used was to keep the filter size constant while creating an image pyramid by re scaling the image. This is much faster because it decreases the total number

of computations required for the convolutions. The scale normalized LoG filter is a symmetric filter with values given by the following:

$$\nabla^2_{norm} g = \sigma^2 \left(\frac{\partial^2 g}{\partial^2 x^2} + \frac{\partial^2 g}{\partial^2 y^2} \right) \tag{1}$$

Which in the case of a gaussian kernel yields the following:

$$(x^2 + y^2 - 2\sigma^2) e^{-(x^2+y^2)/2\sigma^2} \tag{2}$$

First the scale-space is constructed. A minimum value for sigma and maximum value are decided as well as the number of scales. In my implementation I chose to use $\sigma_{min} = 2$, $\sigma_{max} = 50$, and 15 scales. Using these numbers gives us a scale ratio value of $\lambda = 1.2394$. This number will be used to re-scale the images and create the image pyramid. We create the image pyramid by progressively re-scaling the images by a factor of λ^{-i} where i represents the current level of the image pyramid starting at 0. This is illustrated in figure 2.

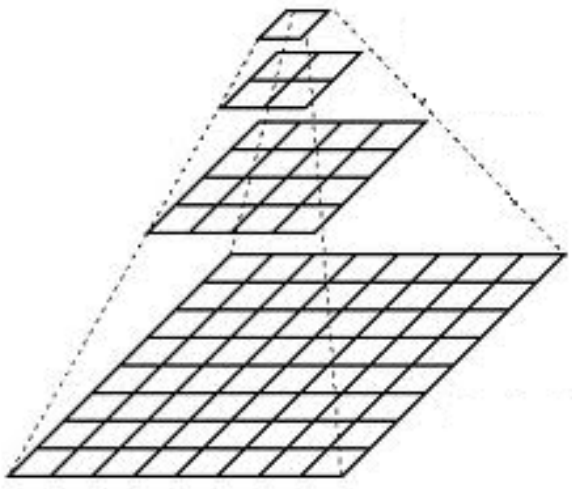

Figure 2: Image Pyramid

To implement this filter we build a symmetric mesh grid of x and y values of size $ceil(3\sigma_{min})/2 \cdot 2 + 1$ in both dimensions. This ensures that the filter size is odd to avoid shifting artifacts. In our case this gave us a filter of size 13×13. We then apply (2) to this mesh grid and multiply by σ^2. We then apply this filter to each scale space image using 2D convolution padding and shape options copy edge and same respectively. After performing the LoG filtering we then rescale the images back to their original size.

After LoG filtering we perform non-max suppression within scales. We implement a simple 3×3 sliding window max filter where the max of the local 3×3 region is assigned to the current pixel under evaluation. We then perform non-max suppression across neighboring scales by assigning the maximum of the pixel in the two neighboring scales to the pixel. We then create a mask by matching the new max value image pyramid to the original image pyramid keeping only the coordinate that match. After that we threshold at a certain value. This threshold value is fine tuned depending on the image being processed and the application needs. We then have a 3-tuple of coordinates representing the x and y position as well as the scale level of each blob. We then convert the scale level, i, to the corresponding value of sigma that it represents by $\sigma_i = \sigma_{min} \cdot \lambda^i$. These σ values correspond to the radii of the blobs by the relationship $r = \sqrt{2}\sigma_i$.

The second and slower implementation is the same as discussed above however rather than rescaling images we rescale the LoG filter.

Figure 3 shows the images used for our testing. We tested the images and give the results of the blob detection with circles in red showing where the blobs are. We perform the blob detection on the images using the faster and more efficient algorithm described previously.

Figure 3: Images tested (from top left to bottom right): (a) butterfly, (b) einstein, (c) fish, (d) sunflowers, (e) circuit board, (f) stars, (g) nighttime satellite, (h) moon.

3 Results

In this section we discuss the results obtained from the blob detection. We can consider our threshold value as a hyperparameter. Figure 4 gives an idea of how the threshold value affects the blobs detected. Obviously the threshold has a major impact on the blobs detected. We found that the optimal value for the threshold for all images was in the range of 0.03 to 0.07.

The rest of the results are included in the attachments of the project submission. Table 1 shows the running times of both implementations of the algorithm with the butterfly image. It is clear that the image rescaling method is far superior with a run time 15% of the other method.

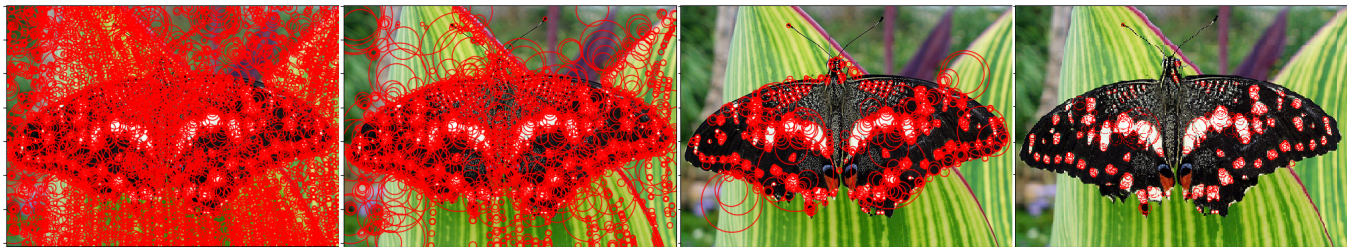

Figure 4: Effect of threshold value on blob detection. Different threshold values from left to right: 0.001, 0.01, 0.05, 0.1.

Method	**Run Time (s)**
Filter Scale	12.65
Image Scale	84.66

Table 1: Running times in seconds for each algorithmic implementation tested on the butterfly image.

4 Running the Code

The main script to run the code and all of the necessary functions are in the "detect_blobs.py" file. The following standard python modules are used: skimage (for image i/o), math, numpy (for array operations), matplotlib.pyplot (for plotting circles), and time (for testing run time). All of these modules can be installed easily through conda or pip if necessary (however these are the standard modules used in class throughout the semester so this should not be needed). The 'butterfly.jpg' file must be in the same directory as the 'detect_blobs.py' script as this is the default file for testing. To run the code fromm the terminal simply run: "python detect_blobs.py" or "python3 detect_blobs.py". Threshold can be changed in the main script in the variable named *thresh*.

References

[1] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.