Lucas Junginger - 0032-4867
02/16/2016

Wherever the run-time is represented in this document, it is in an image of two lines, the first line corresponding to the run-time of the ‘inefficient’ implementation and the second line to the ‘efficient’ implementation. The original images of my picks are also included at the end.

I tried implementing the ‘efficient’ method in a variety of ways and it was always faster but it seemed to perform poorly compared to the ‘inefficient’ implementation. I was having a hard time understanding how to set the kernel size and the value of the factor k by which the image was downsampled each time. I spent a lot of time trying to format my values into a cell of arrays as was hinted in the pdf but was having a hard time converting the values back into an array of doubles to process for non-maximum suppression. Eventually I used a factor k that was low enough for enough small size objects to be detected.

It is pretty clear now my ‘efficient’ implementation is quite buggy because I can see in a most pictures that they lack large objects almost entirely and that many of these objects are superimposed and not exactly surrounding the relevant objects. The ‘inefficient’ implementation did not show these signs to the same extent. The ‘efficient’ implementation did not detect over 1000 blobs unless I tightened the parameters such that it would downscale the image many times around the small blob shapes. Once it got to the large blobs, it seemed that none of my blobs were making any more sense and so I cut the algorithm off once the image got to a diameter of 20 pixels. Additionally, the neighboring algorithm was taking very long to run on the first couple of iterations and was finding under 50 hits for each blob size. To speed things up, I downscaled the image initially by 36/100, which was the third iteration when (k = 6/10), when it finally started detecting hundreds of blobs.

To run non-maximum suppression in 2d slices, I used nlfilter with a custom function that used the radius size as the surrounding area to which each pixel was compared. This turned out to be very slow when the image was large and the kernel was small. as can be seen, all of my images too over 10 seconds to process in both implementations and most of this time was spent “applying neighborhood operations”. Additionally, it did not seem to work completely, especially in my ‘efficient’ implementation.

My ‘inefficient’ implementation had a bit of the opposite problem, as most of the blobs appeared to have a small radius and I believe this can be explained by the way in which I did 3D non-maximum suppression. To run non-maximum suppression across the 3d slices representing different blob sizes, I decided to, at each non-zero pixel at increasing levels of k, compare it to the neighbors determined by its blob radius in the previous level of k. This means that as the blobs were calculated at sequentially higher radii, they were each compared to their lower neighbors, if they were not the local maximum, they would be set to 0. This was implemented based on the assumption that inside a blob of large radius there would be smaller blobs accounting for the detail. However this ignored the fact that blobs more than one level away could be the accurate local maximum. Additionally, this method this not seem to produce any noticeable results in the ‘efficient’ implementation so it was not included.

Setting the threshold was a bit tricky because for my efficient implementation I was not finding blobs with high LoG values even when I could find more than 1000 blobs. For the ‘inefficient’ implementation I also found that the threshold varied within images. For example, the ‘fish.jpg’ image had an average LoG score that was much lower and so a threshold that was good for the other images wouldn’t produce 1000 blobs in

that image. To set a threshold that worked for all images, I discarded LoG values of under 0.01. As expected, this didn't make much of a difference in the images with a large number of objects.

Overall I was impressed with how my algorithms performed. They seemed to do a very good job mapping out regions of interest in the images and I could imagine making use of all that negative space to really characterize the image. Additionally, a more effective way of performing 3D maximum suppression across all blob-scapes could potentially improve the results by a lot, as this could rearrange a large number of the blobs.

Lucas Junginger - 0032-4867
02/16/2016

Blob detection: scale filter

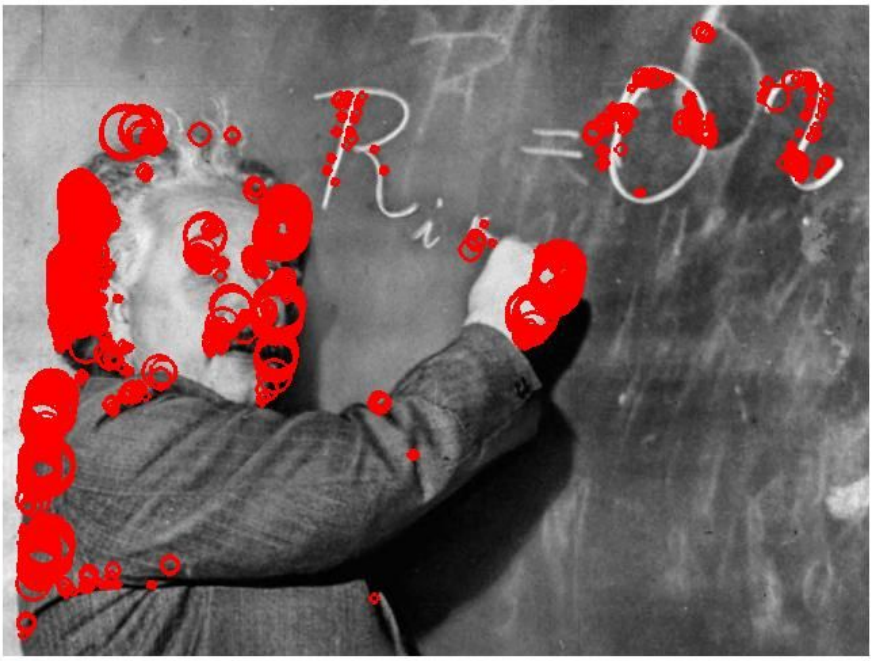
Blob detection: scale image

```
Elapsed time is 49.777426 seconds.
Elapsed time is 17.961725 seconds.
```

Lucas Junginger - 0032-4867
02/16/2016

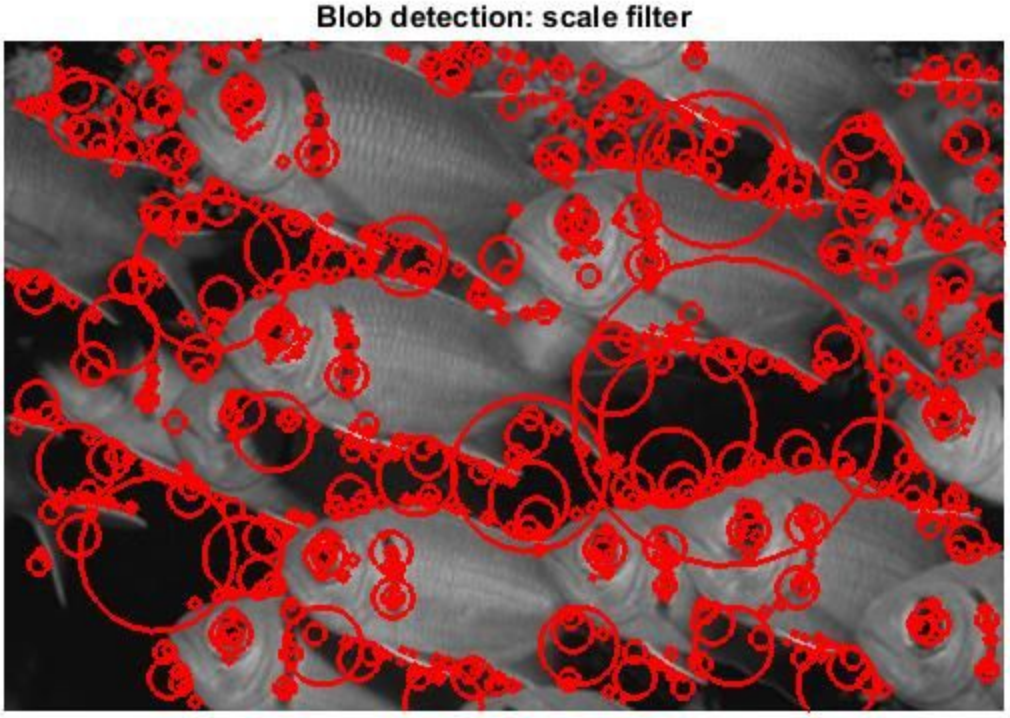
Blob detection: scale filter

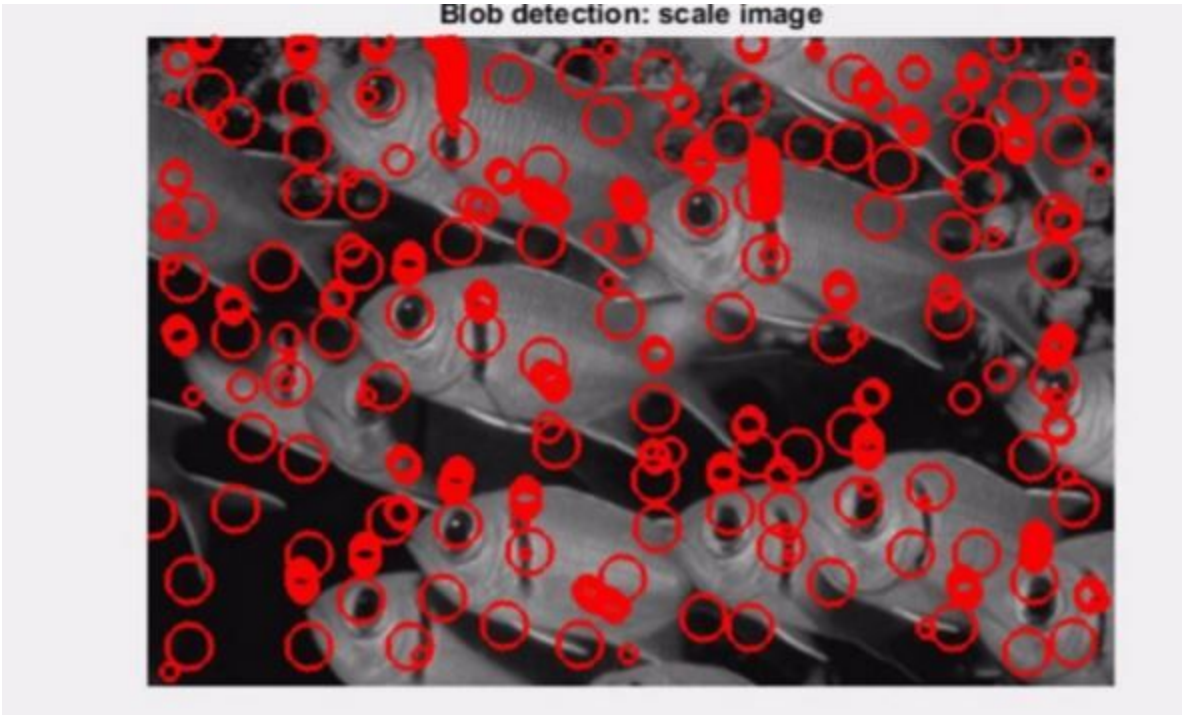
Blob detection: scale image



```
Elapsed time is 75.855241 seconds.
Elapsed time is 31.693864 seconds.
```

Lucas Junginger - 0032-4867
02/16/2016



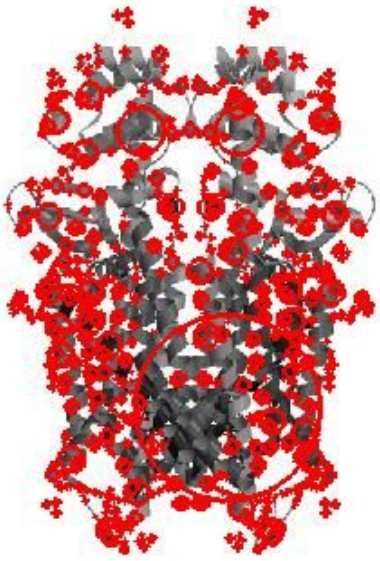



```
Elapsed time is 49.777426 seconds.
Elapsed time is 17.961725 seconds.
```

Lucas Junginger - 0032-4867
02/16/2016

Blob detection: scale image

Blob detection: scale filter



```
Elapsed time is 62.642459 seconds.
Elapsed time is 28.117295 seconds.
```

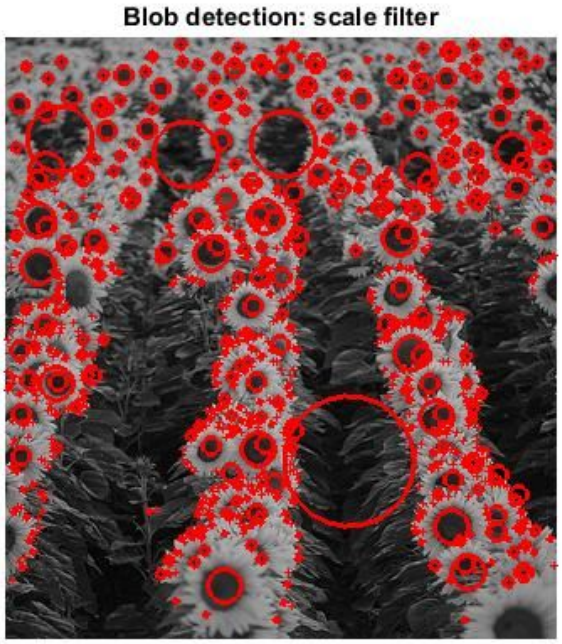
Lucas Junginger - 0032-4867
02/16/2016

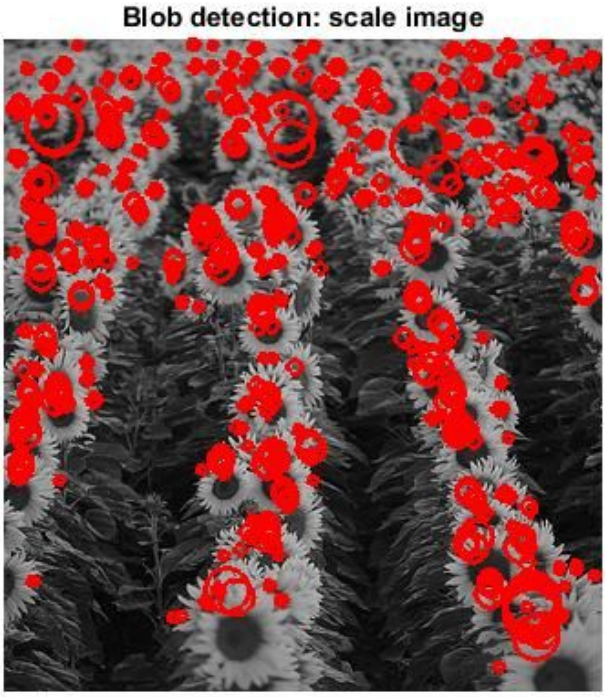
Blob detection: scale filter

Blob detection: scale image

```
Elapsed time is 73.373501 seconds.
Elapsed time is 30.864418 seconds.
```

Lucas Junginger - 0032-4867
02/16/2016






```
Elapsed time is 32.142305 seconds.
Elapsed time is 12.476598 seconds.
```

Lucas Junginger - 0032-4867
02/16/2016

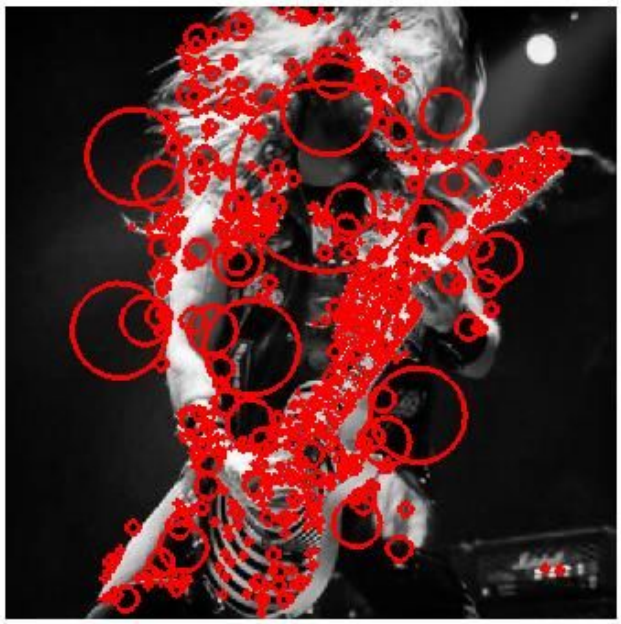
Blob detection: scale filter

Blob detection: scale image

```
Elapsed time is 64.133811 seconds.
Elapsed time is 29.724879 seconds.
```

Lucas Junginger - 0032-4867
02/16/2016

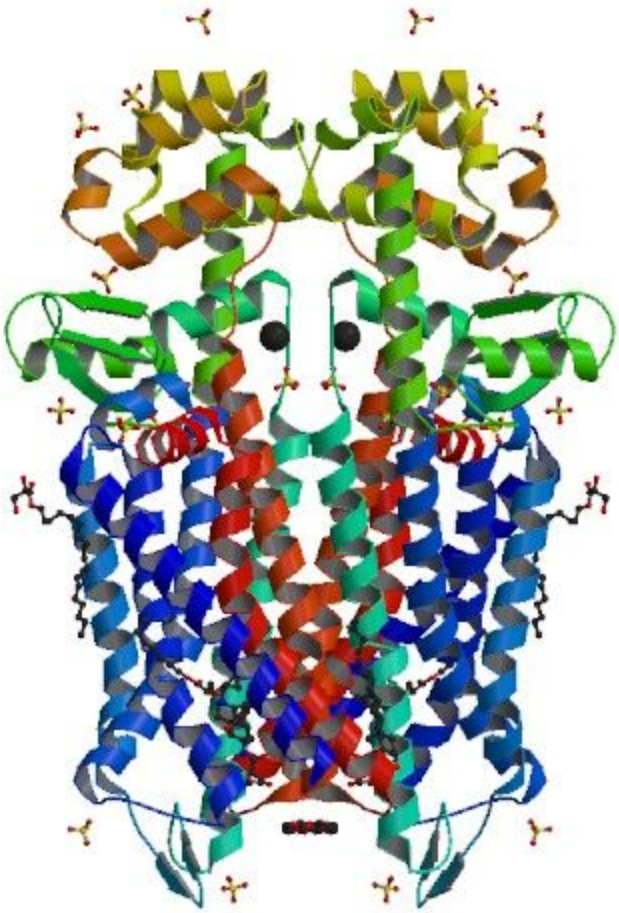
Blob detection: scale filter



Blob detection: scale image

```
Elapsed time is 73.501481 seconds.
Elapsed time is 29.089738 seconds.
```

Lucas Junginger - 0032-4867
02/16/2016



Lucas Junginger - 0032-4867
02/16/2016

Lucas Junginger - 0032-4867
02/16/2016

Lucas Junginger - 0032-4867
02/16/2016