Assignment Report

ENGN6528

Yifan Zhu
u7560434

08/05/2023

COMP-ENGN-6528-Computer Vision Assignment

Q1. The following is a separable filter. What does it mean to be a separable filter?(0.5 mark) Write down the separate components of the following filter. (1 marks)

In image processing, a separable filter can be expressed as the result of two more simple filters. Usually, a 2-dimensional convolution operation is separated into two 1-dimensional filters.

For the given filter, it can be separated as shown in Formula 1.

$$\begin{bmatrix} 4 & 4 & 6 \\ 4 & 4 & 6 \\ 6 & 6 & 9 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 3 \end{bmatrix} * \begin{bmatrix} 2 & 2 & 3 \end{bmatrix} \tag{1}$$

Q2. Calculate the value of the blue patch in Fig. 2 using bilateral filtering. Assume the Domain kernel is of size 5×5, the standard deviation $\sigma_d = 2$, provided as $\begin{pmatrix} 0.0232 & 0.0338 & 0.0383 & 0.0338 & 0.0232 \\ 0.0338 & 0.0492 & 0.0558 & 0.0492 & 0.0338 \\ 0.0383 & 0.0558 & 0.0632 & 0.0558 & 0.0383 \\ 0.0338 & 0.0492 & 0.0558 & 0.0492 & 0.0338 \\ 0.0232 & 0.0338 & 0.0383 & 0.0338 & 0.0232 \end{pmatrix}$ the Range kernel is of size 5×5 and the standard deviation $\sigma_r = 50$. Please 1) provide the range filter associated to the pixel high-lighted in the Fig. 2 (2 marks), and 2) show the filtered value for the high-lighted pixel (2 marks).

128	128	100	100	103	50
150	100	120	30	53	54
150	112	127	40	35	20
132	125	112	43	20	10
133	130	100	30	10	20
140	130	120	20	10	20

Figure 2: Image Patch

In order to calculate the value using bilateral filtering, we use the method shown in Formula 2.

$$BF[I]_p = \frac{1}{W_{\vec{p}}} \sum_{\vec{q} \in S} G_{\sigma_s}(\parallel \vec{p} - \vec{q} \parallel) G_{\sigma_r}(\parallel I_{\vec{p}} - I_{\vec{q}} \parallel) I_{\vec{q}} \tag{2}$$

As a given result $G_{\sigma_s}(\parallel \vec{p} - \vec{q} \parallel)$, now we need to calculate $G_{\sigma_r}(\parallel I_{\vec{p}} - I_{\vec{q}} \parallel)$. The method of calculating $G_{\sigma_r}(\parallel I_{\vec{p}} - I_{\vec{q}} \parallel)$ is shown in Formula 3.

$$r(i,j,k,l) = e^{-\frac{\parallel f(i,j) - f(k,l) \parallel^2}{2\sigma_r^2}} \tag{3}$$

Where the range kernel *r* represents the absolute value of the difference between the grey-scale value *f*(*k*, *l*) of a point (*k*, *l*) in the neighbourhood and the grey-scale value f(*i*, *j*) of the centre point (*i*, *j*).

The result of the range filter is shown in Formula 4.

$$\begin{bmatrix} 0.7492 & 0.9716 & 0.9873 & 0.2606 & 0.4985 \\ 0.7492 & 0.0000 & 0.9560 & 0.3546 & 0.3055 \\ 0.9231 & 0.9668 & 0.0000 & 0.3859 & 0.1840 \\ 0.9156 & 0.9372 & 0.9716 & 0.2606 & 0.1248 \\ 0.8548 & 0.9372 & 0.9873 & 0.1840 & 0.1248 \end{bmatrix} \tag{4}$$

Then we calculate $G_{\sigma_s}(\parallel \vec{p} - \vec{q} \parallel) * G_{\sigma_r}(\parallel I_{\vec{p}} - I_{\vec{q}} \parallel)$ to get a new matrix called weight W_p in Formula 5.

$$\begin{bmatrix} 0.0174 & 0.0328 & 0.0378 & 0.0088 & 0.0116 \\ 0.0253 & 0.0000 & 0.0538 & 0.0174 & 0.0103 \\ 0.0354 & 0.0539 & 0.0000 & 0.0215 & 0.0070 \\ 0.0309 & 0.0461 & 0.0542 & 0.0128 & 0.0042 \\ 0.0198 & 0.0316 & 0.0378 & 0.0050 & 0.0029 \end{bmatrix} \tag{5}$$

Multiply the Wp of each point by the pixel value I(k, l) of the point as shown in Formula 6 and sum it as a molecule.

$$\begin{bmatrix} 2.6100 & 3.2800 & 4.5360 & 0.2640 & 0.6148 \\ 3.7950 & 0.0000 & 6.8326 & 0.6960 & 0.3605 \\ 4.6728 & 6.7375 & 0.0000 & 0.9245 & 0.1400 \\ 4.1097 & 5.9930 & 5.4200 & 0.3840 & 0.0420 \\ 2.7720 & 4.1080 & 4.5360 & 0.1000 & 0.0290 \end{bmatrix} \tag{6}$$

Add the Wp of each point as the denominator and divide the two to get the pixel value of the desired output image's centre point (i, j) as shown in Formula 7.

$$\frac{62.9574}{0.5782} = 108.89 \tag{7}$$

Now we can verify our outcome using code below provided by tutorial in Figure 2.1 and result shown in Figure 2.2.

```
import numpy as np
import cv2

I = np.array([[150, 100, 120, 30, 53],
              [150, 112, 127, 40, 35],
              [132, 125, 112, 43, 20],
              [133, 130, 100, 30, 10],
              [140, 130, 120, 20, 10]])

domain_filter = cv2.getGaussianKernel(5, 2)
domain_filter = np.multiply(domain_filter.T, domain_filter)
range_filter = np.exp(-0.5 * (I-112) ** 2 / 50 ** 2)
weight = range_filter * domain_filter
weight = weight / weight.sum()
filter_value = (I * weight).sum().astype('uint8')

print(range_filter)
print(filter_value)
```

Figure 2.1 Bilateral filtering code

```
[[0.74916202 0.97161077 0.98728157 0.26059182 0.49847592]
 [0.74916202 1.         0.95599748 0.35458755 0.30550168]
 [0.92311635 0.96676484 1.         0.38589113 0.18400359]
 [0.91557774 0.9372549  0.97161077 0.26059182 0.12483031]
 [0.85487502 0.9372549  0.98728157 0.18400359 0.12483031]]
109
```

Figure 2.2 Output value for desired pixel

Q3. Contour Detection [10 marks+5 marks(extra)]

Acknowledgement: Lab material with code are adapted from the one by Professor Saurabh Gupta from UIUC, copyright by UIUC.

In this problem we will build a basic contour detector. We will work with some images from the BSDS dataset (see [1]), and benchmark the performance of our contour detector against human annotations. You can review the basic concepts from lecture on edge detection (Week 3). We will generate a per-pixel boundary score. We will start from a very simple edge detector that simply uses the gradient magnitude of each pixel as the boundary score. We will add non-maximum suppression, image smoothing, and optionally additional bells and whistles. We have provided some starter code, images from the BSDS dataset and the evaluation code. Note that we are using a faster approximate version of the evaluation code, so metrics here won't be directly comparable to ones reported in papers.

Preliminaries. Download the starter code, images and evaluation code from wattle (Assignment.zip)(see contour–data, contour demo.py/contour demo.m). The code has implemented a contour detector that uses the magnitude of the local image gradient as the boundary score. This gives us overall max F-score, average max F-score and AP (average precision) of 0.51, 0.56, 0.41 respectively. Reproduce these results by running contour demo.py/contour demo.m. Confirm your setup by matching these results. Note that the matlab version may be with 0.01 difference from the python code due to the difference in inbuilt functions.

When you run contour demo.py/contour demo.m, it saves the output contours in the folder outputdemo, prints out the 3 metrics, and produces a precision–recall plots at contour–output/demo pr.pdf. Overall max F–score is the most important metric, but we will look at all three.

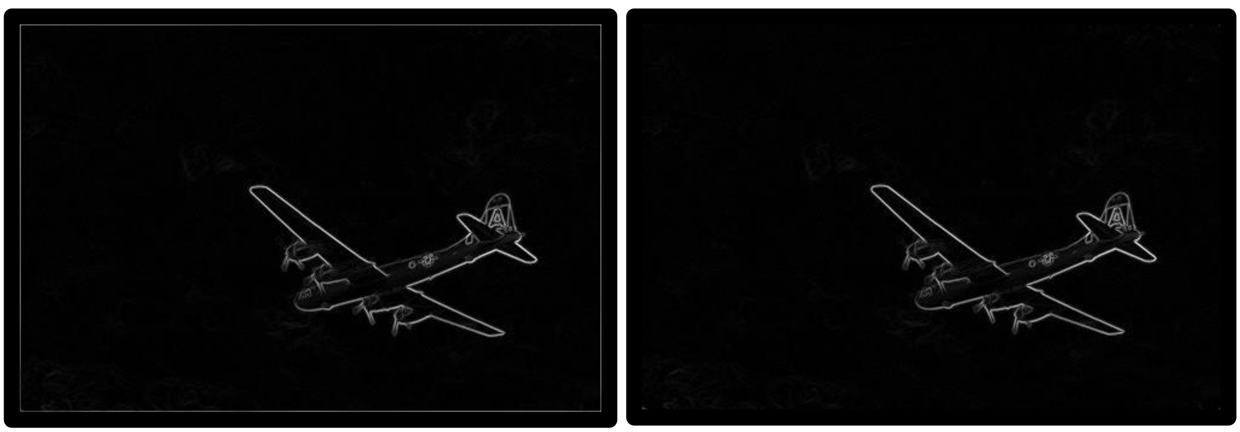
- **Warm-up. As you visualize the produced edges, you will notice artifacts at image boundaries. Modify how the convolution is being done to minimize these artifacts. (1 mark)**

Using padding before convolution might be one method to reduce the artefacts at image boundaries. By extending the image in all directions and enclosing it in a border of zeros, the padding will lessen the effect of the convolution at the image's edges. In this way, we can add boundary = "symm" into the function compute_edges_dxdy(I). As shown in Figure 3.1.

```
def compute_edges_dxdy(I):
    """Returns the norm of dx and dy as the edge response function."""
    I = I.astype(np.float32) / 255.

# adding the boundary argument with the value 'symm'. This will make the
convolution use 'symmetric' padding
    dx = signal.convolve2d(I, np.array([[-1, 0, 1]]), mode='same',
boundary="symm")
    dy = signal.convolve2d(I, np.array([[-1, 0, 1]]).T, mode='same',
boundary="symm")

    # dx = signal.convolve2d(I, np.array([[-1, 0, 1]]), mode='same')
    # dy = signal.convolve2d(I, np.array([[-1, 0, 1]]).T, mode='same')

    mag = np.sqrt(dx ** 2 + dy ** 2)
    mag = mag / np.max(mag)
    mag = mag * 255.
    mag = np.clip(mag, 0, 255)
    mag = mag.astype(np.uint8)
    return mag
```

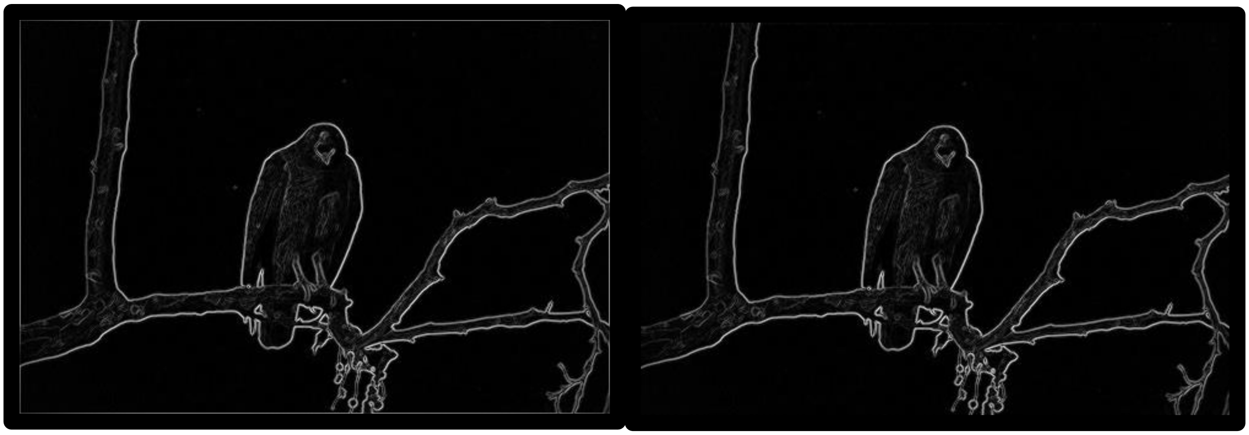
Figure 3.1 Add padding before convolution. Commended code is the original code

Figure 3.2 shows the difference before and after padding.

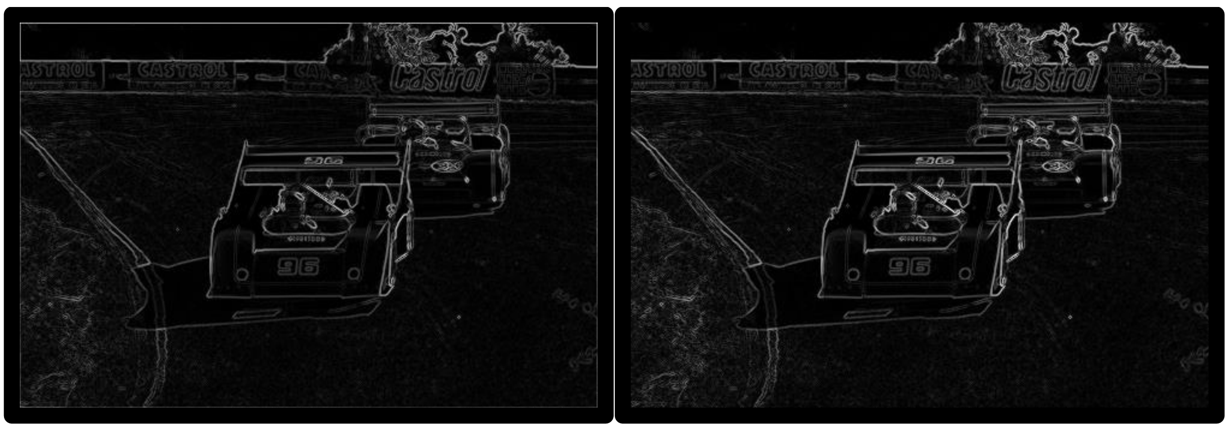


(a) Before 3096.png

(b) After 3096.png



(b) Before 42049.png (d) After 42049.png



(e) Before 21077.png (f) After 21077.png

Figure 3.2 Difference before and after padding. For the left column, it refers to the original code. For right column, it refers to after apply padding to the image. In order to see the artifact clearly, we add a picture border for all images

Below Table 3.1 shows the Contour quality performance metrics before and after the modification.

Table 3.1 Metrics before and after the modification

	before	after
f1 (overall max F-score)	0.514303	0.542282
best_f1 (average max F-score)	0.563120	0.587146
area_pr (AP)	0.409013	0.509025

Below Table 3.2 shows the impact of modification on run-time.

Table 3.2 Run-time before and after the modification

	before	after
run-time(s)	153.8494	159.6061

It seems like more running time after padding because of enlarging the image, i.e. more pixels need to be calculated.

- **Smoothing. Next, notice that we are using [−1, 0, 1] filters for computing the gradients, and they are susceptible to noise. Use derivative of Gaussian filters to obtain more robust estimates of the gradient. Experiment with different sigma for this Gaussian filtering and pick the one that works the best. (3 marks)**

We can get more accurate estimations of the gradient and lessen the impact of noise by using derivatives of Gaussian filters. The derivative process improves the image's edges while the Gaussian filter acts as a low-pass filter to reduce high-frequency noise.

To use derivative of Gaussian filters, we can modify the compute_edges_dxdy(I) function to apply a Gaussian filter before computing the gradients. Code is shown in Figure 3.2.

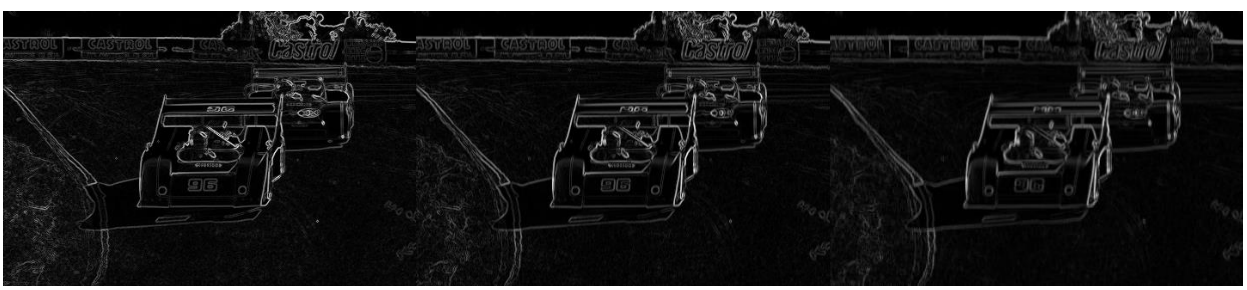
```
def compute_edges_dxdy(I):
    """Returns the norm of dx and dy as the edge response function."""
    I = I.astype(np.float32) / 255.
    # generate Gaussian filter
    sigma = 0.1
    gaussian_filter = cv2.getGaussianKernel(ksize=5, sigma=sigma)
    # Apply Gaussian Filter to the image
    I = signal.convolve2d(I, gaussian_filter, mode='same', boundary="symm")
    # adding the boundary argument with the value 'symm'. This will make the
convolution use 'symmetric' padding
    dx = signal.convolve2d(I, np.array([[-1, 0, 1]]), mode='same',
boundary="symm")
    dy = signal.convolve2d(I, np.array([[-1, 0, 1]]).T, mode='same',
boundary="symm")

    mag = np.sqrt(dx ** 2 + dy ** 2)
    mag = mag / np.max(mag)
    mag = mag * 255.
    mag = np.clip(mag, 0, 255)
    mag = mag.astype(np.uint8)
    return mag
```

Figure 3.2 Add Gaussian filters

We choose Sigma equals 0.1, 1, and 10 to observe the difference. Figure 3.3 shows the different images under different sigma.

(a) sig=0.1, 3096.png (b) sig=1, 3096.png (c) sig=10, 3096.png

(d) sig=0.1, 42049.png (e) sig=1, 42049.png (f) sig=10, 42049.png



(g) sig=0.1, 21077.png (h) sig=1, 21077.png (i) sig=10, 21077.png

Figure 3.3 Difference when choosing different Sigma. First-row sigma = 0.1, second-row sigma = 1, third-row sigma = 10

Below Table 3.3 shows the Contour quality performance metrics before and after the modification.

Table 3.3 Metrics before and after the modification

	before	After Sig = 0.1	After Sig = 1	After Sig = 10
f1 (overall max F-score)	0.542282	0.542282	0.562923	0.556896
best_f1 (average max F-score)	0.587146	0.587146	0.605533	0.600449

area_pr (AP)	0.509025	0.509025	0.550483	0.532105

Below Table 3.4 shows the impact of modification on run-time.

Table 3.4 Run-time before and after the modification

	before	after Sig = 0.1	After Sig = 1	After Sig = 10
run-time(s)	159.6061	158.9195	154.3474	149.0969

With the increase of sigma, the run-time decrease. But it doesn't seem to be decreasing significantly.

According to the view on the images and F-scores and AP values, it seems that when sigma equals to 1 can gain a better outcome.

- **Non-maximum Suppression. The current code does not produce thin edges. Implement non- maximum suppression, where we look at the gradient magnitude at the two neighbours in the direction perpendicular to the edge. We suppress the output at the current pixel if the output at the current pixel is not more than at the neighbors. You will have to compute the orientation of the contour (using the X and Y gradients), and implement interpolation to lookup values at the neighbouring pixels. (6 marks) In the code, you may need to define your own edge detector with non-maximum suppression. Note that all the functions are called in 'detect edges()'.**

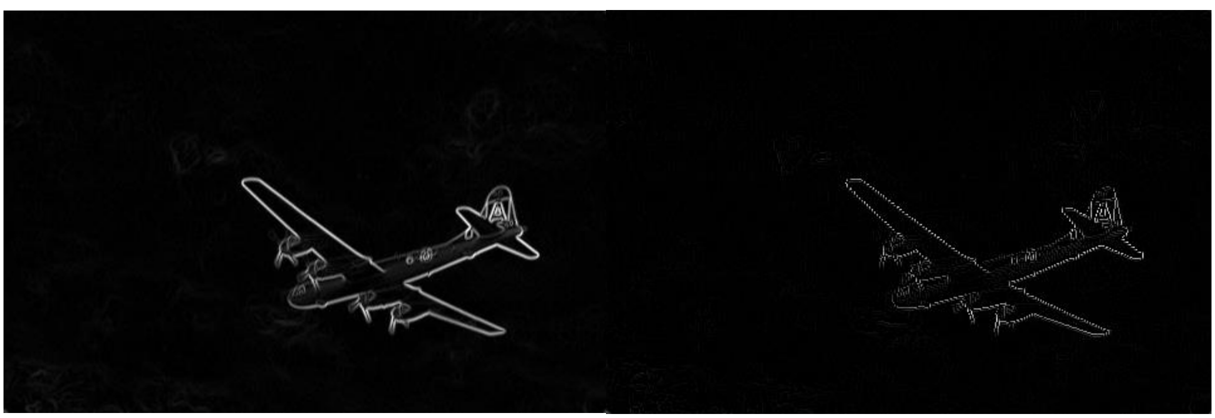
We must first use the X and Y gradients (we have calculated before) to compute the orientation of the contour in order to implement non-maximum suppression. Then, we need to interpolate values at the neighbouring pixels to compute the gradient magnitude at the two neighbours in the direction perpendicular to the edge. Finally, if the output at the current pixel is not more than that at the neighbours, we can suppress the output there.

We write a method called non_maximum_suppress(dx, dy, mag) to perform non-maximum suppression as shown in Figure 3.4.

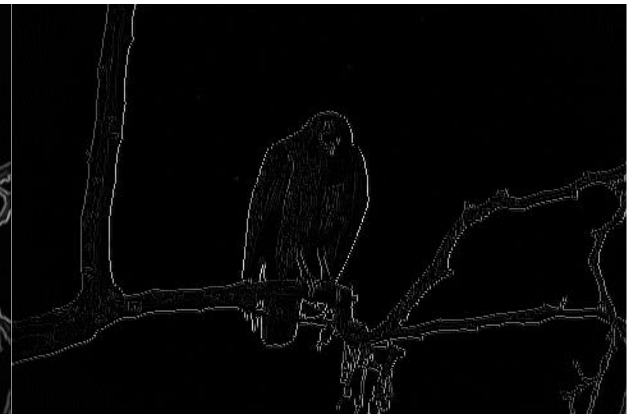
```python
def non_maximum_suppress(dx, dy, mag):
    angle = np.arctan2(dy, dx) * 180 / np.pi
    angle[angle < -90] += 180
    angle[angle > 90] -= 180

    for i in range(1, mag.shape[0] - 1):
        for j in range(1, mag.shape[1] - 1):
            a = angle[i, j]
            if -22.5 <= a <= 22.5 or a <= -157.5 or a >= 157.5:
                p1, p2 = mag[i, j - 1], mag[i, j + 1]
            elif -67.5 <= a < -22.5:
                p1, p2 = mag[i - 1, j - 1], mag[i + 1, j + 1]
            elif -112.5 <= a < -67.5:
                p1, p2 = mag[i - 1, j], mag[i + 1, j]
            else:
                p1, p2 = mag[i + 1, j - 1], mag[i - 1, j + 1]

            if mag[i, j] <= p1 or mag[i, j] <= p2:
                mag[i, j] = 0
    return mag
```

Figure 3.4 Non-maximum suppression function

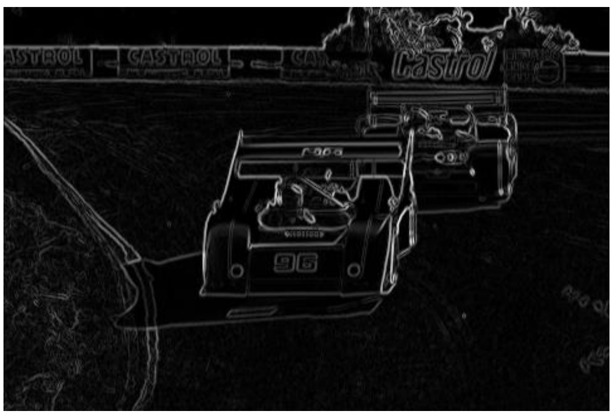
We perform non-maximum suppression function based on adding Gaussian filter with sigma equaling 1. Figure 3.5 will show before and after performing non-maximum suppression.

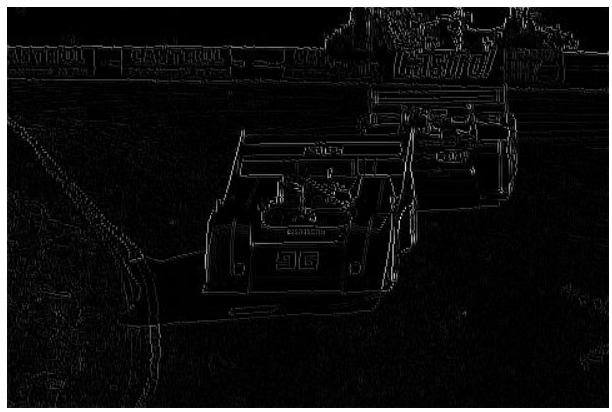


(a) Before 3096.png (b) After 3096.png



(c) Before 42049.png (d) After 42049.png





(e) Before 21077.png (f) After 21077.png

Figure 3.5 Difference before and after performing non-maximum suppression. Left column is before using non-maximum suppression function. Right column is after using non-maximum suppression function

Below Table 3.5 shows the Contour quality performance metrics before and after the modification.

Table 3.5 Metrics before and after the modification

	before	After
f1 (overall max F-score)	0.562923	0.564573
best_f1 (average max F-score)	0.605533	0.586527
area_pr (AP)	0.550483	0.561575

Below Table 3.6 shows the impact of modification on run-time.

Table 3.6 Run-time before and after the modification

	before	After
run-time(s)	154.3474	97.5503

Run-time decreases significantly. The possible reason for this is that non-maximum suppression helps eliminate duplicate or overlapping detections by selecting the most confident or representative detection among them. By discarding redundant detections, the subsequent processing steps can be performed on a reduced set of data, resulting in fewer computations and faster runtime.

- **Extra Credit. You should implement other modifications to get this contour detector to work even better. Here are some suggestions: compute edges at multiple different scales, use color information, propagate strength along a contiguous contour, etc. You are welcome to read and implement ideas from papers on this topic. (upto 5 marks)**

We combine colour information with grayscale edge detection to enhance contour detection. Converting the image to a colour space like CIELAB that separates the colour and intensity components is one approach to accomplish this. Figure 3.6 shows how to use colour information.

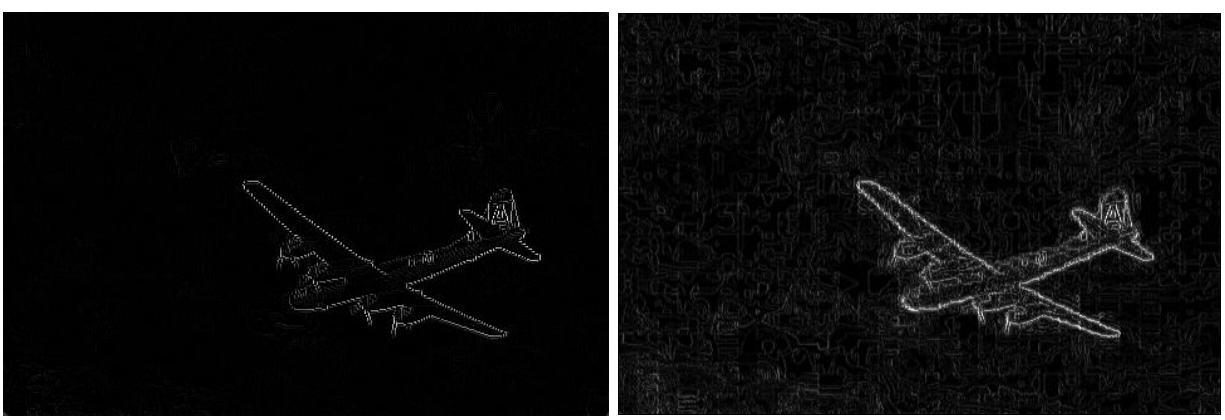
```
def compute_colour_edges(I):
    # Convert image to CIELAB color space
    lab = cv2.cvtColor(I, cv2.COLOR_BGR2LAB)

    # Split into L, A, and B components
    L, A, B = cv2.split(lab)

    # Compute edges for each channel
    edge_L = compute_edges_dxdy(L)
    edge_A = compute_edges_dxdy(A)
    edge_B = compute_edges_dxdy(B)

    # Combine edges into a single image
    colour_edges = np.max((edge_L, edge_A, edge_B), axis=0)

    return colour_edges
```

Figure 3.6 Code for converting image to CIELAB color space

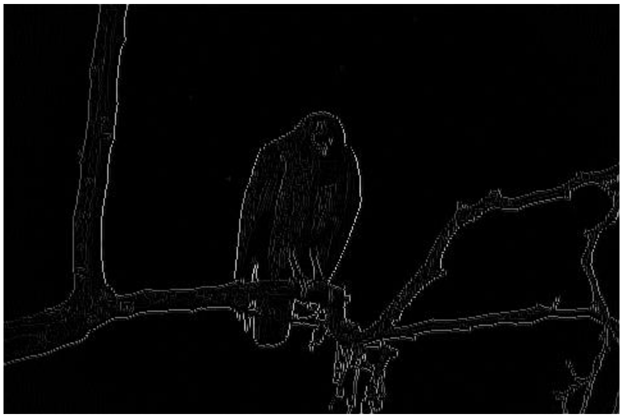
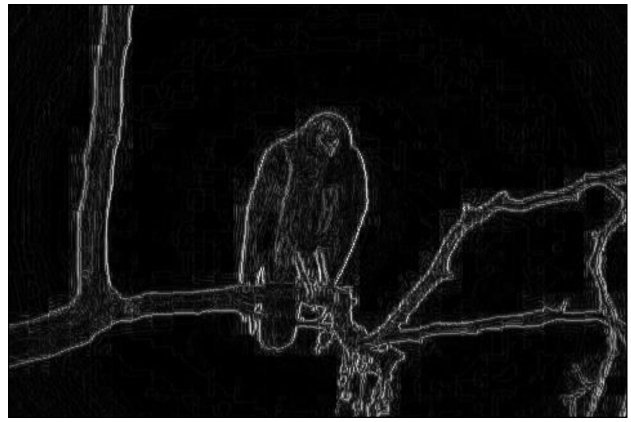
Since contours typically take the form of continuous curves, we may use this knowledge to increase the precision with which contours are detected. The edge strengths can be transmitted along the contour path using a contour propagation technique. This can aid in improving the first edge detection findings and producing contours that are smoother and more precise. The contour strength propagation factor should be added to non_maximum_suppress() function. After Changing, the function non_maximum_suppress() is shown in Figure 3.7.

```
def non_maximum_suppress(dx, dy, mag):
    propagate_factor = 0.5

    angle = np.arctan2(dy, dx) * 180 / np.pi
    angle[angle < -90] += 180
    angle[angle > 90] -= 180

    # implement the contour strength propagation
    propagated = np.zeros_like(mag)

    for i in range(1, mag.shape[0] - 1):
        for j in range(1, mag.shape[1] - 1):
            a = angle[i, j]
            if -22.5 <= a <= 22.5 or a <= -157.5 or a >= 157.5:
                p1, p2 = mag[i, j - 1], mag[i, j + 1]
            elif -67.5 <= a < -22.5:
                p1, p2 = mag[i - 1, j - 1], mag[i + 1, j + 1]
            elif -112.5 <= a < -67.5:
                p1, p2 = mag[i - 1, j], mag[i + 1, j]
            else:
                p1, p2 = mag[i + 1, j - 1], mag[i - 1, j + 1]

            if mag[i, j] <= p1 or mag[i, j] <= p2:
                propagated[i, j] = mag[i, j] * propagate_factor
            else:
                propagated[i, j] = mag[i, j]

    return propagated
```

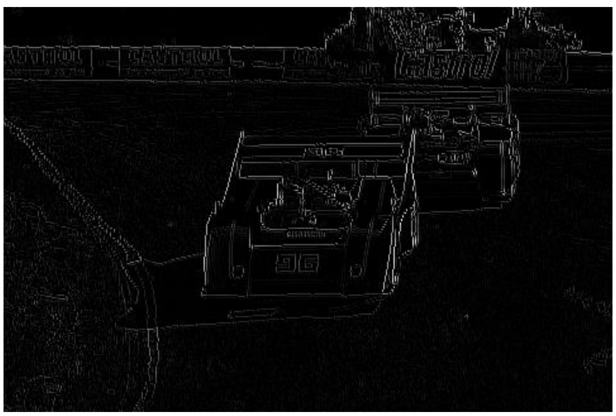
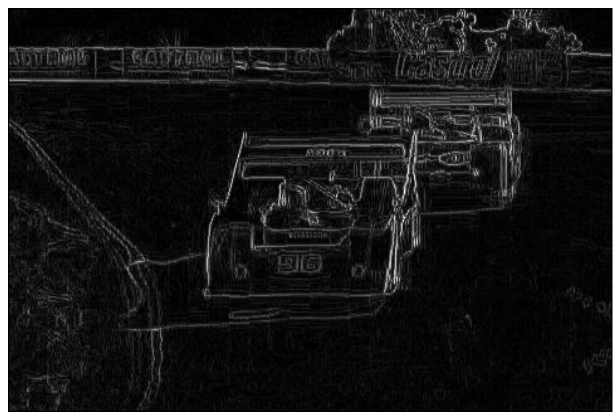
Figure 3.7 Revised non_maximum_suppress() function



(a) Before 3096.png (b) After 3096.png

(c) Before 42049.png (d) After 42049.png

(e) Before 21077.png (f) After 21077.png

Figure 3.8 Difference before and after modifications. Left column is before modifications. Right column is after modifications

Below Table 3.7 shows the Contour quality performance metrics before and after the modification.

Table 3.7 Metrics before and after the modification

	before	After
f1 (overall max F-score)	0.564573	0.598626
best_f1 (average max F-score)	0.586527	0.630397
area_pr (AP)	0.561575	0.581905

Below Table 3.8 shows the impact of modification on run-time.

Table 3.8 Run-time before and after the modification

	before	After
run-time(s)	97.5503	194.9693

After all, there is a slight improvement in accuracy and we can visually observe more details in our edge. But the run-time increase nearly twice.