# ECE549 Computer Vision: Assignment 2

Yutong Xie

NetID: yutongx6

Instructor: Prof. Saurabh Gupta

March 12, 2020

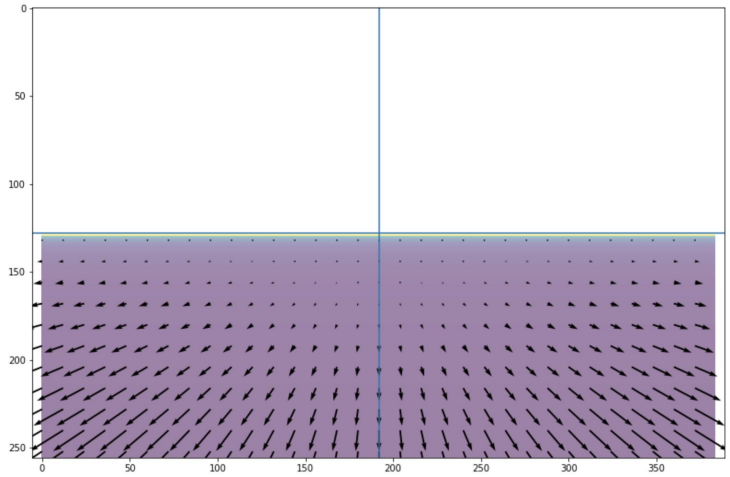
## 1. Question 1b

There are totally five situations, and I show the chosen scene, t, and w with optical flow below.

Situation 1: Looking forward on a horizontal plane while driving on a flat road

Scene: road, t = [[0],[0],[1]], w = [[0],[0],[0]]

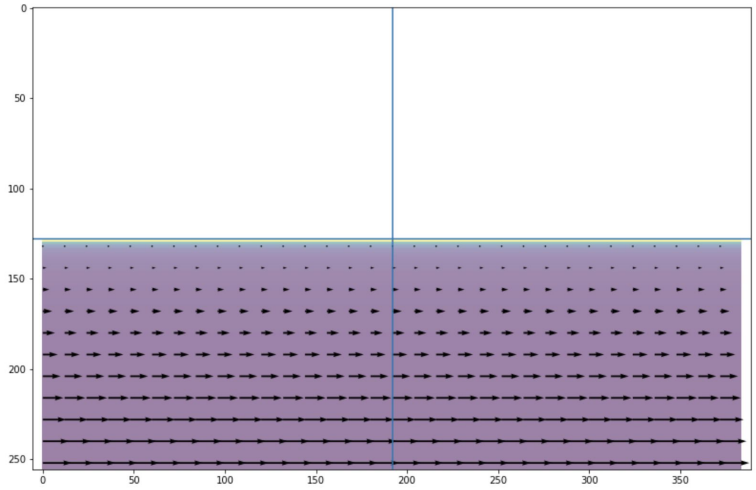
Optical flow：




Situation 2: Sitting in a train and looking our over a flat field from a side window.

Scene: road, t = [[1],[0],[0]], w = [[0],[0],[0]]

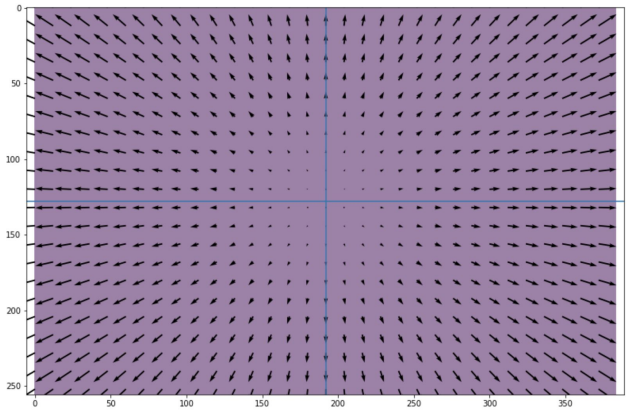
Optical flow：



Situation 3: Flying into a wall head-on

Scene: wall, t = [[0],[0],[1]], w = [[0],[0],[0]]

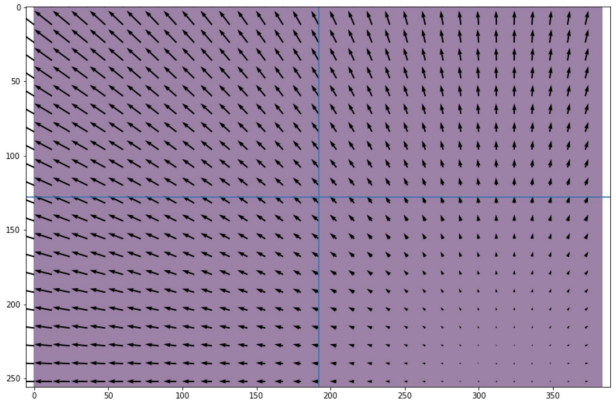
Optical flow：




Situation 4: Flying into a wall but also translating horizontally and vertically

Scene: road, t = [[1],[1],[1]], w = [[0],[0],[0]]

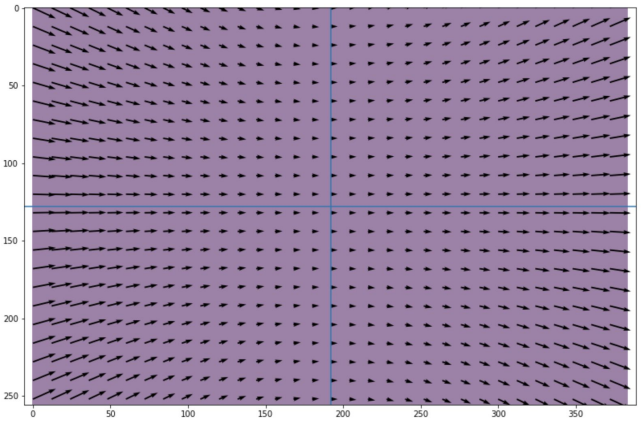
Optical flow：




Situation 5: Counter-clockwise rotating in front of a wall about the Y-axis

Scene: road, t = [[0],[0],[0]], w = [[0],[-1],[0]]

Optical flow：



## 2. Question 2 Contour Detection

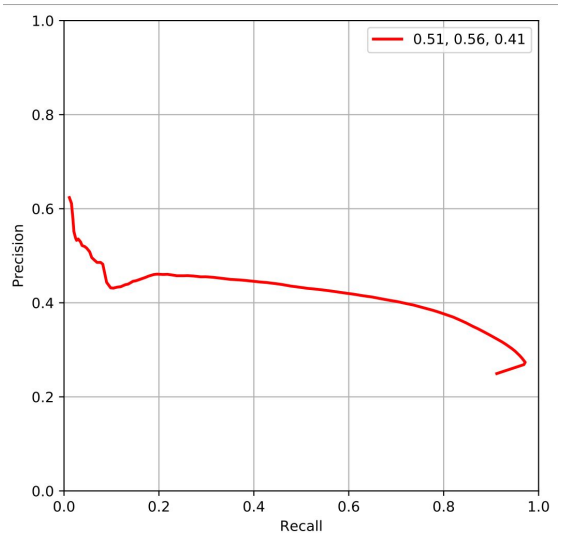
2.1 Warm-up

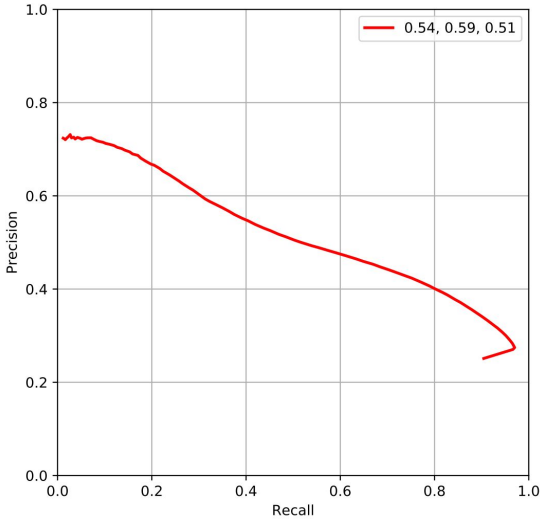
After run the raw code, I found that there are some artifacts at image coundaries. I think the artifacts are generated when we execute the convolution till the boundary. Hence, I add a parameter to the *convolve2d* function which is **boundary='symm'.** The parameter means we take symmetrical boundary conditions.

Before and after the modification, the quality performance metrics are shown below. Left one is before modification, right one is after modification

```
          threshold: 0.220000
overall max F1 score: 0.514369
average max F1 score: 0.562687
            area_pr: 0.408983
```

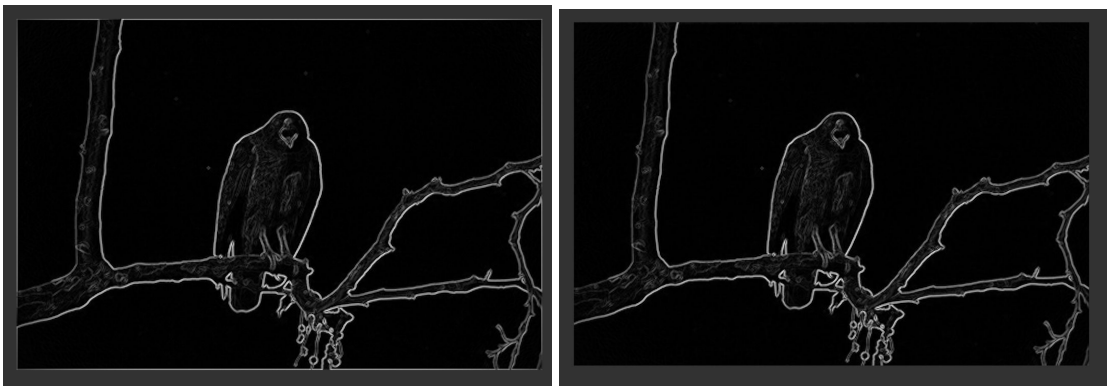
```
          threshold: 0.240000
overall max F1 score: 0.542432
average max F1 score: 0.587287
            area_pr: 0.509132
```




Before modification, the running time is 1.17104s. After modification, running time is 1.33008s. There is no huge difference between these two.

Some examples show here.



2.2 Smoothing

In this part, I use derivative of Gaussian filters to obtain more robust estimates of the gradient. The code is shown below.

```
w = 5
sigma = 10
t = (((w - 1)/2)-0.5)/sigma
dx = ndi.gaussian_filter(I,sigma,order=[1,0],truncate=t) # x Derivative
dy = ndi.gaussian_filter(I,sigma,order=[0,1],truncate=t) # y Derivative
```

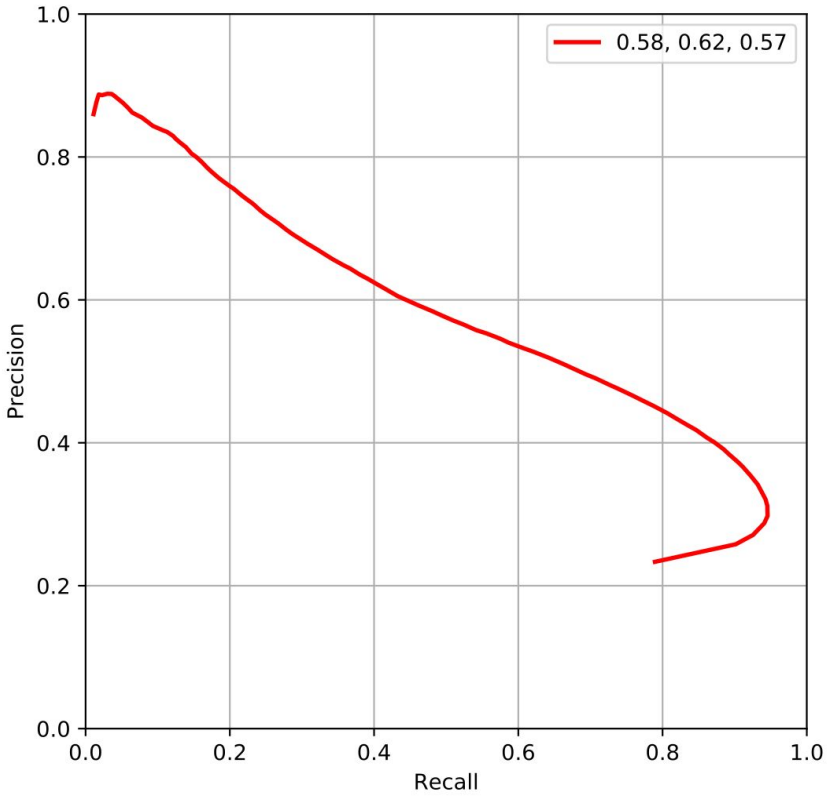
W is width of Gaussian kernel, and we can use sigma and w to calculate the parameter truncate. I use several different w and sigma to get an optimal one, with 5 for w and 10 for sigma.

| sample size 10 | filter size 3x3 | | | | |
|---|---|---|---|---|---|
| Gaussian filter sigma | 0.5 | 1 | 2 | 5 | 10 |
| overall max F1 score | 0.519439 | 0.524348 | 0.534787 | 0.535054 | 0.535086 |
| average max F1 score | 0.49695 | 0.503932 | 0.512582 | 0.510921 | 0.511786 |
| area_pr | 0.470093 | 0.485455 | 0.500502 | 0.501565 | 0.50178 |

| sample size 10 | filter size 5x5 | | | | | |
|---|---|---|---|---|---|---|
| Gaussian filter sigma | 0.5 | 1 | 2 | 5 | **10** | 20 |
| overall max F1 score | 0.519847 | 0.552547 | 0.552355 | 0.553735 | **0.553608** | 0.553744 |
| average max F1 score | 0.497824 | 0.565733 | 0.559566 | 0.56077 | **0.562456** | 0.561811 |
| area_pr | 0.470743 | 0.524094 | 0.52412 | 0.525657 | **0.52609** | 0.525842 |

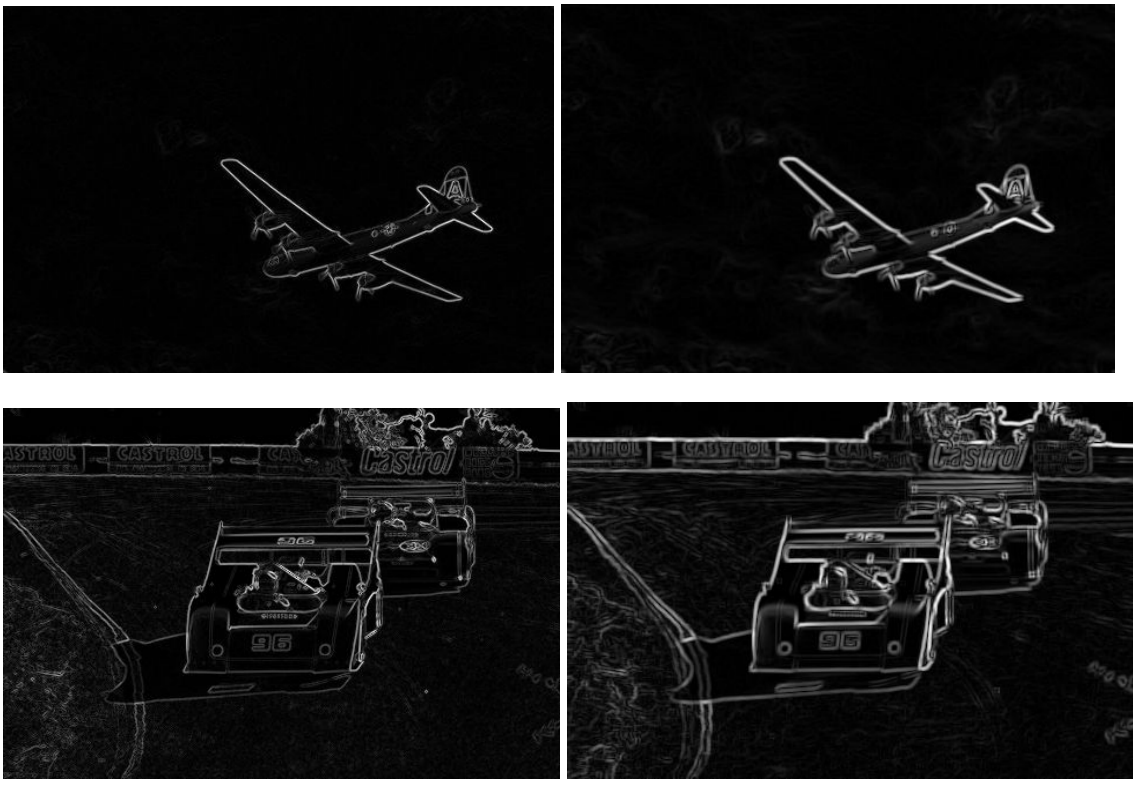
After this modification, the performance metrics is shown below.

```
          threshold: 0.260000
overall max F1 score: 0.579074
average max F1 score: 0.616193
             area_pr: 0.566543
```




After smoothing, the running time is 1.020929s, which has no huge difference with the time before modification.

Some example show here.



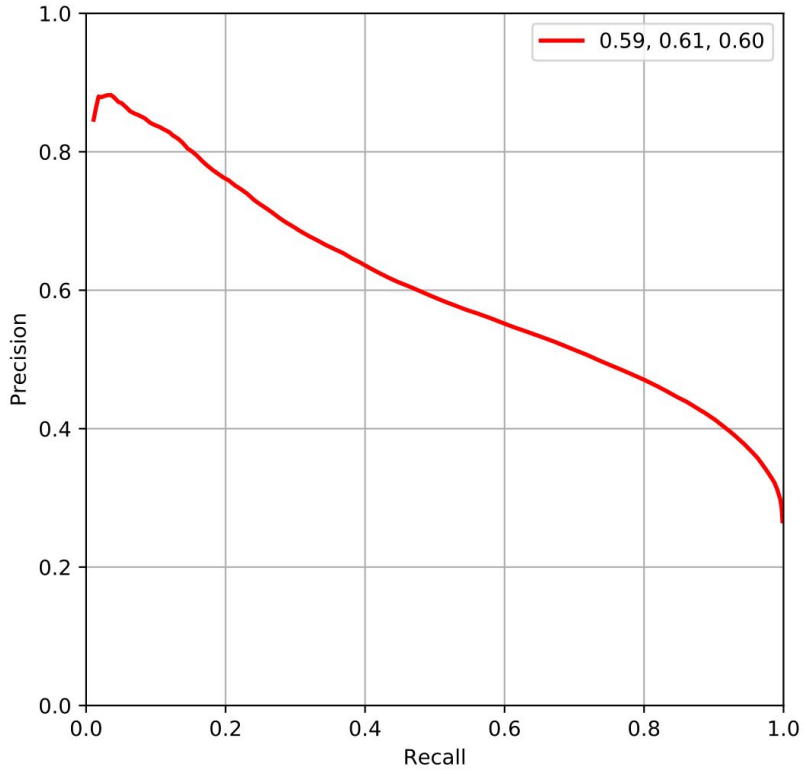
2.3 Non-maximum Suppression

In this part, I realized the non-maximum suppression to product thin edges. First, I use dx and dy to calculate the orientation of gradient. Then, I find two points on and opposite the direction of gradient and do the interpolation. Finally, if the point is not maximum, it will be set as 0. The code is shown below.

```
threshold = 0
for y in range(1, mag.shape[0]-1):
    for x in range(1, mag.shape[1]-1):
        if mag[y][x] > threshold:
            angle = theta[y][x]
            if (0 <= angle < 45):
                w = abs(dy[y][x])/abs(dx[y][x])
                p = w * mag[y-1][x-1] + (1-w) * mag[y][x-1]
                r = w * mag[y+1][x+1] + (1-w) * mag[y][x+1]

            elif (45 <= angle <= 90):
                w = abs(dx[y][x])/abs(dy[y][x])
                p = w * mag[y-1][x-1] + (1-w) * mag[y-1][x]
                r = w * mag[y+1][x+1] + (1-w) * mag[y+1][x]

            elif (90 < angle < 135):
                w = abs(dx[y][x])/abs(dy[y][x])
                p = w * mag[y-1][x+1] + (1-w) * mag[y-1][x]
                r = w * mag[y+1][x-1] + (1-w) * mag[y+1][x]

            elif (135 <= angle <= 180):
                w = abs(dy[y][x])/abs(dx[y][x])
                p = w * mag[y-1][x+1] + (1-w) * mag[y][x+1]
                r = w * mag[y+1][x-1] + (1-w) * mag[y][x-1]
            if mag[y][x] >= p and mag[y][x] >= r:
                # NMS[y][x] = mag[y][x]
                continue
            else:
                mag[y][x] = 0
```

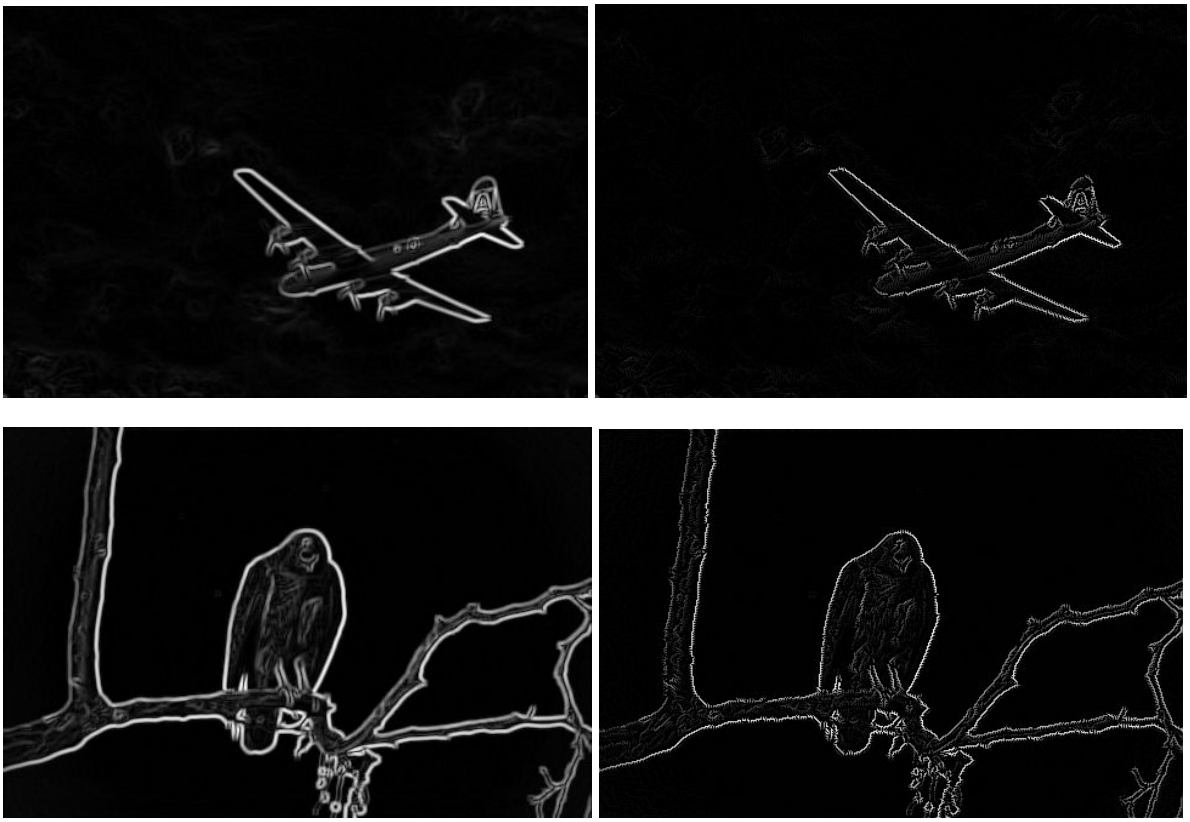
The perforamnce metrics is here.

```
           threshold: 0.240000
overall max F1 score: 0.594514
average max F1 score: 0.606784
             area_pr: 0.600951
```




The running time is 193.40675s after NMS, because all the pixels needed to be judge.

The example is here.



## 3. Question 3: Scale-space blob detection

3.1 Basic Implementation

In this part, I choose two methods for NMS.

The first one is using generic filter.

*nms_3d = generic_filter(scale_space, nms, size = (3,3,3))*

The scale space is input array, nms is the function I defined to check whether the pixel is maximum value among the neighbourhoods, and also I set the size as (3,3,3).

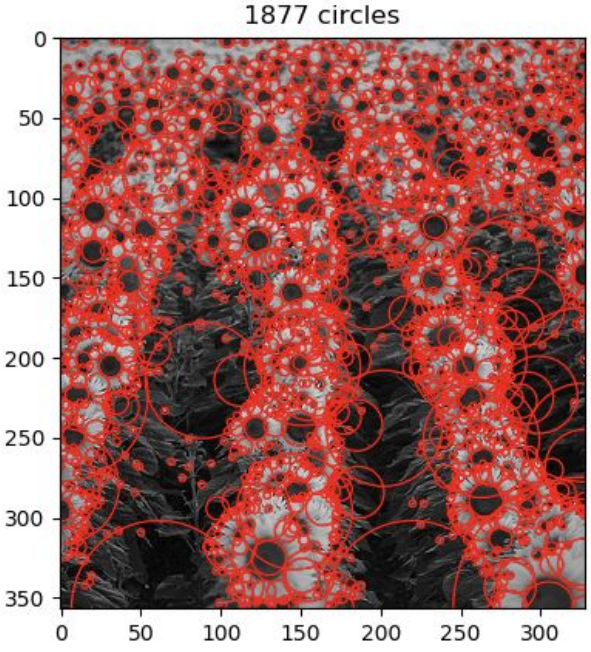
The second one is using rank filter.

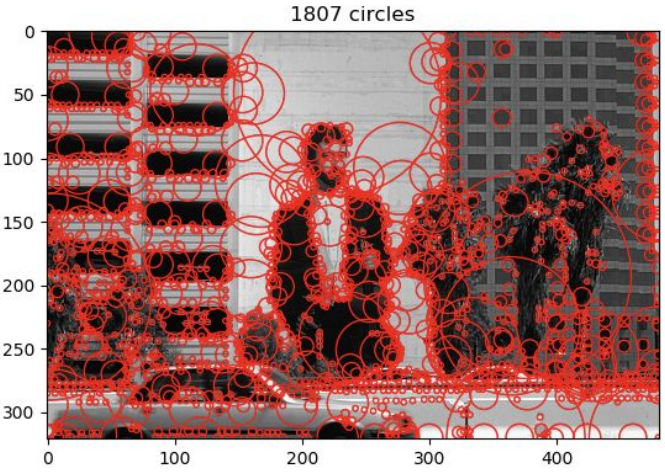
```
for i in range(level):
    nms_2d[:,:,i] = rank_filter(scale_space[:,:,i], -1, (3,3))
```

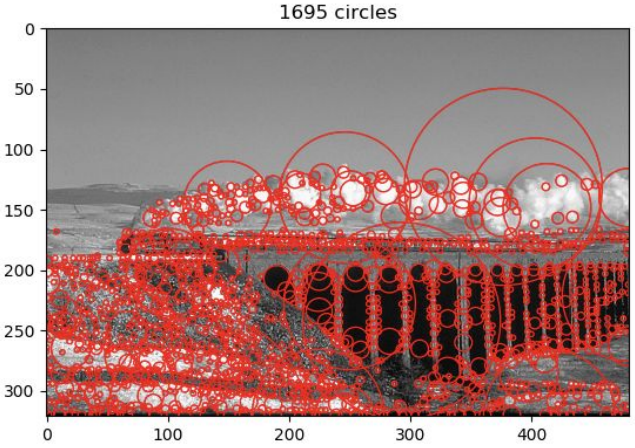
Same with generic filter, scale space is the input, -1 means we want to get the maximum value, and (3,3) indicates the region we want to compare.

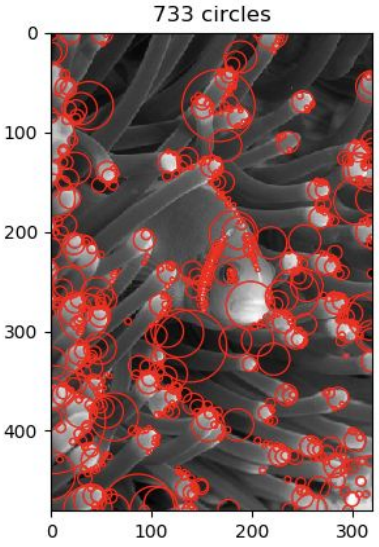
I tried several different initial sigma, different k, and diferent scale levels. And I found that sigma = 1.5, k = 1.5, level = 12 would give a good detection. And the threshold is set to 0.01. If I increase the threshold, the number of circles will become less, but if i decrease the threshold, there are too many circles.

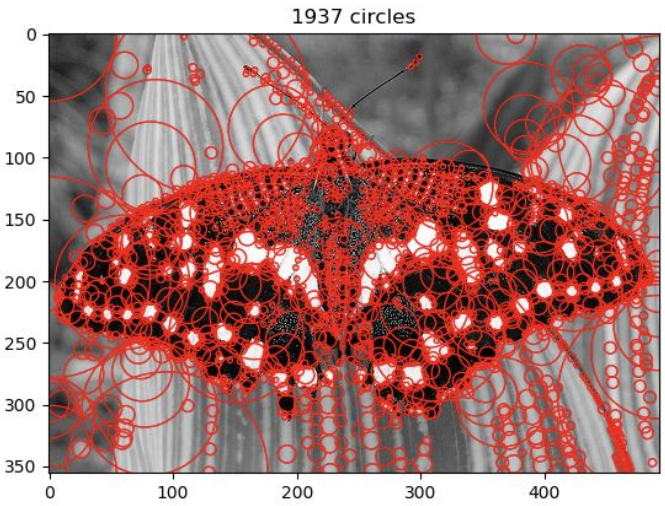
Here are eight outputs.

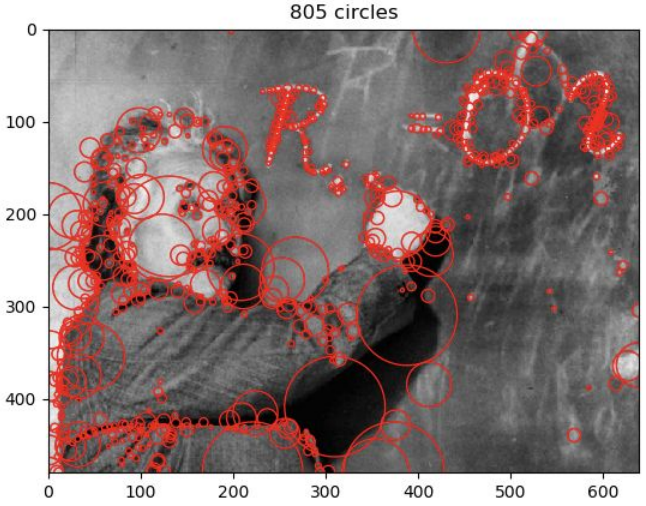


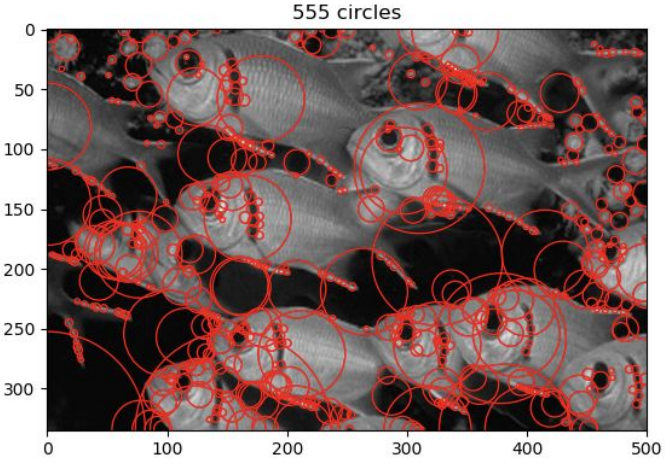

1807 circles
0
50
100
150
200
250
300
0
100
200
300
400

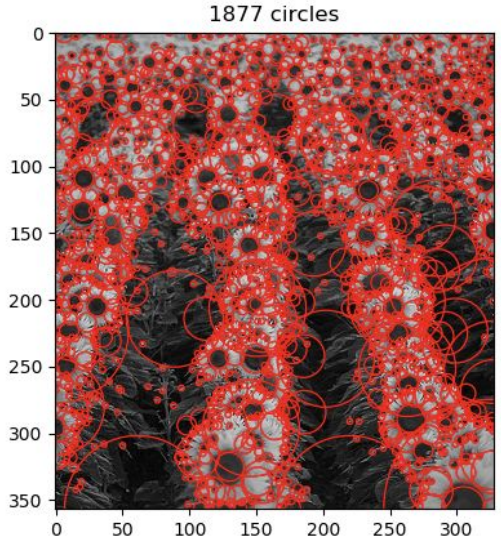

1695 circles
0
50
100
150
200
250
300
0
100
200
300
400


733 circles
0
100
200
300
400
0
100
200
300


1937 circles
0
50
100
150
200
250
300
350
0
100
200
300
400


805 circles
0
100
200
300
400
0
100
200
300
400
500
600


555 circles
0
50
100
150
200
250
300
0
100
200
300
400
500

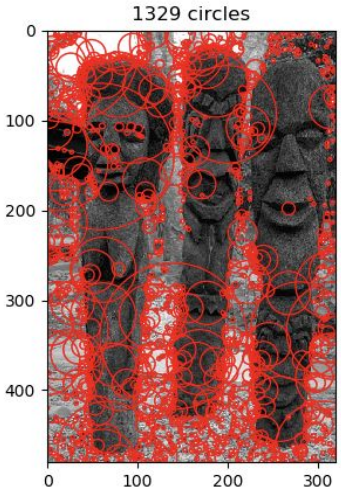


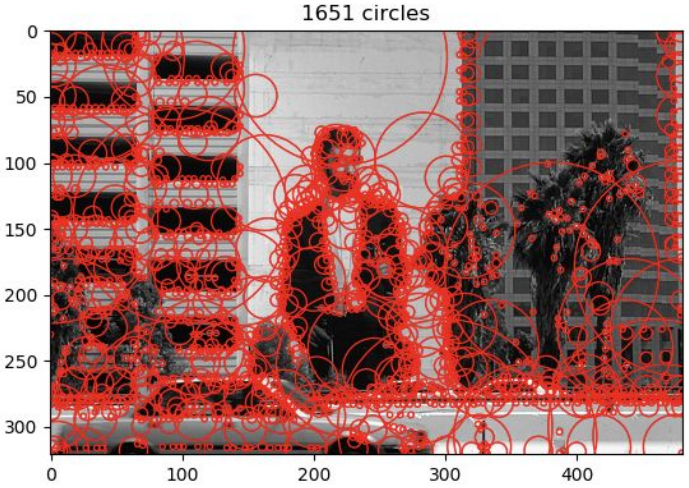

3.2 Efficient Implementation

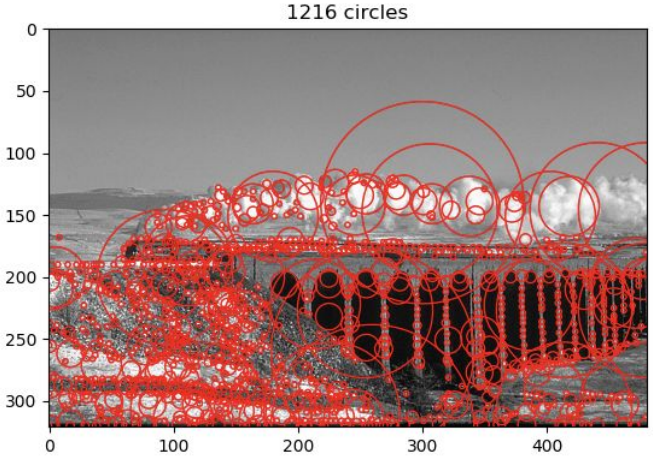
In this part, I downsample the image by 1/k instead of increasing the kernel size. For each level, I use resize function to change the shape of image array, and recover it after we apply the gaussian laplace filter. For the efficient method, I choose 0.003 as the threshold, because there are only few points satisfied threshold 0.01. The running time for basic and efficient implementation. We can see that the time for down sample method is similar to the basic implementation. However, if we set their thresholds equal, then the down sample method needs less time and it is more efficient.

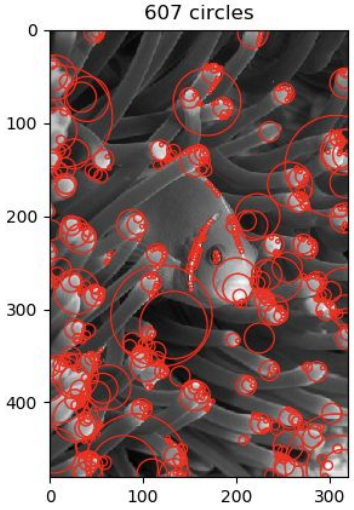
```
MacBook-Pro:mp2-release yutong$ python blobs_show_all_circles.py
running time for up sample is: 7.712148189544678 s.
running time for down sample is: 7.180080890655518 s.
running time for up sample is: 6.716811895370483 s.
running time for down sample is: 7.333600044250488 s.
running time for up sample is: 7.016278028488159 s.
running time for down sample is: 7.403585195541382 s.
running time for up sample is: 6.524410009384155 s.
running time for down sample is: 6.456428050994873 s.
running time for up sample is: 9.31346321105957 s.
running time for down sample is: 8.154247045516968 s.
running time for up sample is: 11.586487054824829 s.
running time for down sample is: 11.557905912399292 s.
running time for up sample is: 5.765262126922607 s.
running time for down sample is: 5.823497772216797 s.
running time for up sample is: 5.968029260635376 s.
running time for down sample is: 5.737262010574341 s.
```

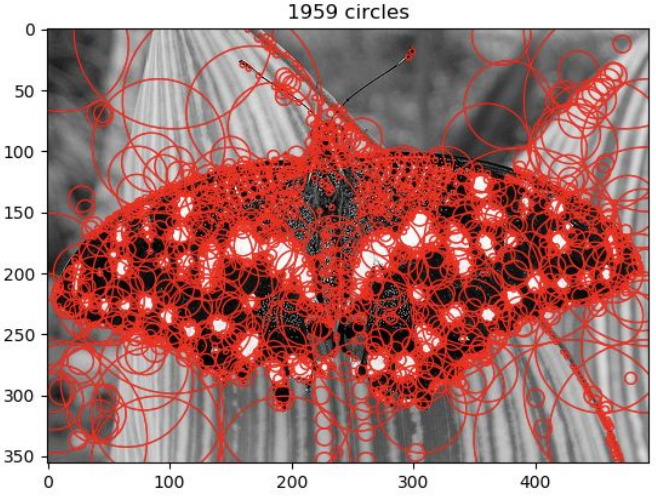
Here is eight outputs for efficient implementation.

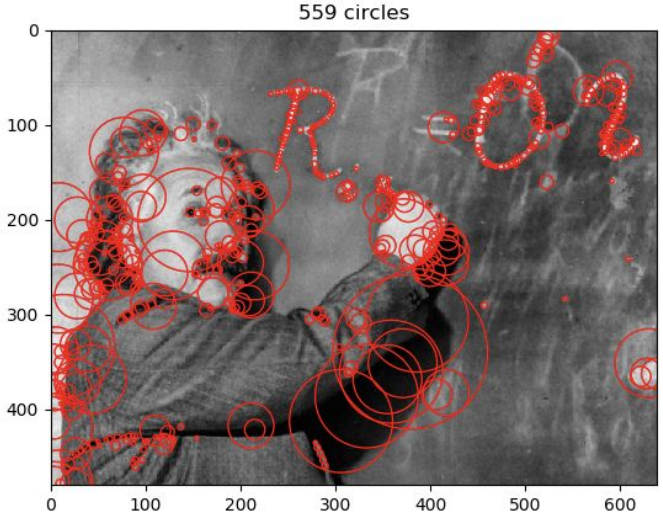

1329 circles
0
100
200
300
400
0
100
200
300

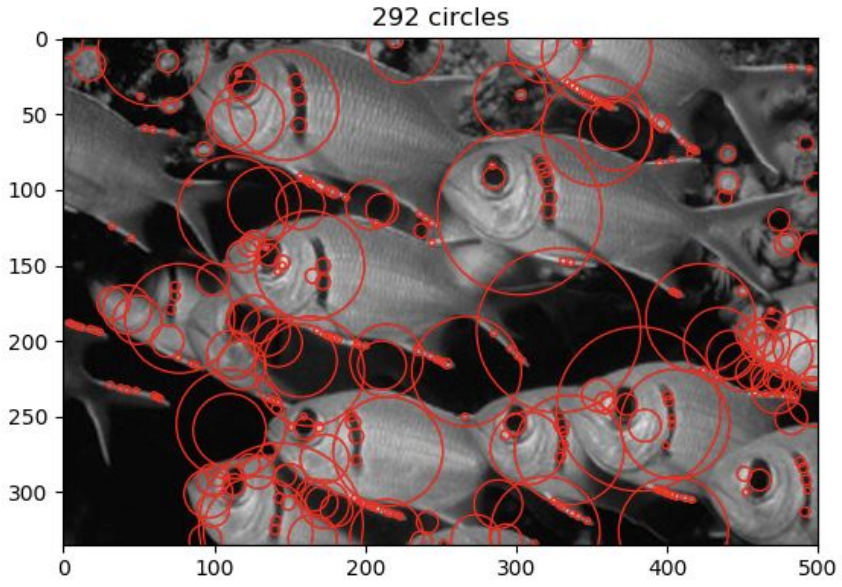

1651 circles
0
50
100
150
200
250
300
0
100
200
300
400

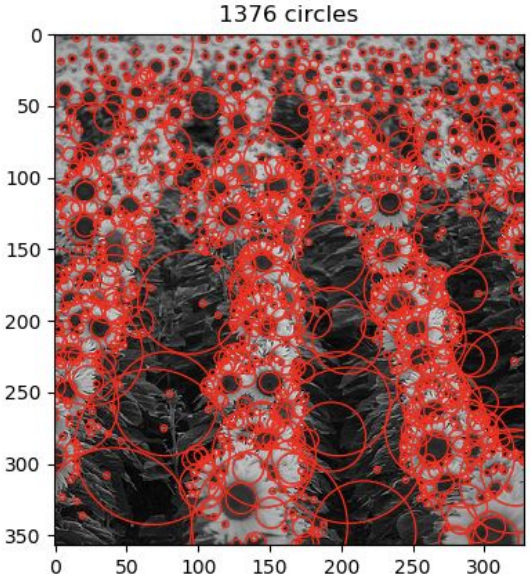

1216 circles
0
50
100
150
200
250
300
0
100
200
300
400


607 circles
0
100
200
300
400
0
100
200
300


1959 circles
0
50
100
150
200
250
300
350
0
100
200
300
400


559 circles
0
100
200
300
400
0
100
200
300
400
500
600


292 circles
0
50
100
150
200
250
300
0
100
200
300
400
500


1376 circles
0
50
100
150
200
250
300
350
0
50
100
150
200
250
300

1. **Dynamic Perspective [20 pts].** In this question, we will simulate optical flow induced on the image of a static scene due to camera motion. You can review these concepts from lecture 5.

   (a) **[10 pts]** Assuming a camera moving with translation velocity of $\mathbf{t}$ and angular velocity of $\omega$. Derive the equation that governs the optical flow at a pixel $(x, y)$ in terms of the focal length $f$, and the depth of the point $Z(x, y)$. Note that a point $(X, Y, Z)$ in the world projects to $\left(f\frac{X}{Z}, f\frac{Y}{Z}\right)$ in the image.

$$\dot{P} = -t - \omega \times P$$

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = -\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} - \begin{bmatrix} \omega_y Z - \omega_z Y \\ \omega_z X - \omega_x Z \\ \omega_x Y - \omega_y X \end{bmatrix}$$

$$x = f\frac{X}{Z} \qquad y = f\frac{Y}{Z}$$

$$\dot{x} = \frac{f(\dot{X}Z - \dot{Z}X)}{Z^2} \qquad \dot{y} = \frac{f(\dot{Y}Z - \dot{Z}Y)}{Z^2}$$

$$\dot{x} = f\,\frac{((-t_x - \omega_y Z + \omega_z Y)Z - (-t_z - \omega_x Y + \omega_y X)X)}{Z^2}$$

$$= f\left(\frac{-t_x - \omega_y Z + \omega_z Y}{Z} + \frac{t_z + \omega_x Y - \omega_y X}{Z} \cdot \frac{X}{Z}\right)$$

$$= \frac{1}{Z}\begin{bmatrix} -f & 0 & x \end{bmatrix}\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \begin{bmatrix} \frac{xy}{f} & -(f + \frac{x^2}{f}) & y \end{bmatrix}\begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

similar,

$$\dot{y} = \frac{1}{Z}\begin{bmatrix} 0 & -f & y \end{bmatrix}\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \begin{bmatrix} (f + \frac{y^2}{f}) & -\frac{xy}{f} & -x \end{bmatrix}\begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

$$\therefore \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \frac{1}{Z}\begin{bmatrix} -f & 0 & x \\ 0 & -f & y \end{bmatrix}\begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \begin{bmatrix} \frac{xy}{f} & -(f + \frac{x^2}{f}) & y \\ (f + \frac{y^2}{f}) & -\frac{xy}{f} & -x \end{bmatrix}\begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$