Introduction to Computer Vision

Assignment #1

Due: Oct.-18 (Fri.) (before 11:59pm)

Instruction

a. Submit your source codes in a single compressed file “CV_A1_*StudentID*.zip” to iCampus.

b. Python 3.7 or higher / OpenCV 3.4 or higher will be used to execute your submitted codes.

c. In this assignment, you will take grayscale images as input. In order to open an image as grayscale, you can use the following statement:

```
img = cv2.imread( IMAGE_FILE_PATH , cv2.IMREAD_GRAYSCALE )
```

d. You can submit at most 5 python files. In other words, you can add 2 additional python files to implement functions commonly utilized across different parts.

e. Any work that you turn in should be your own.

Part #1. Image Filtering [34 pts]

The requirements of Part #1 will be evaluated by running ‘***A1_image_filtering.py***’ file.

1-1. Image Filtering by Cross-Correation

a) Implement two functions that return cross-correlation between an image and a 1D/2D kernel:

```
function filtered_img = cross_correlation_1d ( img , kernel )

function filtered_img = cross_correlation_2d ( img , kernel )
```

b) Your function `cross_correlation_1d` should distinguish between vertical and horizontal kernels based on the shape of the given kernel.

c) You can assume that the all kernels are odd sized along both dimensions.

d) Your functions should preserve the size of the input image. In order words, the sizes of `img` and `filtered_img` should be identical. To do this, you need to handle boundary cases on the edges of the image. Although you can take various approaches, you are asked to pad the image such that pixels lying outside the image have the same intensity value as the nearest pixel inside the image.

e) You cannot use any built-in function that performs cross-correlation, colvolution, filtering or image padding.

1-2. The Gaussian Filter

a) Implement two functions that return 1D and 2D correaltion kernels for the Gaussian filter:

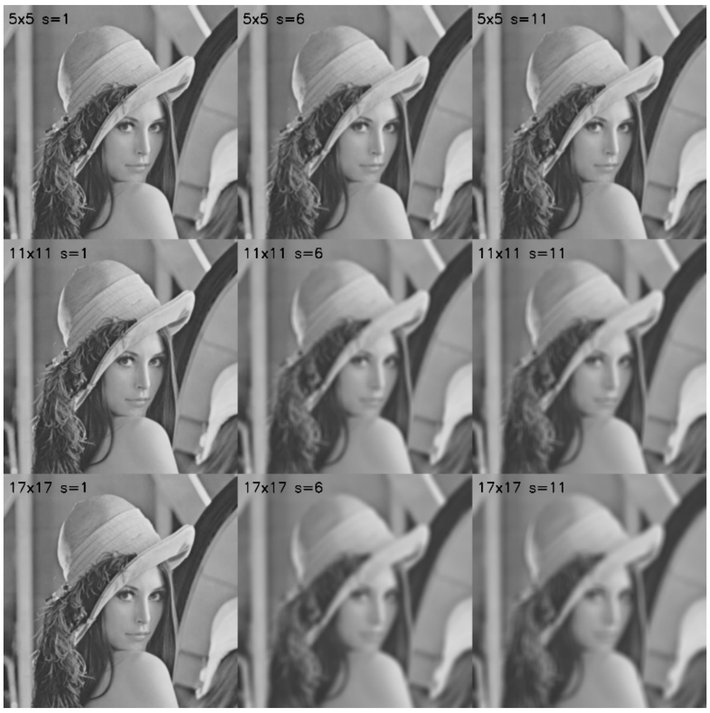
```
function kernel = get_gaussian_filter_1d ( size , sigma )

function kernel = get_gaussian_filter_2d ( size , sigma )
```

b) You can assume that `size` is an odd number.

c) Print the results of `get_gaussian_filter_1d(5,1)` and `get_gaussian_filter_2d(5,1)` to the console.

d) Perform at least 9 different Gaussian filtering to an image (e.g., combinations of 3 different kernel sizes and sigma values). Show the filtered images in a single window and write them as a single image file '*./result/part_1_gaussian_filtered_INPUT_IMAGE_FILE_NAME*'. You are also asked to display a text caption describing the filter parameters on each filtered image.



e) Perform the Gaussian filtering by applying vertical and horizontal 1D kernels sequantially, and compare the result with a filtering with a 2D kernel. Specifically, visualize a pixel-wise difference map and report the sum of intensity differences to the console. You are also required to report the computational times of 1D and 2D filterings to the console.

f) Your script should produce results of (d) and (e) for 'lenna.png' and 'shapes.png'.
(i.e. '*result_gaussian_filtered_lenna.png*' and '*result_gaussian_filtered_shapes.png*')

g) When performing filtering, you have to use the function implemented in 1-1.

h) You cannot use any built-in function that produces Gaussian filters.

Part #2. Edge Detection [33 pts]

The requirements of Part #2 will be evaluated by running '***A1_edge_detection.py***' file.

2-1. Apply the Gaussian filtering to the input image.

a) You need to use functions implemented in Part #1.

b) The Gaussian filter $(7, 1.5)$ is utilized to produce the following examplar results.

2-2. Implement a function that returns the image gradient by referring the lecture slide
(page #83 of *CV_01_Image_Filtering.pdf*):

```
function mag, dir = compute_image_gradient ( img )
```

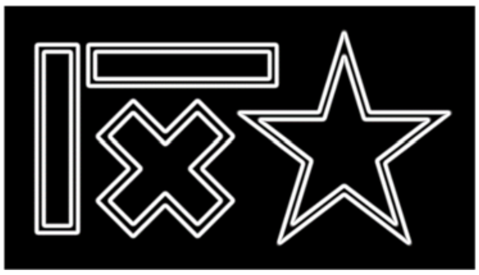
a) First apply the Sobel filters to compute derivatives along x and y directions

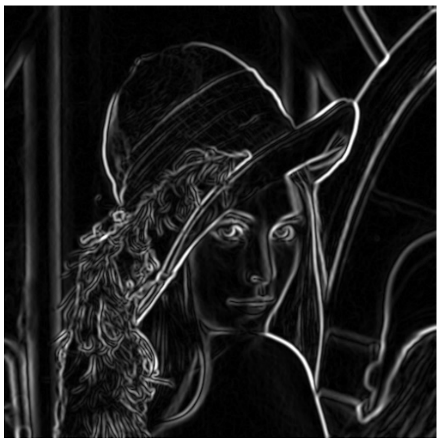
b) For each pixel, compute magnitude and direction of gradient.

c) You need to use functions implemented in Part #1.

d) Report the computational time taken by `compute_image_gradient` to the console. Show the computed manitude map and store it to an image file '*./result/part_2_edge_raw_ INPUT_IMAGE_FILE_NAME*'.

e) Your script should produce (d) for 'shapes.png' and 'lenna.png'





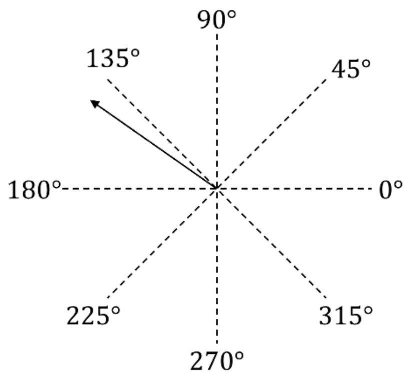
2-3. Implement a function that performs Non-maximum Suppression (NMS):

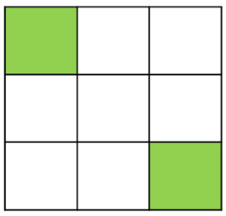
```
function suppressed_mag = non_maximum_suppression_dir ( mag , dir )
```

a) You are asked to implement an approximated version of NMS by quantizing the gradient directions into 8 bins. If a direction is represented by an angle in degrees, we can map the direction to the closet representative angle among $[0°, 45°, \dots, 315°]$.

b) Once the direction is quantized, compare the gradient magnitude against two magnitudes along the quantized direction. In this assignment, you do not have to interpolate the magnitude for the simplicity. If the gradient magnitude at the center position is not greater than the ones along the gradient direction, it is suppressed (the magnitude is set to 0).

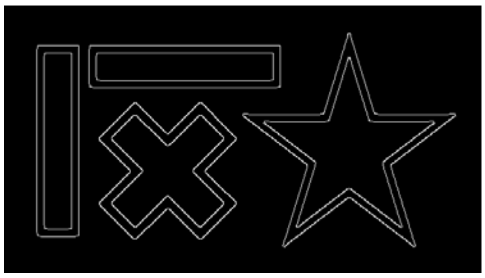
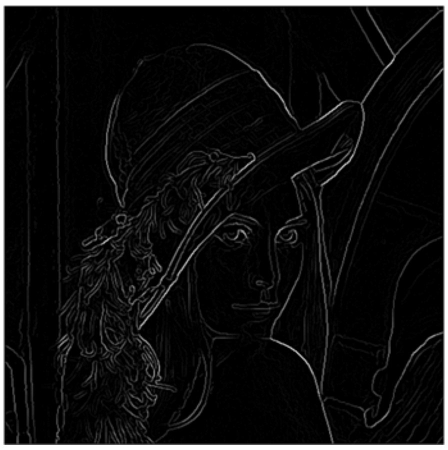
c) For instance, if the gradient direction is $145°$ then it is quantized to $135°$. In this case, the magnitude at the center position in the window should be compared to the north-west and south-east positions as illustrated in the figure below.





d) Report the computational time consumed by `non_maximum_suppression_dir` to the console. Show the supressed manitude map and store it to an image file '*result_edge_sup_ INPUT_IMAGE_FILE_NAME*'.

e) Your script should produce (d) for 'shapes.png' and 'lenna.png'

Part #3. Corner Detection [33 pts]

The requirements of Part #3 will be evaluated by running '***A1_corner_detection.py***' file.

3-1. Apply the Gaussian filtering to the input image.

a) You need to use functions implemented in Part #1.

b) The Gaussian filter $(7, 1.5)$ is utilized to produce the following examplar results.

3-2. Implement a function that returns corner response values:

```
function R = compute_corner_response ( img )
```

a) First apply the Sobel filters to compute derivatives along x and y directions

b) For each pixel, compute the second moment matrix M. You can utilize an uniform window function (i.e. $w(x, y) = 1$ if (x, y) is lying in the window, otherwise $w(x, y) = 0$). Use 5×5 window to compute the matrix M.

c) You should use the following response function with $\kappa = 0.04$:

$$R = \lambda_1\lambda_2 - \kappa(\lambda_1 + \lambda_2)^2$$

d) Once the response values for all the pixels are computed, update all the negative responses to 0 and then normalize them to a range of $[0,1]$.

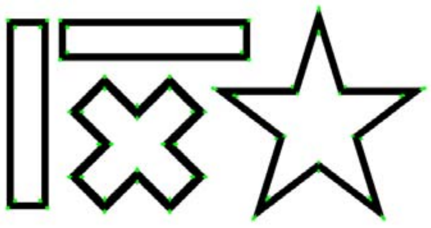
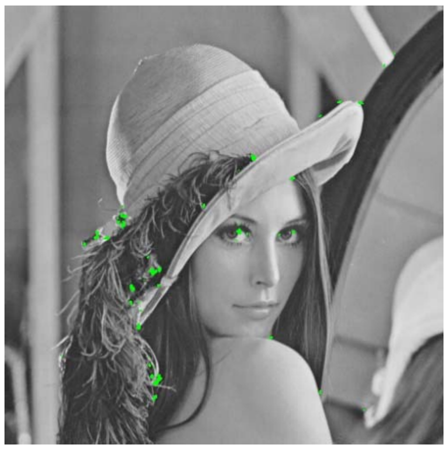
e) Report the computational time of `compute_corner_response` to the console. Visualize the computed corner response intensities and store them as an image file '*./result/part_3_corner_ raw_ INPUT_IMAGE_FILE_NAME*'.

f) Your script should produce (d) for 'shapes.png' and 'lenna.png'

3-3. Thresholding and Non-maximum Suppression (NMS):

a) Change the color of pixels having corner response greater than a threshold of 0.1 to green.

b) Display the result of (a) and store it as an image file
‘*./result/part_3_corner_bin_ INPUT_IMAGE_FILE_NAME*’

c) Implement a function that compute local maximas by non-maximum suppression:

```
function suppressed_R = non_maximum_suppression_win ( R , winSize )
```

This function suppresses (i.e. set to 0) the corner response at a position (x, y) if it is not a maximum value within a squared window sized `winSize` and centered at (x, y). Althogh the response is a local maxima, it is suppressed if it not greater than the threshold 0.1.
Set the parameter `winSize = 11.`

d) Report the computational time taken by `non_maximum_suppression_win` to the console.
Display the result of (c) by draw green circles and store it as an image file
‘*./result/part_3_corner_sup_ INPUT_IMAGE_FILE_NAME’*.

e) Your script should produce (b) and (d) for ‘shapes.png’ and ‘lenna.png’