Assignment 1

Xinqi shen

1. (a) The process of performing h * I require m^2 operations per pixel, thus the total cost is $\mathbf{n^2 \times m^2}$ if h is not separable.

   (b) The process of performing h * I require 2m operations per pixel, thus the total cost is $\mathbf{n^2 \times 2m}$ if h is separable, since we separate the operation by first performing a 1D horizontal convolution followed by a 1D vertical convolution.

2. Steps of Canny edge:

   1. **Filter image with derivative of Gaussian (horizontal and vertical directions)**: the purpose of this step is to smooth the image by applying the Gaussian filter in order to remove the noise. To implement this step, we just apply the derivative of Gaussian and perform a convolution with the original image.

   2. **Find magnitude and orientation of gradient**: the purpose of this step is to compute the direction and magnitude of the gradient, so we can find the orientation of edge normal and edge strength accordingly. To implement this step, we first compute the gradient of an image $\nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}]$, then the gradient direction is given by: $\theta = \tan^{-1}(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$ and the gradient magnitude is given by $|| \nabla f ||$ = $\sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$.

   3. **Non-maximum suppression**: the purpose of this step is to get rid of spurious pixel to make the edge thinner and sharper. To implement this step, we just chose the pixel with local maximum along gradient direction.

   4. **Linking and thresholding (hysteresis):** the purpose of this step is to find an accurate representation of real edges in an image since some pixels along the edge

didn't survive the thresholding, so we need verify whether these pixels are extracted from the true edge or just the noise. To implement this step, we set up two thresholds, high threshold and low threshold. If an edge pixel's gradient value is higher than high threshold value, it is a strong edge pixel. If an edge pixel's gradient value is between high and low threshold, it is a weak edge pixel. if an edge pixel's value is smaller than low threshold, it is the noise. Thus, in terms of hysteresis, we use a high threshold to start edge curves and a low threshold to continue them, eventually for those weak pixels that are connected to the strong edge pixels will be preserved.

3. Laplacian of Gaussian is used in the zero crossing edge detector since it calculates the second spatial derivative of an image. This means that when the Laplacian changes sign, it happens where the intensity of the image changes rapidly, such points often occur at the edge in images. Also, the positive value of Laplacian usually on the darker side, and negative on the lighter side. More mathematically, the second derivative changes sign when its first derivative reaches its local max/min, which is the sharpest intensity change of the image – the edge of the image.

4. (a)

Original image

Gaussian blur image by using MyCorrelation function

Code:

```
def MyCorrelation(I,F,mode):
  image_h, image_w = I.shape[0], I.shape[1]
  filter_h, filter_w = F.shape[0], F.shape[1]

  if mode == 'valid':
    padding_v_h = image_h - filter_h + 1
    padding_v_w = image_w - filter_w + 1
    padding = np.zeros((padding_v_h,padding_v_w))

    for i in range(padding.shape[0]):
      for j in range(padding.shape[1]):
        padding[i,j] = np.sum(F * I[i:i+filter_h,j:j+filter_h])
    return padding

  elif mode == 'same':
    padding_s_h = image_h + filter_h - 1
    padding_s_w = image_w + filter_w - 1
    padding = np.zeros((padding_s_h,padding_s_w))

    padding[(filter_h - 1)//2:(image_h + (filter_h-1)//2),(filter_w - 1)//2:
            (image_w + (filter_w-1)//2)] = I
    return MyCorrelation(padding,F,'valid')

  elif mode == 'full':
    padding_f_h = image_h + 2 * filter_h - 2
    padding_f_w = image_w + 2 * filter_w - 2
    padding = np.zeros((padding_f_h,padding_f_w))

    padding[(filter_h - 1):(padding_f_h - (filter_h - 1)), (filter_w - 1):
            (padding_f_w - (filter_w - 1))] = I
    return MyCorrelation(padding,F,'valid')
```

(b)

Gaussian blur image by using MyConvolution function

Code:

```
def MyConvolution(I,F,mode):
  new_F = F[::-1,::-1]
  return MyCorrelation(I,new_F,mode)
```

(c) I use the Gaussian filter with k size 5 (5 * 5 kernel) and $\sigma = 3$. Since the portrait mode will make background objects out of focus, then we can easily apply Gaussian filter to make the background image a little fuzzy and keep the original portrait part same. Additionally, we still want to keep the background recognizable, so I select the reasonable k size and σ.

The original image

The image in portrait mode

Code:

```
def portrait(I,mask):
  face = I * mask
  kernelX = cv2.getGaussianKernel(5, 3);
  kernelY = cv2.getGaussianKernel(5, 3);
  kernel = kernelX * np.transpose(kernelY)
  res = MyConvolution(I,kernel,'same')
  image_h, image_w = I.shape[0], I.shape[1]
  for i in range(face.shape[0]):
    for j in range(face.shape[1]):
      if face[i][j] == 0:
        face[i][j] = res[i][j]
  return face
```

5. (a) A separable filter can be written as product of two simple filters. For example, a 2D convolution operation can be separated into two 1D filters, which reduces the computing cost (we calculated the cost in question 1). Mathematically, a filter is separable if only one singular value is non-zero by looking at the singular value decomposition (SVD).

   (b)

```
separablefilter_init = np.array([
    [1, 2, 1],
    [2, 4, 2],
    [1,2,1]
])

separablefilter = 1/16 * separablefilter_init

nonSeparablefilter = np.array([
    [1,2,3],
    [4,5,6],
    [7,8,9]
])

def isSeparableFilter(F):
  u, s, v = np.linalg.svd(F)
  if np.linalg.matrix_rank(np.diag(s)) == 1:
    print (np.sqrt(s[0])* u[:,0])
    print (np.sqrt(s[0])* v[0,:])
    return True
  else:
    return False

res1 = isSeparableFilter(separablefilter)
print(res1)
res2 = isSeparableFilter(nonSeparablefilter)
print(res2)
```

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

6. (a)

The original image

The image with random noises

Code:

```
def AddRandNoise(I, m):
  new_gray = I/255
  noise = np.random.uniform(low=-m, high=m, size=(I.shape[0],I.shape[1]))
  result = noise + new_gray
  return result*255
```

(b) I use the Gaussian filter with k size 5 (5 * 5 kernel) and $\sigma = 5$. Since Gaussian filter can be used to smooth the image in order to remove the noise and it performed good to the random noise where local variations caused by grain are reduced.

Gaussian filtered image

(c)

The image with Salt and Pepper noise

Code:

```
def AddSaltAndPepperNoise(I,prob):
    if(len(I.shape) == 2):
        output = np.zeros((I.shape[0],I.shape[1]),np.uint8)
        lower_threshold = prob/2
        upper_threshold = 1 - prob/2
        for i in range(I.shape[0]):
            for j in range(I.shape[1]):
                random_num = np.random.rand()
                if random_num < lower_threshold:
                    output[i][j] = 0
                elif random_num > upper_threshold:
                    output[i][j] = 255
                else:
                    output[i][j] = I[i][j]
    else:
      output = np.zeros((I.shape[0],I.shape[1],I.shape[2]),np.uint8)
      lower_threshold = prob/2
      upper_threshold = 1 - prob/2
      for i in range(I.shape[0]):
        for j in range(I.shape[1]):
          for k in range(I.shape[2]):
            random_num = np.random.rand()
            if random_num < lower_threshold:
                output[i][j][k] = 0
            elif random_num > upper_threshold:
                output[i][j][k] = 255
            else:
                output[i][j][k] = I[i][j][k]

    return output
```

(d) It does not work very well when removing the Salt and pepper noise by Gaussian filter. Since the corrupted pixels are either 0 or 255, the Gaussian filter are very sensitive to the extreme values. Thus, we can use median filter to remove the Salt and pepper noises, median is more robust and less sensitive to outliers.

remove Salt and Pepper noise by median filter

(e) I use the following method:

Step 1: Select a window size, e.g. 3*3. Assume the processing pixel is P which lies at the center of window.

Step 2: If 0 < P < 255, the processing pixel p is labeled as uncorrupted and keep the same value

Step 3: If p = 0 or p = 255, then pixel p is labeled as corrupted we considered 4 cases

Case 1: If the selected window has all the pixels as 0 and p is 255, then we change the p value to 0

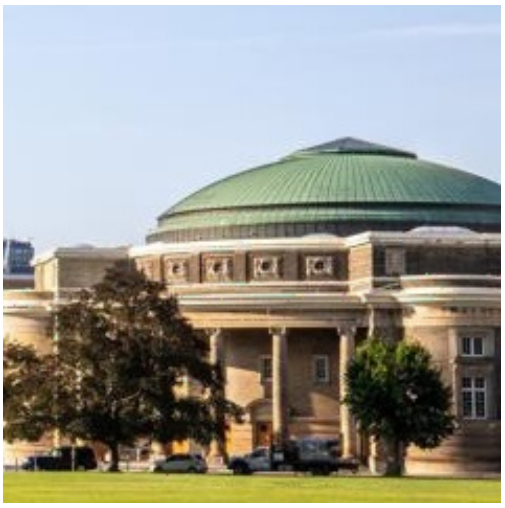
Case 2: If the selected window has all the pixels are 255 and p is 0, then we change the p value to 255

Case 3: If the selected window contains both 0 and 255, then we replace p value by the mean value of current window.

Case4: If the selected window contains only 0 or 255, then we remove 0 and 255, and replace p value by the median in the remaining elements.

The median filter will apply to each pixel and replace its value by the median, it works fine in gray scale image due to the color in each pixel only determined by a single value. However, for the color image, each pixel's color is decided by 3 value and if we simply use median filter, it will eventually cause many noisy color blocks occurs. In terms of my method, I only change the value for the corrupted pixel depend on different cases, it decreases the possibility that causes the noisy color and generate more similar color with its neighbors

The image with Salt and Pepper



The image after filter

Code:

```
def remove_multichannel_SP_noise(I,k_size):
  padding_h = I.shape[0] - k_size + 1
  padding_w = I.shape[1] - k_size + 1
  filter_mean = np.ones((k_size, k_size),np.uint8)/(k_size*k_size)
  output = np.zeros((padding_h,padding_w,I.shape[2]),np.uint8)
  for i in range(output.shape[0]):
    for j in range(output.shape[1]):
      for k in range(output.shape[2]):
        if I[i+k_size//2][j+k_size//2][k] == 0 or I[i+k_size//2][j+k_size//2][k] == 255:
          window = I[i:i+k_size,j:j+k_size,k]
          if I[i+k_size//2][j+k_size//2][k] == 0:
            if np.sum(window) == (k_size*k_size - 1) * 255:
              output[i][j][k] = 255
            elif not np.isin(255,window):
              index = np.argwhere(window==0)
              new_window = np.delete(window,index)
              output[i][j][k] = np.median(new_window)
            else:
              output[i][j][k] = np.sum(filter_mean*I[i:i+k_size,j:j+k_size,k])

          if I[i+k_size//2][j+k_size//2][k] == 255:
            if np.sum(I[i:i+k_size,j:j+k_size,k]) == 255:
              output[i][j][k] = 0
            elif not np.isin(0,window):
              index = np.argwhere(window==255)
              new_window = np.delete(window,index)
              output[i][j][k] = np.median(new_window)
            else:
              output[i][j][k] = np.sum(filter_mean*I[i:i+k_size,j:j+k_size,k])
        else:
          output[i][j][k] = I[i+k_size//2][j+k_size//2][k]
  return output
```