CSC420: Assignment 1

Xiangyu Kong, kongxi16

September 24, 2019

Problem 1

1. The computational cost for $h * I$ when h is not separable is $O(n^2m^2)$.

   This is because each pixel in I gets computed for m^2 times, and there are n^2 pixels in total.

2. The computational cost for $h * I$ when h is separable is $O(m^2 2n)$.

   This is because each pixel in I gets computed for $2m$ times (only vertical and horizontal edge detecting vectors), and there are n^2 pixels in total

Problem 2

Canny Edge Detection Steps:

1. Filter the image with derivative of Gaussian in both horizontal and vertical directions.
   - The purpose of this step is to smooth the image and remove random noise.
   - To do this, we apply Gaussian filter to convolve with the image.
2. Find the magnitude and direction for the gradients.
   - The purpose of this step is to find all possible edges
   - To do this, we apply edge detection filters (for example, Sobel) with different directions (vertical, horizontal and diagonal) and convolve with the image.
3. Non-maximum suppression
   - The purpose of this step is to get rid of the spurious response from edge detection produced by noise.
   - To do this, we only take local maximum or minimum of the edges.
4. Linking and Thresholding
   - The purpose of this step is to connect the unlinked edges.
   - To do this, define 2 thresholds: low and high. We use the high threshold's results to start the edge curves (get rid of the weak edges) and use low threshold's results to connect the unlinked edges.

Problem 3

Laplacian of Gaussian is the second derivative of Gaussian. This means that all the zero-crossing pixels are either local maxima or local minima. In an image, if a pixel is a local maximum or minimum, then this pixel is on an edge.
Laplacian of Gaussian filters are also symmetrical. It has the shape of Figure 1 both vertically and horizontally, so it can detect edges both vertically and horizontally.

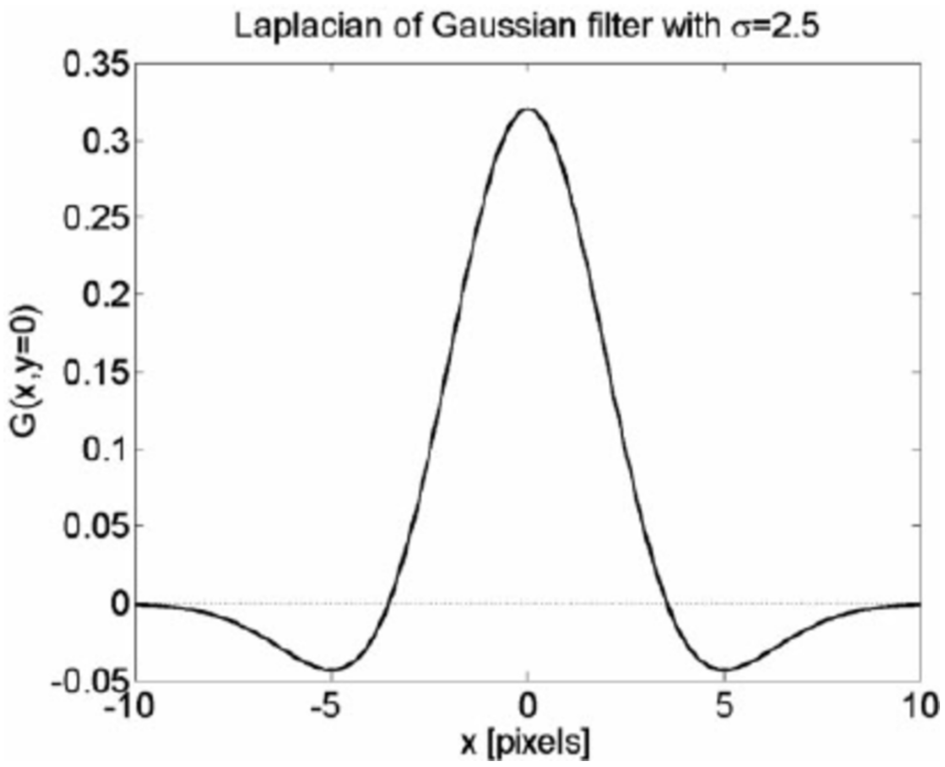


Figure 1: Laplacian of Gaussian with $\sigma = 2.5$

Problem 4

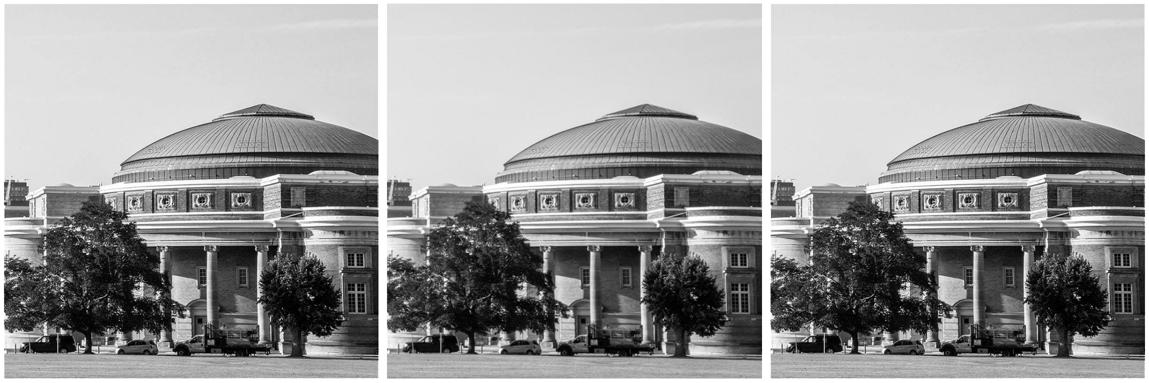
See code implementations in question_4.py.

1. See results of my_correlation in Figure 2a. The result is produced by correlating the original image (Figure 2b) with the sharpening filter $\begin{bmatrix}0 & 0 & 0\\0 & 2 & 0\\0 & 0 & 0\end{bmatrix} - \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$.

   Note that the image is sharper comparing to the original image. (One specific location to look for would be the edge of the dome.)

2. See results of my_convolution in Figure 2c. The result is produced by convolving the original image (Figure 2b) with the sharpening filter $\begin{bmatrix}0 & 0 & 0\\0 & 2 & 0\\0 & 0 & 0\end{bmatrix} - \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$.

   Note that the image is sharper comparing to the original image. (One specific location to look for would be the edge of the dome.)



(a) my_correlation (b) Original image (c) my_correlation

Figure 2

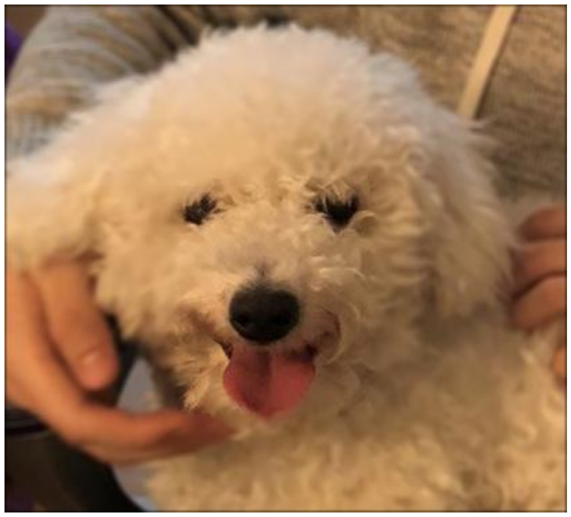
Note: Since the sharpening filter is symmetrical, the results for correlation and convolution are the same, but they are different in implementation. The difference between convolution and correlation is that the filter is flipped in convolution.

3. To achieve portrait mode, we want to blur the background. To do this a blur filter is chosen.

   The filter chosen was $\frac{1}{25}\begin{bmatrix}1 & 1 & 1 & 1 & 1\\1 & 1 & 1 & 1 & 1\\1 & 1 & 1 & 1 & 1\\1 & 1 & 1 & 1 & 1\\1 & 1 & 1 & 1 & 1\end{bmatrix}$.

   Note that the normal 3×3 blur matrix was not used. This is because the output for $\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$ is not obvious enough. Taking average of the 3×3 neighbor is less powerful than taking average of the 5×5 neighbor.

   See portrait mode and original photo in Figure 3. The face for the dog is the focus rectangle. Notice that the hand and the ears are fuzzed out.


(a) Portrait mode

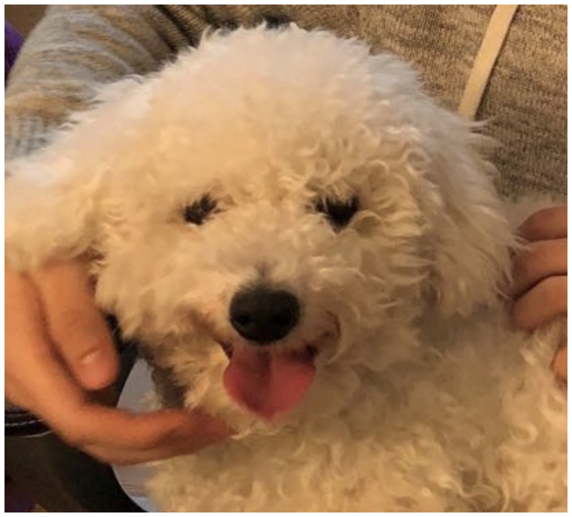

(b) Original image

Figure 3

Problem 5

See code implementations in question_5.py.

1. Separable Filters allow correlation and convolution operations to be performed at $2K$ operation cost instead of K^2. This is because separable filters are able to be split into $1D$ horizontal and $1D$ vertical filters. The result of convolving the image with the two $1D$ filters is the same as convolving the image with the original separable filter.

   This is achieved when when the Singular Value Decomposition has only one non-zero single value.

$$F = U\Sigma V^T = \Sigma_{i=0}^{k} \sigma_i u_i v_i^T$$

   where $\sqrt{\sigma_1}\boldsymbol{u}_1$ and $\sqrt{\sigma_1}\boldsymbol{v}_1$ are the vertical and horizontal filters.

2. - The Separable filter is $\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$.

     The output for the separable filter is

```
[[-0.25]
 [-0.5 ]
 [-0.25]]

[[-0.25 -0.5  -0.25]]
True
```

   - The Inseparable filter is $\begin{bmatrix}0 & -1 & 0\\ -1 & 5 & -1\\ 0 & -1 & 0\end{bmatrix}$.

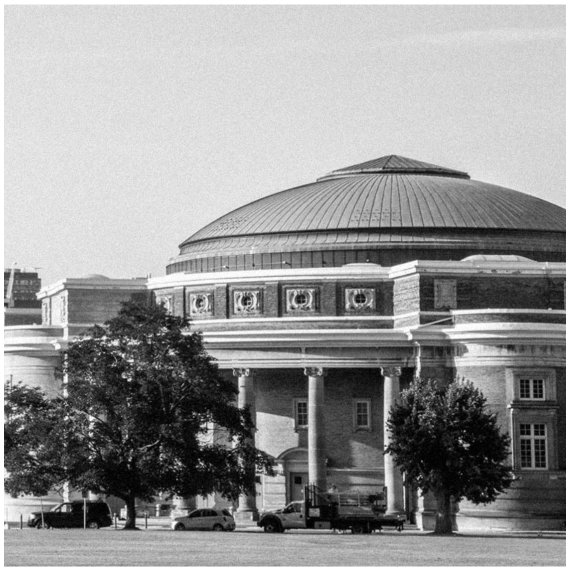
     The output for the separable filter is

```
False
```

Problem 6

See code implementations in question_6.py. Notice the

1. The image with noise is shown in Figure 4a. Notice that Figure 4a contains random noise (One specific location to look for is the sky).


(a) add_rand_noise

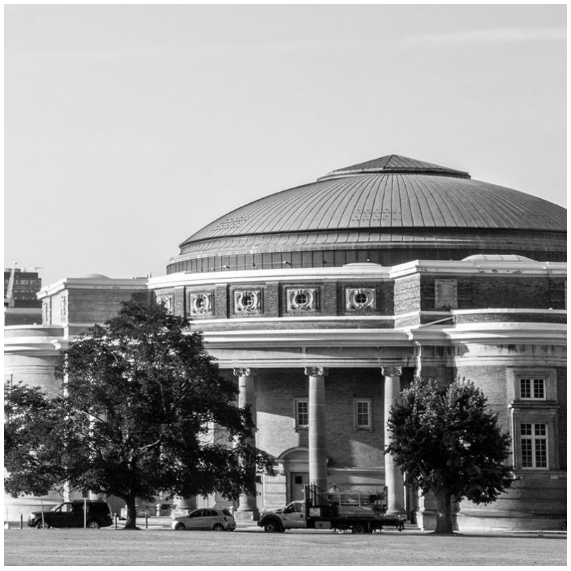
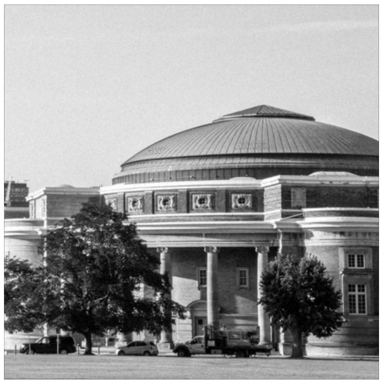

(b) Original image

Figure 4

2. The recovered image is shown in Figure 5b. The chosen filter was the mean filter $\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$. This filter was chosen because averaging the pixels will reduce the effect of the noise and make the noisy pixel resemble the average of its neighbors.

(a) add_rand_noise


(b) The recovered image

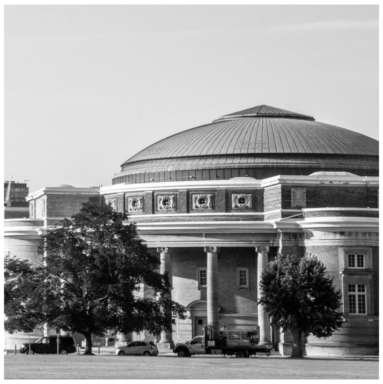
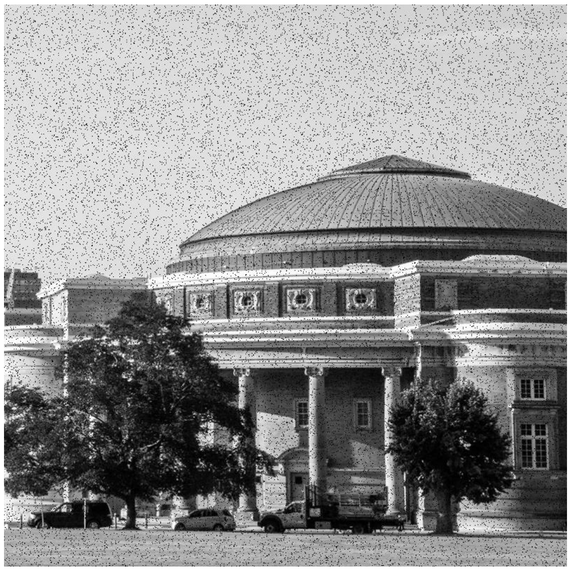

(c) Original image

Figure 5

3. The image with salt and pepper noise is shown in Figure 6a. Notice that there are random black and white pixels across the image.


(a) add_salt_and_pepper

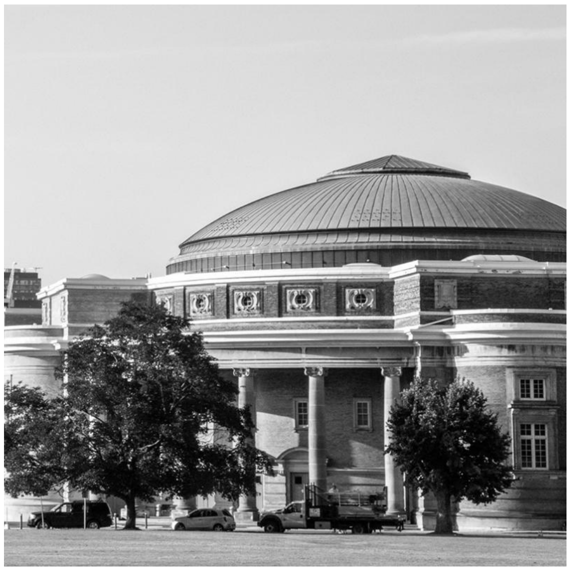
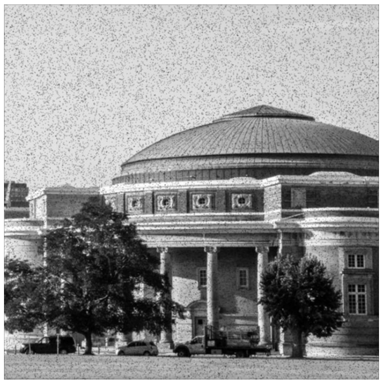

(b) Original image

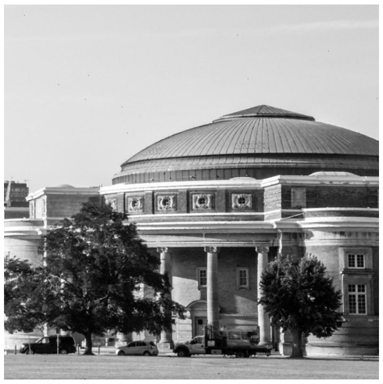
Figure 6

4. Trying to recover salt and pepper noisy image with the mean filter does not yield a very good result: the output image is still noisy. The result is shown in 7a. This is because salt and pepper noise is very extreme. The noisy pixel has signal of either 1 or 0 (white or black). This affects the mean vastly.

   The chosen filter to remove the salt and pepper noise is the median filter. It is similar to the mean filter: it looks at each pixel's neighbors and take the median value instead of the mean value.

   This is achieved with OpenCV's *medianBlur* method. This works because salt and pepper noise's extreme noisy signals are less likely to affect the median of the nearby pixels. The only way that the median is affected is that more than half of the pixels within the neighbor frame are affected. Thus, by taking the median of the near by pixels, it is more likely that we will get a better resemblance of the neighbor pixels. The result of the median filter is shown in Figure 7b.


(a) mean filter


(b) median filter

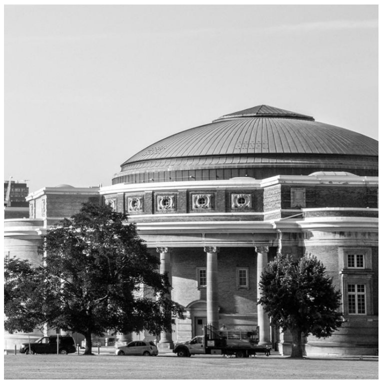

(c) Original image

Figure 7

5. The salt and pepper noisy image applied to the color image is shown in 8a. This is achieved by adding salt and pepper noise to each channel and then combining the noisy channels.

   When applying the median filter as in part d, most of the noise are gone.

   As in part d, this is achieved with OpenCV's *medianBlur* method. By taking the median of the near by pixels, we will likely get a better resemblance of the neighbor pixels. The result is shown in Figure 8b.

   I was not able to produce the effect shown in the assignment. However, we can see that the median filter image (Figure 8b) is slightly blurred when comparing to the original image (Figure 8c). This is because by taking the median, the pixel is be affected by its neighbor pixels.

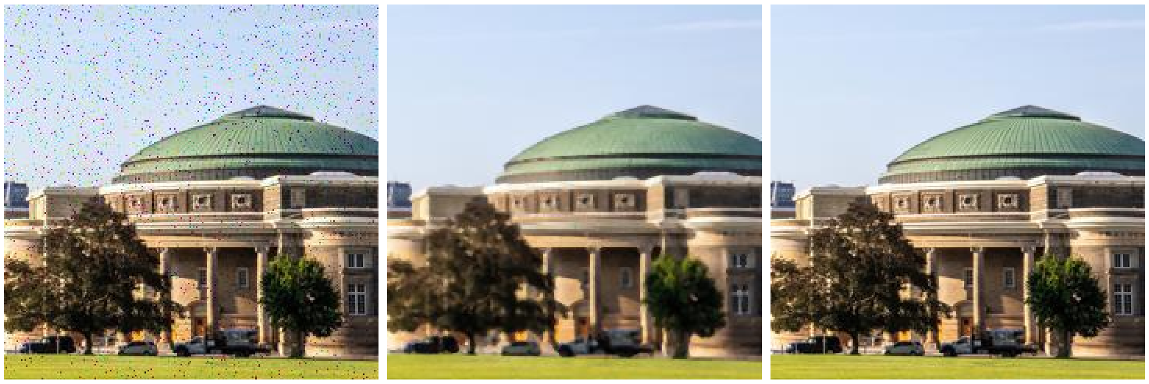

(a) salt_and_pepper noisy image (b) median filter (c) Original image

Figure 8