SIP Assignment 4

mnp553

March 7, 2022

1 Image filtering

1.1

The filter kernels using correlation are

$$\begin{bmatrix} -1/2 & 0 & 1/2 \end{bmatrix} \text{ and } \begin{bmatrix} -1/2 \\ 0 \\ 1/2 \end{bmatrix}$$

And using convolution, they are

$$\begin{bmatrix} 1/2 & 0 & -1/2 \end{bmatrix} \text{ and } \begin{bmatrix} 1/2 \\ 0 \\ -1/2 \end{bmatrix}$$

In both cases, 0 is the center pixel in the filter kernel.
It will not effect either mean or Gaussian filters since they are both symmetric.

1.2

For the y-derivatives Prewitt filter, the separable form is

$$f * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} = f * \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

For the y-derivatives Sobel filter, the separable form is

$$f * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} = f * \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

The simple finite difference approximation can amplify the noises because noises will make neighbour pixels have a large difference, while for Prewitt and Sobel filter, they can be decomposed into the products of an averaging and a differentiation filter, and thus compute the gradient with smoothing.

1.3

I chose variances of 0.05 and 0.1 to add Gaussian noise to the original image.

```python
eight = imread('/content/drive/MyDrive/eight.tif', as_gray=True)
eightWithNoise1 = random_noise(eight, mode='gaussian', var=0.05)
eightWithNoise2 = random_noise(eight, mode='gaussian', var=0.1)
```

Listing 1: Adding Gaussian noise

```python
def sobel_filter(img):
    eightSobelX = sobel(img, axis=1)
    eightSobelY = sobel(img, axis=0)
    squaredGradientSobel = eightSobelX**2 + eightSobelY**2
    return squaredGradientSobel

def prewitt_filter(img):
    eightPrewittX = prewitt(img, axis=1)
    eightPrewittY = prewitt(img, axis=0)
    squaredGradientPrewitt = eightPrewittX**2 + eightPrewittY**2
    return squaredGradientPrewitt
```

Listing 2: Sobel and Prewitt filters

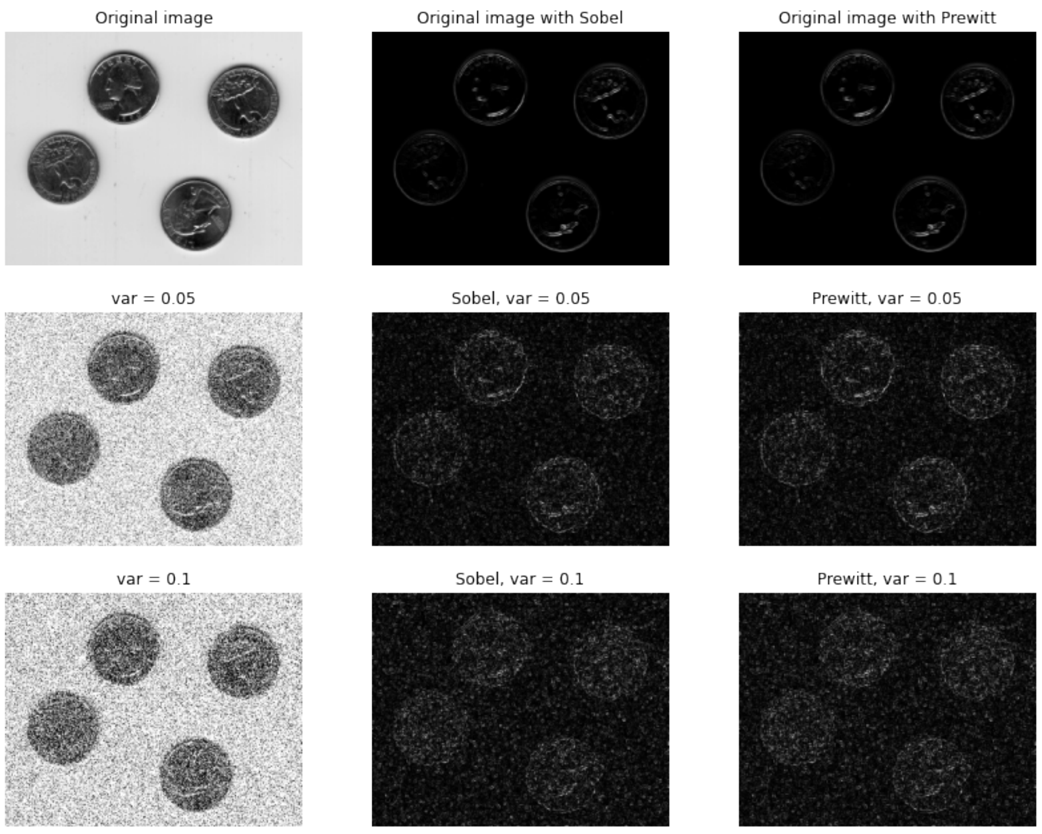

Figure 1: Results

From the results, we can see that the Gaussian noise effects the performance of the both filters and the increase of the variance reduces their ability to detect edges. The reason is that noise affects the computation of gradients. The two filters perform similarly on different variance. The X gradient form is used for detecting vertical edges, and the Y gradient form is used for detecting horizontal edges. Therefore, the squared gradient magnitude image shows both the vertical and horizontal edges.

2 Histogram-based processing

2.1

Since I_1 and I_2 are two constant images with respective values a and b, $C_1(I_1(x,y)) = 1, C_2(I_2(x,y)) = 1$, and $C_1^{(-1)}(1) = a, C_2^{(-1)}(1) = b$.
For every pixel $I(x,y)$,

$$\begin{aligned}\tilde{I}_1(x,y) &= \frac{1}{2}(C_1^{(-1)}(C_1(I_1(x,y))) + C_2^{(-1)}(C_1(I_1(x,y)))) \\ &= \frac{1}{2}(C_1^{(-1)}(1) + C_2^{(-1)}(C_1(1))) \\ &= \frac{a+b}{2}\end{aligned}$$

Follow the same steps and we can also get $\tilde{I}_2(x,y) = (a+b)/2$.
The cumulative histogram of the midway specifications is not equal to the average of the cumulative histograms. Take the same example that $I_1(x,y) = a, I_2(x,y) = b$ and $a < b$. The average of the normalized cumulative histogram is 0 when intensity $i < a$, $1/2$ when $a \leq i < b$ and 1 when $i \geq b$, which is not equal to the normalized cumulative histogram of the constant image with $I(x,y) = (a+b)/2$.

2.2

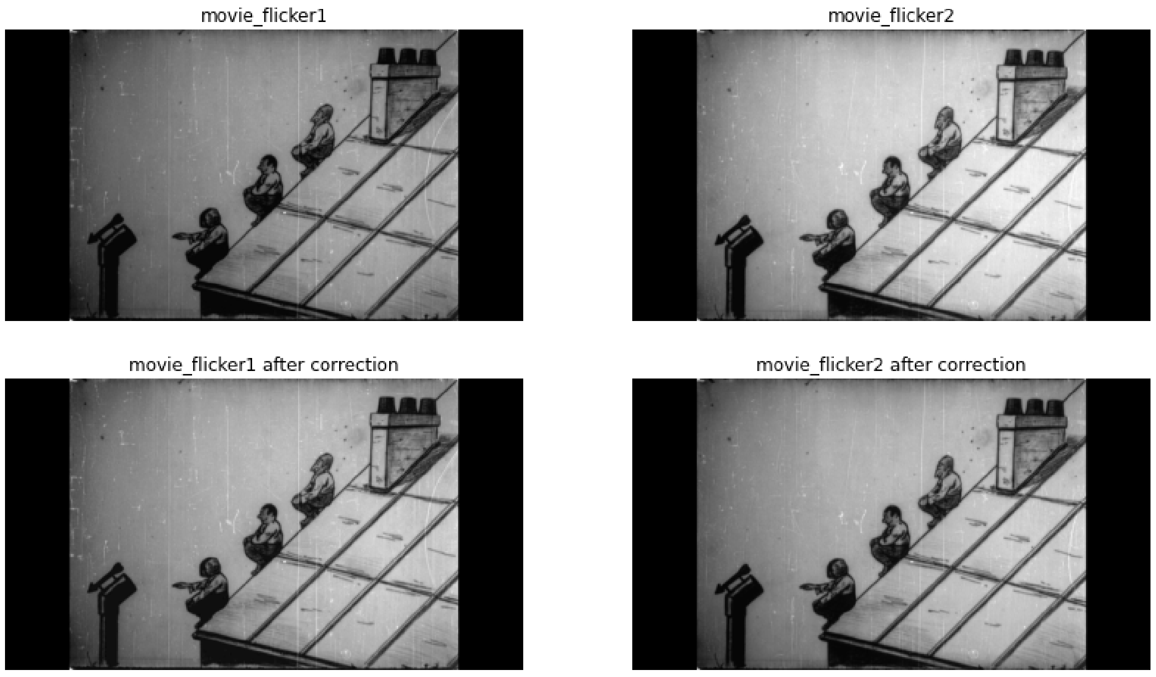
movie_flicker1
movie_flicker2
movie_flicker1 after correction
movie_flicker2 after correction

Figure 2: Original images and resulting images

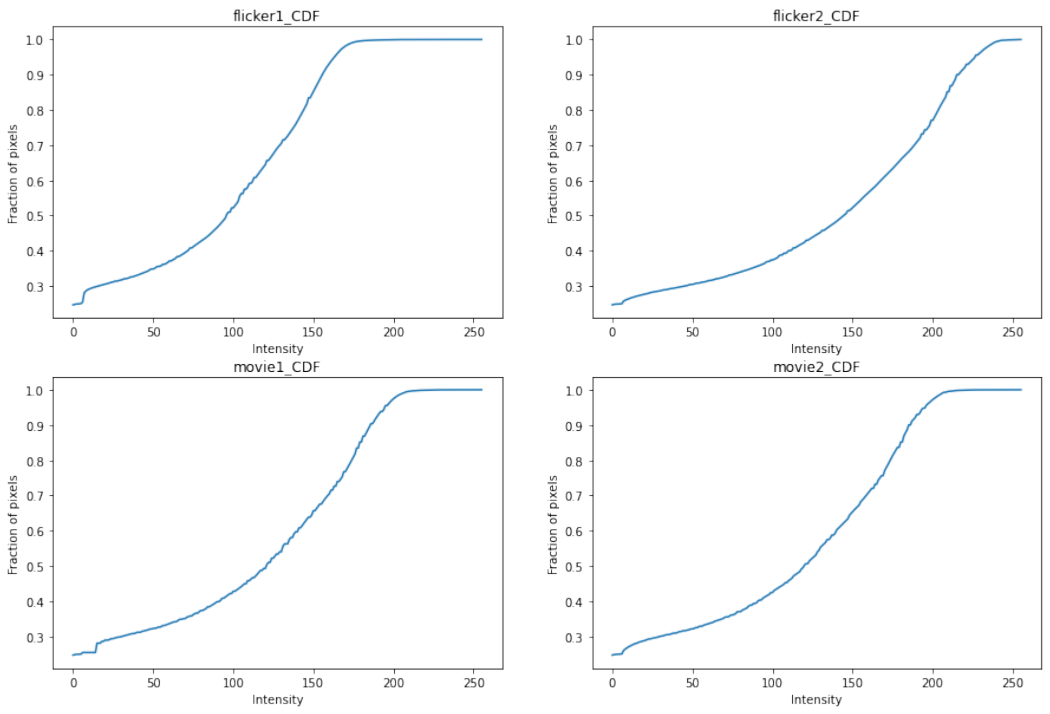

Figure 3: CDF

As can be seen from the resulting images and the change in CDFs, we can see that appyling midway specification, the darker parts are brighter, brighter parts are darker and their brightness is closer, thus correcting flicker effects to some extent.
For an arbitrary number of images, the midway specification is

$$\phi(x) = \frac{1}{n}\left(C_1^{(-1)}(x) + ... + C_n^{(-1)}(x)\right)$$

where the image $\tilde{I}_i = \phi(C_i(I_i))$, $i \in [1, n]$.

3 Fourier transform

3.1

```python
def gaussian_kernel(image,sigma):
    x, y = image.shape
    X = np.linspace(-(x - 1) / 2., (x - 1) / 2., x)
    Y = np.linspace(-(y - 1) / 2., (y - 1) / 2., y)
    gauss_x = np.exp(-0.5 * np.square(X) / np.square(sigma))
    gauss_y = np.exp(-0.5 * np.square(Y) / np.square(sigma))
    kernel = np.outer(gauss_x, gauss_y)
    return kernel / np.sum(kernel)

def scale_fft(img, sigma):
    fft_img = fft.fftshift(fft.fft2(img))
    fft_kernel = gaussian_kernel(img,sigma)
    fft_new_img = fft_img * fft_kernel
    new_img = fft.ifft2(fft.ifftshift(fft_new_img))
    return new_img
```

Listing 3: Implementation of the function scale_fft

My choices for *sigma* are 1, 10, and 20. See Figure 4 for the results of applying the function to **trui.png**.

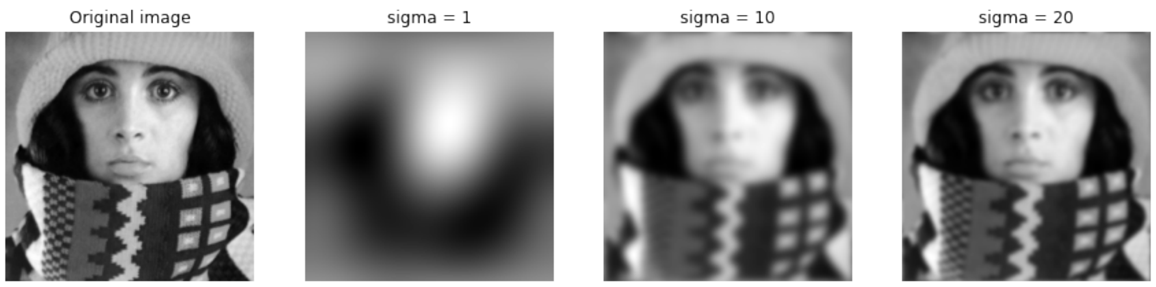

Figure 4: Original image and results with different σ

From the Convolution theorem, the convolution of two functions in real domain is the same as the product of their respective Fourier transforms in frequency domain, i.e. $\mathcal{F}\{(f * g)(x)\} = F(u)G(u)$. Therefore, the convolution can be implemented as in Listing 4.

3.2

```
def convolution_fft(image, kernel):
    fft_image = fft.fft2(image)
    fft_kernel = fft.fft2(kernel, image.shape)
    fft_new_image = fft_image * fft_kernel
    new_image = fft.ifft2(fft_new_image, image.shape)
    return new_image

def image_derivative(image, x, y):
    kernel_x = [[1, -1]]
    kernel_y = [[1], [-1]]
    derivative = image
    for i in range(x):
      derivative = convolution_fft(derivative, kernel_x)
    for i in range(y):
      derivative = convolution_fft(derivative, kernel_y)
    return derivative
```

Listing 4: The function that returns the partial derivative of an image

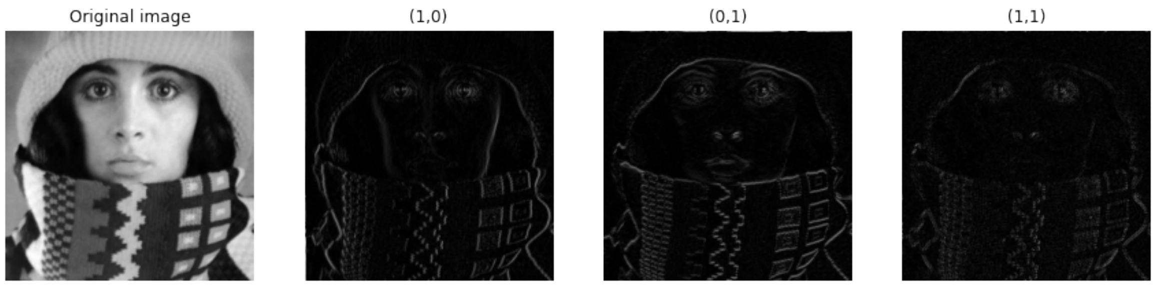

Figure 5: Original image and results with different combinations of orders of derivatives

4 Morphology

4.1

The original binary image and resulting images after dilation are as follows:

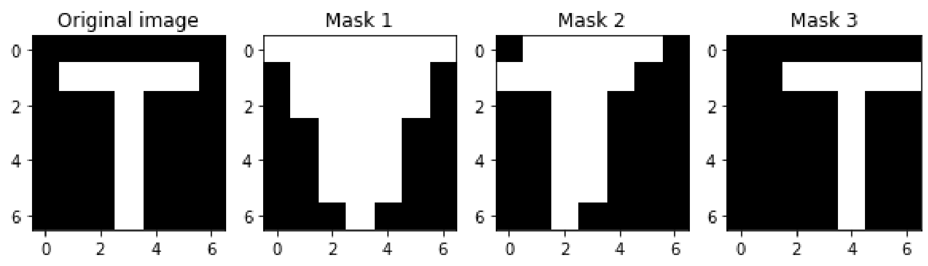

Figure 6: Input and outputs

```
result_1 = binary_dilation(t, mask_1) # t is the input image
result_2 = binary_dilation(t, mask_2)
result_3 = binary_dilation(t, mask_3)
```

Listing 5: Dilation

i. SE does not need to be symmetric.
Here the pixel at the exact center of the image is selected as the center pixel. See Figure 7.

ii. SE does not need to have an odd number of pixels in both directions.
Here the pixel in the bottom right corner of the image is selected as the center pixel. See Figure 7.

iii. SE does not need to have the middle pixel set to true/on.
Here the pixel at the exact center of the image is selected as the center pixel. See Figure 7.

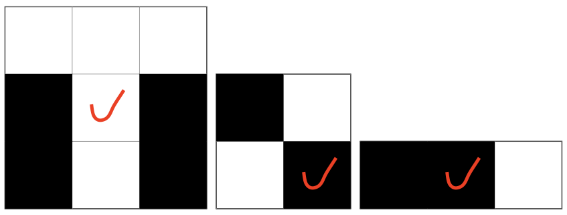

Figure 7: center pixel

4.1.1

Zero padding is necessary for interchange.

```
mask_1 = np.pad(mask_1, ((10,10),(10,10)), constant_values=(0,0))
mask_2 = np.pad(mask_2, ((10,10),(10,10)), constant_values=(0,0))
mask_3 = np.pad(mask_3, ((10,10),(10,10)), constant_values=(0,0))

result_1 = binary_dilation(mask_1, t)
result_2 = binary_dilation(mask_2, t)
result_3 = binary_dilation(mask_3, t)
```

Listing 6: Padding and interchange

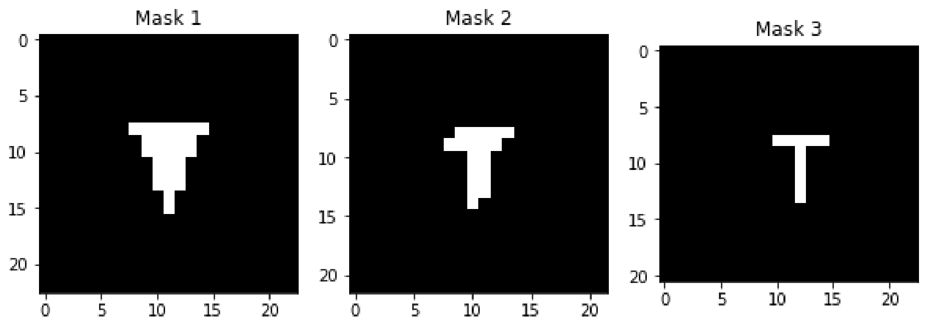

Figure 8: results after input image and SE are interchanged

The results after input image and SE are interchanged is exactly the same as before, indicating that SE and input images can be interchanged. According to the definition of Dilation,

$$X \oplus B = \{p \in \Omega : p = x + b, x \in X \text{ and } b \in B\} = \{p \in \Omega : p = b + x, b \in B \text{ and } x \in X\} = B \oplus X$$

showing the interchange does not affect the results.
The benefit is lower computation cost since the resulting image is the same whether or not they are interchange.