COMP-ENGN-6528-Computer Vision Assignment Template

This is just a template for Assignment. Please make sure you follow all the requirements in assignment file if it is not mentioned in this template.

You may need to refer to the assignment file for detailed requirements

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)

Answer the meaning of a separable filter

A separable filter is a filter that can be presented as the outer product of two 1D column filter.

Report the separated components of the given filter

The given filter has two same separated components: [2 2 3]^T and [2 2 3]

Mistake	Marks Deducted
Wrong explanation of the meaning	0.5
Wrong separated components	0.5 for each

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

Report the range filter with brief description of computing process. Provide a screen shot of the code used to calculate it.

The range filter is shown below:

```
[[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]]
```

It is a filter with 5x5 size with each element calculated by the formula:

$$w_{range} = \exp\left(-\frac{||I(i,j)-I(k,l)||^2}{2\sigma_r^2}\right)$$

In the formula, I is the input matrix, sigma_r is the standard deviation provided as a hyper parameter, k and l are the index of the highlighted element, and i and j are the index of the 5x5 range around the highlighted element corresponding to the index of current computing element in the range filter.

The code is shown below:

```python
def w_range(I,k,l,size=5):
    # initialize a zero matrix with shape (size, size)
    ans = np.zeros((size,size))
    # traverse through the filtered region in I
    for i in range(k-size//2,k+size//2+1):
        for j in range(l-size//2,l+size//2+1):
            # update the zero matrix with value calculated by current traversing and central element
            ans[i-k+size//2,j-l+size//2] = np.exp(-np.linalg.norm(I[i,j]-I[k,l])**2/2/sigma_r**2)
    return ans
# generate range filter
range_filter = w_range(I,3,2)
```

Report the filtered value for the high-lighted pixel with the given domain kernel and your range kernel. Give a screen shot of code used to get the value.

The filtered value is: 109.19301842434827.

The code is shown below:

```
domain_filter = np.array([[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]])
def filterer_value(I,k,l,size=5):
    # obtain domain and range filter
    domain_kernel = domain_filter
    range_filter = w_range(I,3,2)
    # calculate wp by summing up the elementwise product of the two filters
    wp = np.sum(domain_kernel*range_filter)
    # calculate the filtered matrix
    ans = I[k-size//2:k+size//2+1,l-size//2:l+size//2+1]*range_filter*domain_kernel/wp
    # summing up to obtain filtered value
    return np.sum(ans)
# generate filtered value
f_value = filterer_value(I,3,2)
```

Mistake	Marks Deducted
Wrong range filter	0.5
Wrong description	0.5
Wrong range filter code	1.0
Wrong filtered value	1.0
Wrong filtering code	1.0

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)

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)

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()'.

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)

For each of the modifications above, your report should include:

- key implementation details focusing on the non-trivial aspects, hyper-parameters that worked the best,
- contour quality performance metrics before and after the modification.
- impact of modification on run-time,
- visual examples that show the effects of the modification on two to three images.

Follow the four requirements listed above in your report

For each question, you should also provide screenshots of your own code.

For run-time analysis, you should report the exact running time of your code as well as some discussions.

Each task (including extra credit), we will give according to the correctness your implementation, performance, running time analysis and visualization.

For Extra Credit, we will also give marks on valuable discussion of your modifications.

1. Warmup

The artifacts at boundaries are caused by the convolve2d function. By default, it will fill 0 to the padding boundaries. As a result, the boundaries have a high probability to be recognized as edge. To solve the problem, the mode of boundary is changed to "symm" instead of "fill". The code is shown below:

```
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')
```

I measured function compute_edges_dxdy since I modified code in detect function. The running time is recorded for baseline code and modified code and shown in Table 1.

Table 1 Run-time before and after the modification

	before	after
run-time(s)	0.5731348991394043	0.7784221172332764

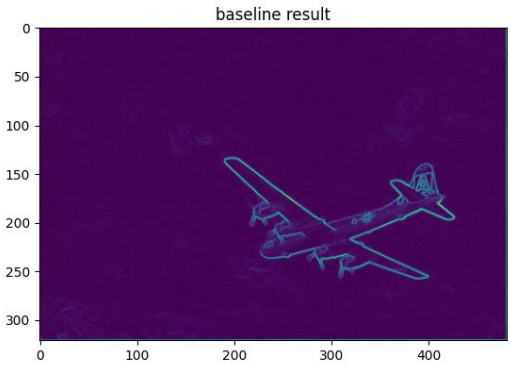
We can see slight running time increase. I think it is because filling edge with default value is faster than initializing paddings w.r.t. the border values.

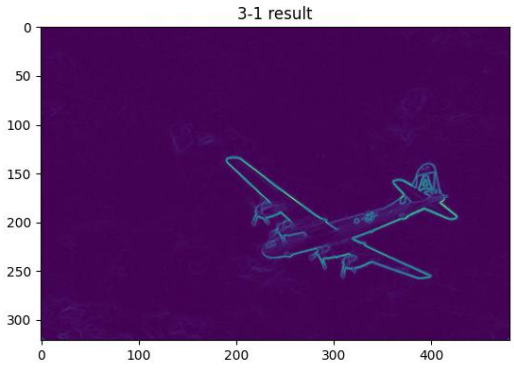
The performance is recorded and shown in Table 2.

Table 2 Metrics before and after the warmup modification

	before	after
f1 (overall max F-score)	0.514369	0.542432
best_f1 (average max F-score)	0.562687	0.587287
area_pr (AP)	0.408983	0.509132

It is obvious that the performance is significantly better than the baseline method because the pixels at border will be misclassified as edges in baseline method. By solving the problem, the number of misclassified pixels decreases. To visualize the effect, the two images generated by baseline and modified method is shown below:





We can see that there is a light line at the border of baseline result but it will not happen in modified result (3-1 result).

2. Smoothing

The idea of the derivative of Gaussian is that using the derivative of Gaussian filter to convolve the image instead of directly calculating the derivative of image. The method will smooth the image and thus is robust to noises. According to the derivative of Gaussian method, I implemented the algorithm in function compute_edges_dxdy. The function is shown below:

```python
def compute_edges_dxdy(img):
    I = img.astype(np.float32) / 255
    sigma = sigmaa
    ksize = sizee
    # obtain Gaussian kernel with hyperparameters provided
    gaussian_kernel = cv2.getGaussianKernel(ksize=ksize, sigma=sigma)
    gaussian_kernel = np.dot(gaussian_kernel, gaussian_kernel.T)
    # obtain derivative of Gaussian kernel
    dx_g = signal.convolve2d(gaussian_kernel, np.array([[-1, 0, 1]]), mode='same', boundary='symm')
    dy_g = signal.convolve2d(gaussian_kernel, np.array([[-1, 0, 1]]).T, mode='same', boundary='symm')
    # convolve image with derivative of Gaussian kernel
    dx = signal.convolve2d(I, dx_g, mode='same', boundary='symm')
    dy = signal.convolve2d(I, dy_g, mode='same', boundary='symm')
    # compute magnitude of convolved image
    mag = np.sqrt(dx ** 2 + dy ** 2)
    # normalize magnitude
    mag = mag / np.max(mag)
    mag = mag * 255.
    mag = np.clip(mag, 0, 255)
    mag = mag.astype(np.uint8)
    return mag, dx, dy
```

I measured function compute_edges_dxdy since I modified code in detect function. The running time is recorded for baseline code and modified code and shown in Table 3.

Table 3 Run-time before and after the modification

	before	after
run-time(s)	0.7784221172332764	1.0051114559173584

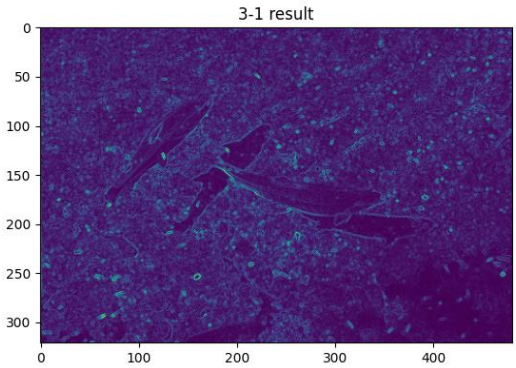
We can see the increasing time consumption because of the extra calculation to derive the derivative of Gaussian filter.

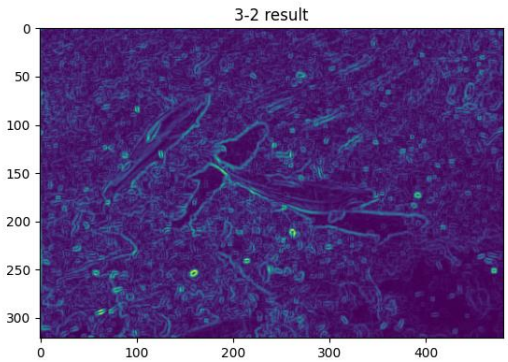
Contour quality performance metrics before and after the modification.

Table 4 Metrics before and after the modification

	before	after
f1 (overall max F-score)	0.542432	0.575478
best_f1 (average max F-score)	0.587287	0.614760
area_pr (AP)	0.509132	0.570332

We can observe slight improvement in F1 score but huge improvement in precision. It is because denoising decreased the number of false positive cases caused by noise.





Looking into the second vertical fish from left, we can see less pixels are classified as edges, which is an expected effect after de-noising.

3. Non-maximum Suppression

Non-maximum suppression can be used to produce thin edges. Since the gradient has two directions: x and y, we can obtain a weighted mixed gradient to represent the close pixels on the edge direction. The code with detailed procedures about the algorithm is shown below:

```python
def non_maximum_suppression(gxy, gx, gy):
    ans = np.zeros(gxy.shape)
    for i in range(gxy.shape[0] - 2):
        for j in range(gxy.shape[1] - 2):
            y = i + 1
            x = j + 1
            # if no gradient then continue
            if gx[y, x] == 0 and gy[y, x] == 0:
                continue
            # if the gradient is close to y direction
            elif np.abs(gx[y, x]) <= np.abs(gy[y, x]):
                # the main direction
                g_main_b = gxy[y - 1, x]
                g_main_f = gxy[y + 1, x]
                # degree from y direction
                w = abs(gx[y, x]) / abs(gy[y, x])
                # minor direction gradient x
                if gx[y, x] * gy[y, x] > 0:
                    g_minor_b = gxy[y - 1, x - 1]
                    g_minor_f = gxy[y + 1, x + 1]
                else:
                    g_minor_b = gxy[y - 1, x + 1]
                    g_minor_f = gxy[y + 1, x - 1]
            # if the gradient is close to x direction
            else:
                # the main direction
                g_main_b = gxy[y, x - 1]
                g_main_f = gxy[y, x + 1]
                # degree from x direction
                w = abs(gy[y, x]) / abs(gx[y, x])
                # minor direction gradient y
                if gx[y, x] * gy[y, x] > 0:
                    g_minor_b = gxy[y + 1, x - 1]
                    g_minor_f = gxy[y - 1, x + 1]
                else:
                    g_minor_b = gxy[y - 1, x - 1]
                    g_minor_f = gxy[y + 1, x + 1]
            # mix the gradients according to the degree
            g_mix_b = w * g_minor_b + (1 - w) * g_main_b
            g_mix_f = w * g_minor_f + (1 - w) * g_main_f
            # suppress the pixel if it is not the maximum
            if gxy[y, x] >= g_mix_b and gxy[y, x] >= g_mix_f:
                ans[y, x] = gxy[y, x]
    return ans
```

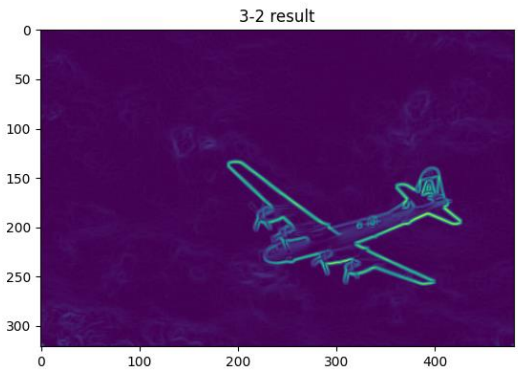
The non-max suppression is performed in function compute_edges_dxdy before magnitude matrix is normalized.

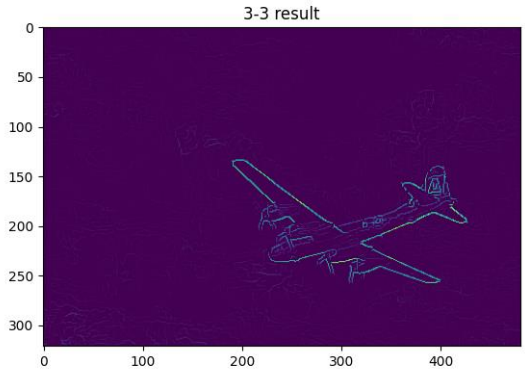
Contour quality performance metrics before and after the modification is shown in Table 5.

Table 5 Metrics before and after the modification

	before	after
f1 (overall max F-score)	0.575478	0.576699
best_f1 (average max F-score)	0.614760	0.598168
area_pr (AP)	0.570332	0.585199

We can observe increasing precision as the predicted contour is getting thinner and the number of false positive cases is shrinking. However, since the thin contour may not be located on the ground truth location, some of the ground truth may be classified as negative, causing F1 score to decrease. We can visualize the effect with comparison shown below:





We can observe that the contour generated by 3-3 became significantly thinner than the result in 3-2

The impact of modification on run-time is shown in Table 6

Table 6 Run-time before and after the modification

	before	after
run-time(s)	1.0051114559173584	21.924161672592163

We can see that traversing through the matrix and comparing cost a lot of time. The code might need to be optimized with vectorization in numpy.