CSC420 Assignment 1

Chen Liang

Sept 21, 2019

Question 1

(a). If we apply the convolution on one pixel, it takes $O(m^2)$ time, as we might need to change all values within this $m \cdot m$ filter. There are three different scenarios when applying the filter to the border of the graph:
(i) "full": output size is bigger than the image:
In this case, we could apply $(n + m - 1)$ convolutions on one whole line of pixels of image, so we the total number of applying convolution would be $(n + m - 1)^2$. Then the total running time would be $O(m^2 \cdot (n + m - 1)^2)$.
(ii) "same": output size is the same as the image:
In this case, we could apply n convolutions on one whole line of pixels of image, so we total number of applying convolution would be n^2. Then the total running time would be $O(m^2 \cdot n^2)$.
(iii) "valid": output size is smaller than the image:
In this case, we could apply $n - m + 1$ convolutions on whole line of pixels of image, so we the total number of applying convolution is $(n-m+1)^2$. Then the total running time would $O(m^2 \cdot (n-m+1)^2)$.
In all three cases, if m is significantly smaller than n, we could igore the m term in the complexity analysis, then the total running time would all be $O(m^2 \cdot n^2)$.

(b). Since the filter is separable, we could break down into the horizontal direction and the vertical direction. In this case, if we apply the convolution on one pixel, it takes $m + m = 2m$ time, as we might need to change all vertical filter values and all horizontal filter values. Similar to the analyze above, we could analyze the three cases individually, then for the "full" case, the total running time is $2m \cdot (n - m + 1)$, which is bounded by $O(m \cdot (n - m + 1)^2)$. For the "same" case, the running time would be $O(m \cdot n^2)$, and for the "valid" case, the running time would be $O(m \cdot (n + m - 1)^2)$.
In all cases, if m is significantly smaller than n, we could simply ignore the m term in the complexity analysis, then the total running time would be $O(m \cdot n^2)$.

Question 2

We could break down the procedure of Canny Edge Detection into four basic steps:
Step One: Filter image with derivative of Gaussian.
Detailed Explanation and Purpose: By applying Gaussian filter on this image, we could reduce the noise of this image. Knowing that an image with a significant amount of noise would make the detector observing way more edges than expected, as the derivative on one line of a relatively noisy photo would fluctuate more violently then a normal one. Therefore, reducing the noise of the photo would be crucial if we don't want to detect unwanted edges. Hence, by achieving so, we could Gaussian filter to smooth the photo.

Step Two: Find magnitude and orientation of gradient.
Detailed Explanation and Purpose: We know that an edge is a place of rapid change in the image intensity function, so by calculating the gradient of the image, we could easily detect where exits a rapid change in the pixel's intensity and the direction of the change is. After doing the gradient calculation of the whole graph, we could detect all possible edges.

Step Three: Non-maximum suppression.
Detailed Explanation and Purpose: By implementing Non-maximum suppression, we could thin out the edges. To achieve this, we could check whether a pixel is local maximum along gradient direction, and if it's indeed an local maximum, then take this pixel, otherwise simply exclude this pixel from edge. By conducting non-maximum algorithm, the resulting image would be the same, but with thinner edges.

Step Four: Linking and thresholding (hysteresis).
Detailed Explanation and Purpose: We need to define two threshold, a low one and a high one, then use the high threshold, and after getting these two thresholds, we could identify pixels into three categories: high pixels, low pixels, and irrelevant pixels. High pixels are pixels that have an intensity so high that we are sure they contribute to the final edge; irrelevant pixels are considered as non-relevant for the edge; low pixels are pixels that have an intensity value that is not enough to be considered as strong ones, but yet not small enough to be considered as non-relevant for the edge detection. By applying the two threshold algorithm, the result of this step is an image with only two pixel intensity values, high ones and low ones. Based on this result, hysteresis consists of transforming low pixels into strong ones, if and only if at least one of the pixels around the one being processed is a high one. Therefore, edges could be tracked by hysteresis.

Question 3

Laplacian of Gaussian applied to graph is in fact the second derivative on one line of pixels contained in the graph. Observe the pattern of $\frac{\partial^2}{\partial x^2} h \star f$ we notice that when the first derivative attains the local maxima, the second second derivative crosses the point of zero. Similar to the mechanism to detect edge by using first derivative, once the second derivative or Laplacian function has a zero crossing, an edge could be detected.

$$(\frac{\partial^2}{\partial x^2} h) \star f$$

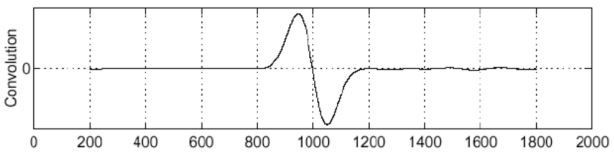


Figure 1: Source: Taati, Babak, https://q.utoronto.ca/courses/124091/pages/lecture-2, Sept 2019

Question 4

(a) For this question, I provided following test cases:

```
if __name__ == '__main__':
    # (a) Test sample in matrix representation
    img = np.array([[90, 0, 90, 0, 90], [90, 0, 90, 0, 90], [90, 0, 90, 0, 90], [90, 0, 90, 0, 90], [90, 0, 90, 0,
     90]])
    kernal = np.array([[1 / 9, 1 / 9, 1 / 9], [1 / 9, 1 / 9, 1 / 9], [1 / 9, 1 / 9, 1 / 9]])

    matrix_valid = MyCorrelation(img, kernal, "valid")
    print("Valid:{} ".format(matrix_valid))
    matrix_same = MyCorrelation(img, kernal, "same")
    print("Same:{} ".format(matrix_same))
    matrix_full = MyCorrelation(img, kernal, "full")
    print("Full:{} ".format(matrix_full))

    # (a) Test sample in actual grayscale graph
    img = cv2.imread('mona_lisa.jpg', 0)
    print("THE SHAPE OF THE ORIGINAL GRAPH IS ", img.shape)
    img_valid = MyCorrelation(img, kernal, "valid")
    print("THE SHAPE OF THE MODIFIED GRAPH IS ", img_valid.shape)
    img_same = MyCorrelation(img, kernal, "same")
    print("THE SHAPE OF THE MODIFIED GRAPH IS ", img_same.shape)
    img_full = MyCorrelation(img, kernal, "full")
    print("THE SHAPE OF THE MODIFIED GRAPH IS ", img_full.shape)
```

And the result is:

```
/Users/chenliang/PycharmProjects/CSC420A1/venv/bin/python /Users/chenliang/PycharmProjects/CSC420A1/Q4.py
Valid:[[60. 30. 60.]
 [60. 30. 60.]
 [60. 30. 60.]]
Same:[[20. 40. 20. 40. 20.]
 [30. 60. 30. 60. 30.]
 [30. 60. 30. 60. 30.]
 [30. 60. 30. 60. 30.]
 [20. 40. 20. 40. 20.]]
Full:[[10. 10. 20. 10. 20. 10. 10.]
 [20. 20. 40. 20. 40. 20. 20.]
 [30. 30. 60. 30. 60. 30. 30.]
 [30. 30. 60. 30. 60. 30. 30.]
 [30. 30. 60. 30. 60. 30. 30.]
 [20. 20. 40. 20. 40. 20. 20.]
 [10. 10. 20. 10. 20. 10. 10.]]
THE SHAPE OF THE ORIGINAL GRAPH IS  (426, 381)
THE SHAPE OF THE MODIFIED GRAPH IS  (424, 379)
THE SHAPE OF THE MODIFIED GRAPH IS  (426, 381)
THE SHAPE OF THE MODIFIED GRAPH IS  (428, 383)

Process finished with exit code 0
```

(b) For this question, I provided a test case like this,

```
in_order_kernel = np.array([[1/9, 2/9, 3/9], [2/9, 3/9, 4/9], [3/9, 4/9, 5/9]])
reversed_kernel = reverse_convolution(in_order_kernel)
print(reversed_kernel)
```

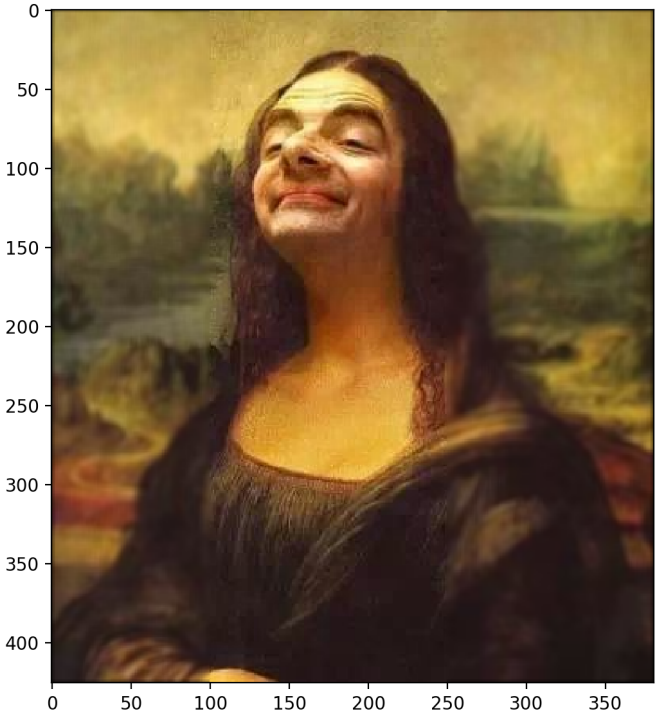
And the result is,

```
/Users/chenliang/PycharmProjects/CSC420A1/venv/bin/python /Users/chenliang/PycharmProjects/CSC420A1/Q4.py
[[0.55555556 0.44444444 0.33333333]
 [0.44444444 0.33333333 0.22222222]
 [0.33333333 0.22222222 0.11111111]]

Process finished with exit code 0
```

(c) The code for creating the portrait is below:

```
# (c) portrait mode
img = cv2.imread('mona_lisa.jpg')
plt.imshow(img, cmap='gray')
img2 = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
kernel_3x3 = np.ones((3, 3), np.float32) / 9.0
kernel_5x5 = np.ones((5, 5), np.float32) / 25.0
img2[:, :, 0] = MyCorrelation(img2[:, :, 0], kernel_5x5, "same")
img2[:, :, 1] = MyCorrelation(img2[:, :, 1], kernel_5x5, "same")
img2[:, :, 2] = MyCorrelation(img2[:, :, 2], kernel_5x5, "same")
img3 = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
# select 100:250 as the region of portrait
img2[:, 100:250, 0] = img3[:, 100:250, 0]
img2[:, 100:250, 1] = img3[:, 100:250, 1]
img2[:, 100:250, 2] = img3[:, 100:250, 2]
plt.imshow(img2)
plt.show()
```

Explanation: The reason why I choose to use the mode "same" is that I need to apply function call $MyCorrelation$ on $img2$ three times, and after each time calling this function, one colour channel in the original image would be blurred, and the returned image is still $img2$. If we choose modes other than "*same*" we are modifying the size of the $img2$ after each time calling $MyCorrelation$, and this might rise an error due to the change of the matrix size. The kernel I choose is a 5 times 5 mean filter, which could smooth the background by setting every pixels as the average of its nearby pixels. Hence, the background are significantly blurred while the portrait itself looks clear.
The photo created looks like below:


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

Question 5

(a) A separable filter can be written as the product of two more filters. If a two-dimensional filter is separable, then it could be written as the product of two one-dimensional filters.
(b) The test case I listed as follows,

```python
if __name__ == '__main__':
    # (b) test cases to show whether the input kernel is separable or not.
    # if it could be broken into two 1-D kernels, print these two kernels.
    non_sep_example_one = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
    non_sep_example_two = np.array([[-1, 2, 0, 4], [0, 7, 1, -12], [0, 0, 0, 0], [0, 0, 0, 0]])
    sep_example_one = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])
    sep_example_two = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])

    example_list = [non_sep_example_one, non_sep_example_two, sep_example_one, sep_example_two]

    for i in range(len(example_list)):
        separable, h1, h2 = isSeparable(example_list[i])
        if separable is False:
            print("FOR CASE ", i, " THE MATRIX IS NOT SEPARABLE")
        else:
            print("FOR CASE ", i, " THE MATRIX IS SEPARABLE, AND COULD BE SEPARATED INTO", h1, " AND", h2)
```

and the result is,

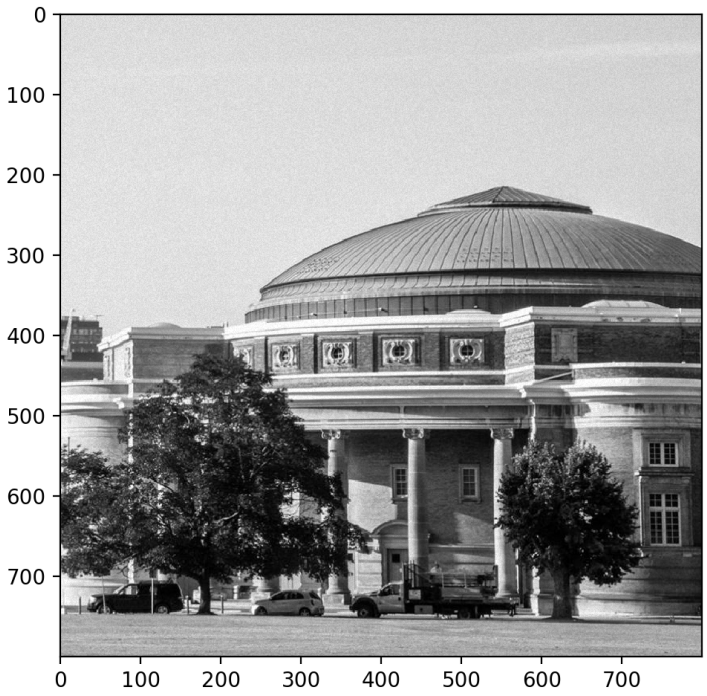
```
/Users/chenliang/PycharmProjects/CSC420A1/venv/bin/python /Users/chenliang/PycharmProjects/CSC420A1/Q5.py
FOR CASE  0  THE MATRIX IS NOT SEPARABLE
FOR CASE  1  THE MATRIX IS NOT SEPARABLE
FOR CASE  2  THE MATRIX IS SEPARABLE, AND COULD BE SEPARATED INTO [-1. -2. -1.]  AND [-1. -2. -1.]
FOR CASE  3  THE MATRIX IS SEPARABLE, AND COULD BE SEPARATED INTO [-0.75983569 -1.51967137 -0.75983569]  AND [
  1.31607401  0.         -1.31607401]

Process finished with exit code 0
```

Question 6

(a) Here's the instructions how to add random noise to 'gray.img';

```
if __name__ == '__main__':
    # (a) add random noise to the image
    img = cv2.imread('gray.jpg', 0)
    img2 = addRandomNoise(img, 0.05)
    plt.imshow(img2, cmap='gray')
    plt.show()
```

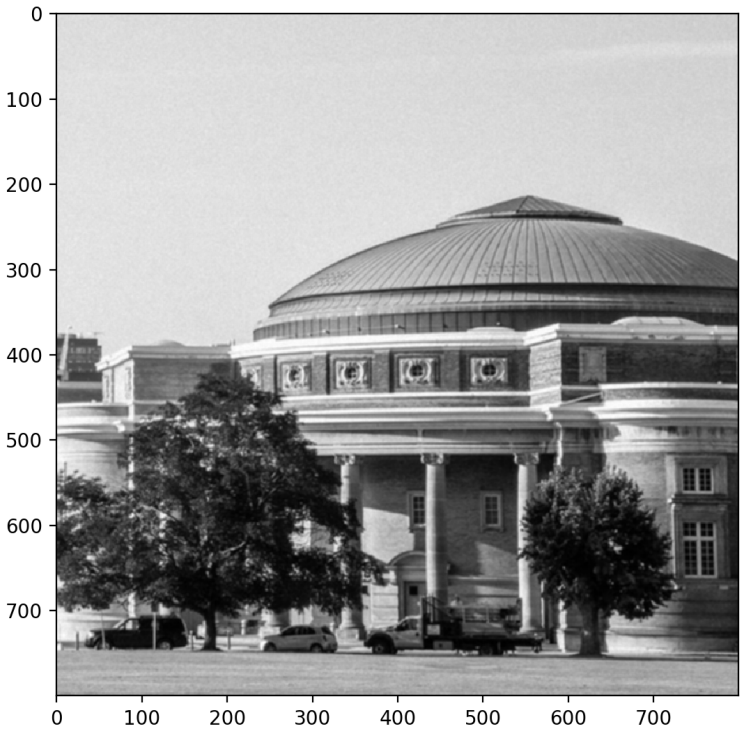
and the resulting graph is,



(b) We choose mean filter to smooth the image. In this case, because the noise are random number ranging between 0 to 255, we use the average of the intensity of nearby pixels to reduce the amount of intensity variation between one pixel and the next, and in this way we could make the image looks more smooth.
The instructions I used looks like below;

```
# (b) the filter I chose is the mean filter
# create a mean filter
kernel = np.ones((3, 3), np.float32) / 9
img3 = cv2.filter2D(img2, -1, kernel)
plt.imshow(img3, cmap='gray')
plt.show()
```

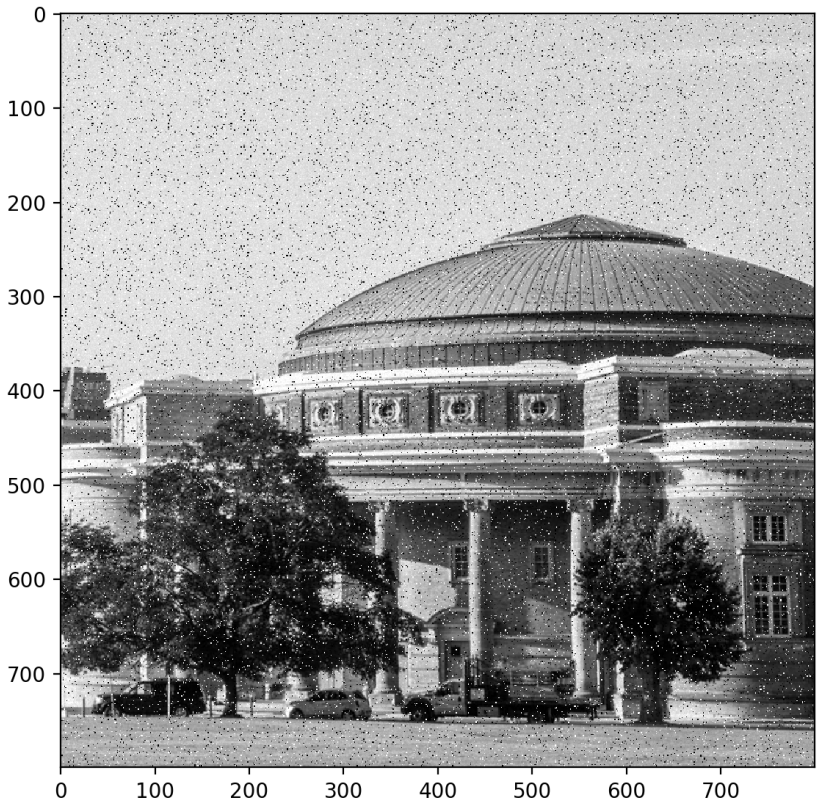
and the resulting image looks like this,



(c) Add salt and pepper noise to 'gray.img':

```
# (c) add salt and pepper noise to the image
img4 = addSaltAndPepperNoise(img, 0.05)
plt.imshow(img4, cmap='gray')
plt.show()
```

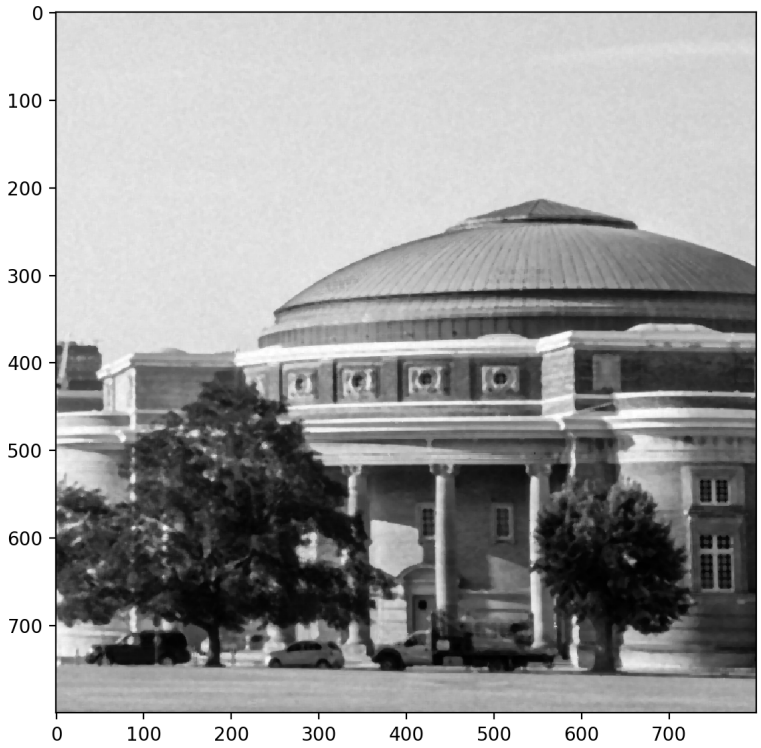
The resulting image looks like:



(d) For salt and pepper noise, we choose median filter to reduce noise. The reason is that since salt and pepper noise adds 0 or 255 pixel to the image, calculating the average of nearby pixel intensities might probably differ a lot from the original pixel intensity due to the reason that for pixels like 0 and 255, their intensities are so extreme and they might have a very string impact on the average number. So the better option would be finding the median number, which is a non-linear method, to reduce the intensity difference. In this way, we could make the image more smooth by using salt and pepper filter.

```
# (d) use median filer to remove salt and pepper noise
img5 = cv2.medianBlur(img4, 5)
plt.imshow(img5, cmap='gray')
plt.show()
```

And the noise-reduced image looks like below,



(e) My proposal is by using MyMedianFilter, it could remove all spots where salt and pepper noise are (pixels with full color intensity), and by doing so we could effectively remove some noisy points.

```
# (e) use myMedianFilter to filter out salt and pepper noise
img6 = cv2.imread('color.jpg')
img6 = cv2.cvtColor(img6, cv2.COLOR_BGR2RGB)
img6 = addSaltAndPepperNoise(img6, 0.05)
img6[:, :, 0] = myMedianFilter(img6[:, :, 0], 3)
img6[:, :, 1] = myMedianFilter(img6[:, :, 1], 3)
img6[:, :, 2] = myMedianFilter(img6[:, :, 2], 3)
plt.imshow(img6)
plt.show()
```

And the resulting image looks like below,