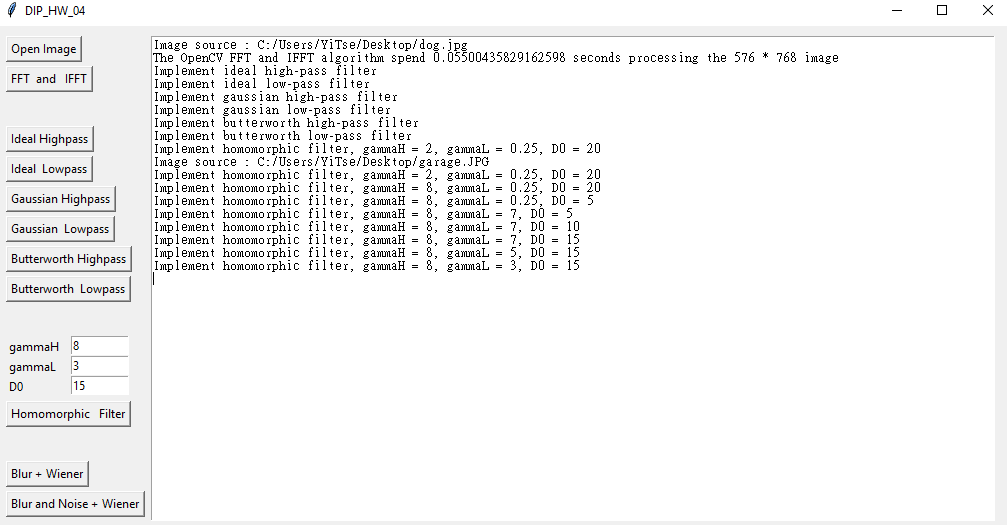
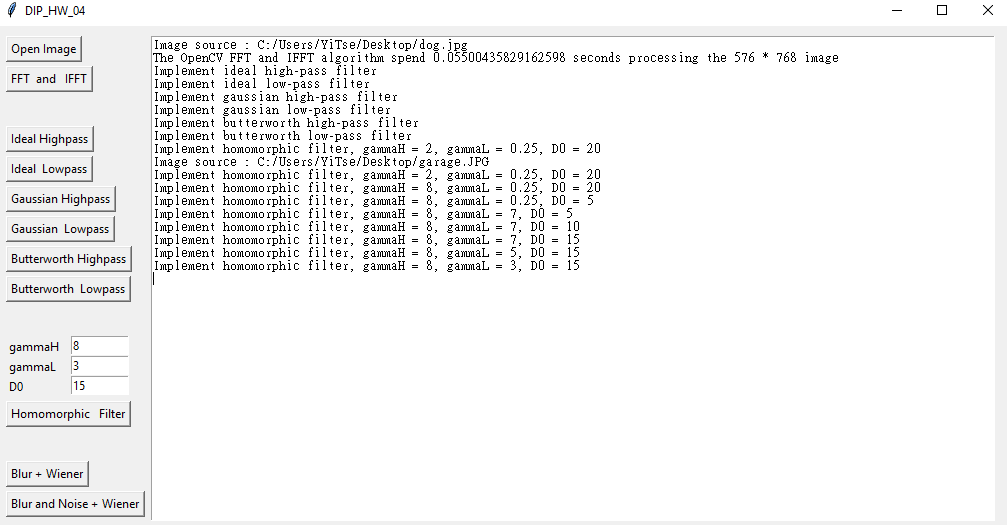
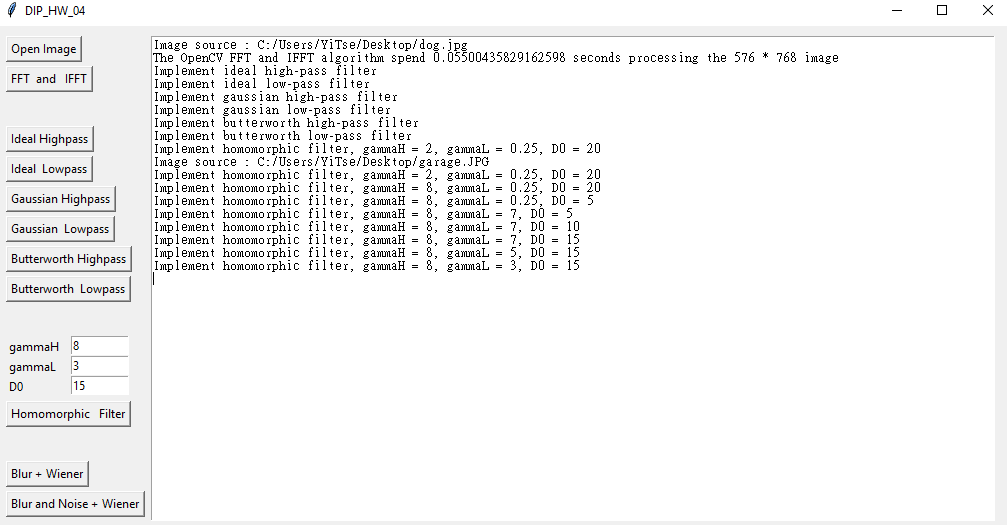
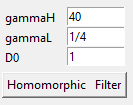
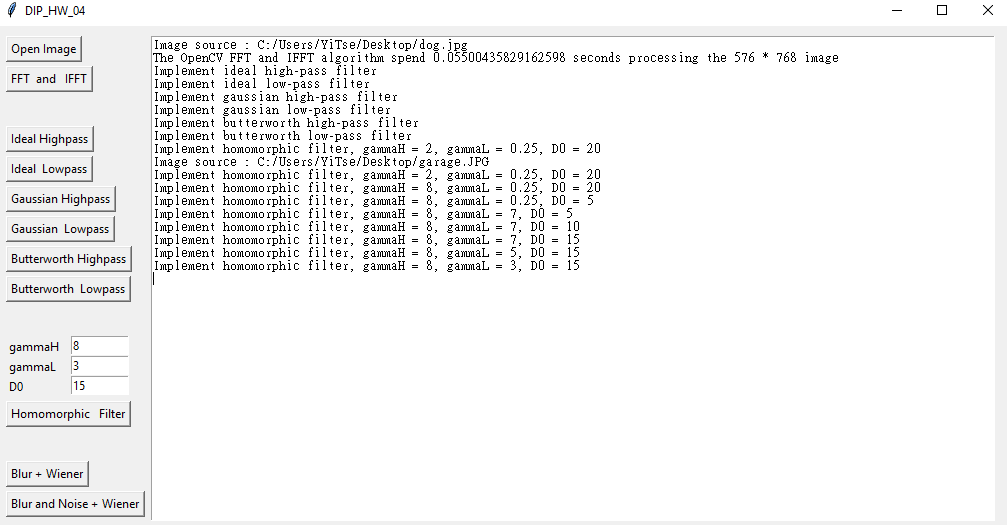
Digital Image Processing HW04 Report

生機碩一 R09631007 吳乙澤

**一、軟體簡介**

本次作業以 python 配合 tkinter 寫成。一共有用到 os、opencv、time、random、numpy、tkinter、matplotlib、math 等多個函數庫。見下圖，軟體介面分兩部分，一是輸入區域，二是輸出區域。輸入區域有多個按鈕，需先按下 open image 並選擇圖片後，才能做後續濾波。按下濾波按鈕後，如 Ideal Highpass，程式會自動地另開視窗，並將圖片顯示在該視窗中。輸出區域則是介面右方的白色方框，在使用者按下按鈕後，會依按鈕性質，顯示相對應的提示訊息，如圖片路徑、使用者所做動作、homomorphic filter 的係數等。最後，請注意，在輸入 homomorphic filter 係數時，要以整數或分數的形式輸入，若以小數形式輸入會導致程式出錯。



**二、演算法說明與結果討論**

本作業有四個功能需完成。以下依序說明這些功能的演算法並討論結果。

* 1. **Fourier Transform**

見下方的FFT程式碼。我利用cv2.dft做傅立葉轉換，之後以np.fft.fftshift做平移，讓低頻訊號移至中心。再以cv2.magnitude，對實部與虛部做平方後相加，並利用log調整對比，就能得到頻譜圖。相位部分，用np.angle便能得到。逆傅立葉轉換也是類似步驟。在此不加贅述。

# FFT

imgGray = cv2.imread(path, cv2.IMREAD\_GRAYSCALE)

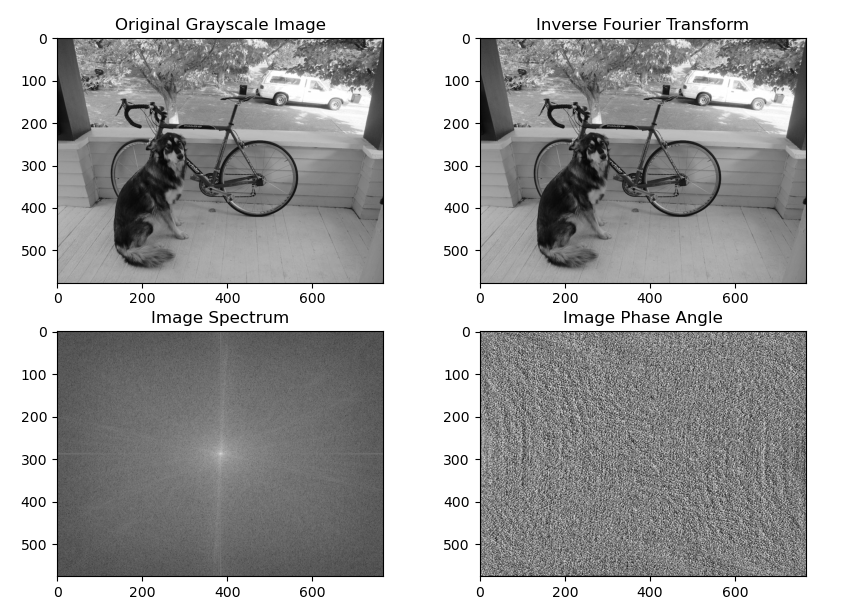
fft = cv2.dft(np.float32(imgGray),flags=cv2.DFT\_COMPLEX\_OUTPUT)

fftShift = np.fft.fftshift(fft)

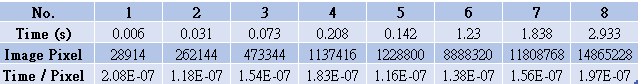
Spectrum=np.log(cv2.magnitude(fftShift[:,:,0],fftShift[:,:,1]))

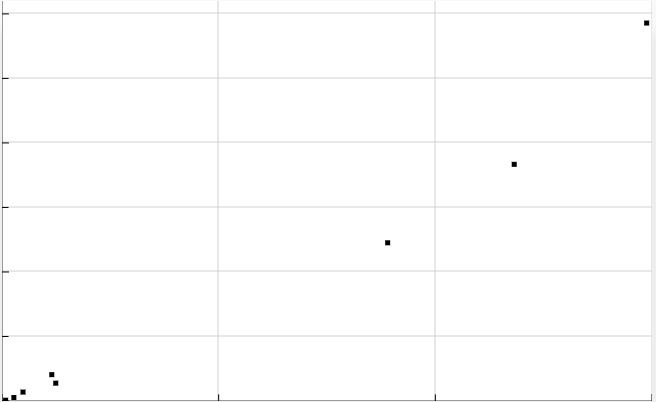
Phase = np.angle(np.fft.fft2(imgGray))

見下圖，以肉眼比較轉換前後的圖片，分不出有何區別。不過，若將兩圖片相減，則會發現他們是不一致的。想來這是因為演算法的問題，而不是理論出錯。因為在數學上，傅立葉正轉換與傅立葉逆轉換，是相互對應的關係。



我以python 函數庫的函數 time，計算傅立葉正轉換與傅立葉逆轉換的運算時間。得到時間後，一律以四捨五入法取至小數點後三位。158\*183的圖片花了0.006秒，512\*512的圖片花了0.031秒，688\*688的圖片花了0.073秒。我一共測試了八張不同大小的圖片，並將運算時間、圖片像素總數、每個像素所花時間共三個參數，放在一個表格中做對照。如下所示，越大張的圖片，運算時間有越長的趨勢。並且，將表格中的像素數作為X軸，運算時間作為Y軸作圖。可以發現，到一定大小後，運算時間突然陡峭上升。代表該演算法的時間複雜度為非線性，推測快速傅立葉轉換的時間複雜度為n\*log(n)或n^2。





* 1. **Highpass and Lowpass Filtering of Images**

該題皆是對已平移的頻譜做濾波處理後，再平移回去後做逆傅立葉轉換，到空間域顯示出來。見下方程式碼，這裡只列出理想濾波器的部分，其他濾波器類似該操作，故不重複解釋。不同的濾波器，只是 for loop 內部的處理不一樣而已。可參照課本的公式進行處理即可。另外，R 便是cut-off frequency，更改 R 便能改變濾波結果。

# Do ideal highpass

R = 60

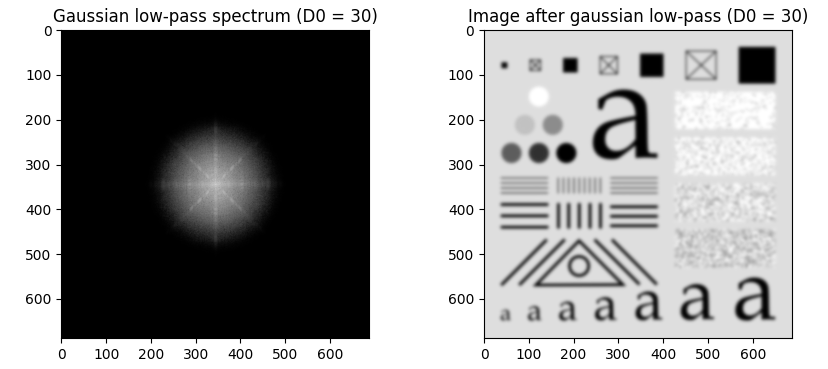
for i in range(imgGray.shape[1]):

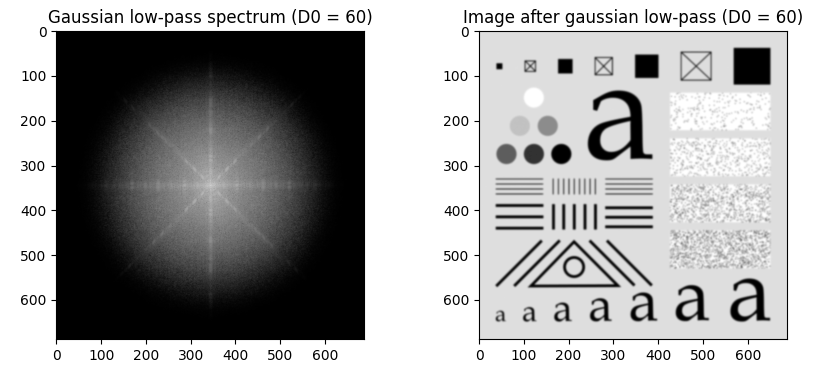
for j in range(imgGray.shape[0]):

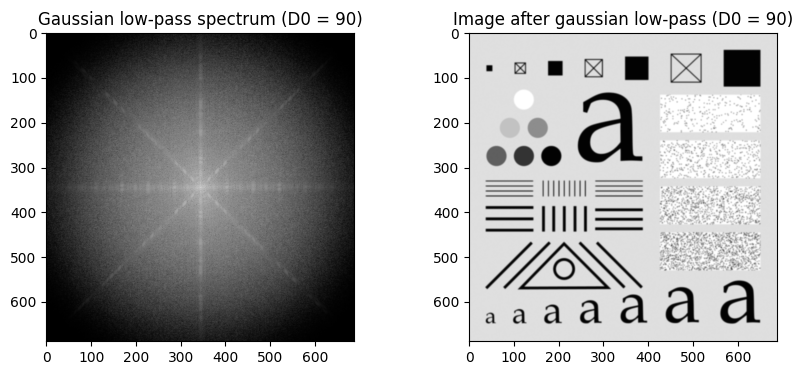
if (i-centerX)\*(i-centerX)+(j-centerY)\*(j-centerY) <= R\*R:

fftShift\_IdealHighPass[j,i] = 0

結果圖的部分，由於圖形眾多，這裡只列出不同cut-off frequency對高斯低通濾波器的不同影響。見下圖，D0是高斯濾波的cut-off frequency。觀察頻譜圖，中央是低頻訊號，周圍是高頻訊號。D0越小，截掉的高頻較多，模糊的效果越顯著；D0越大，截掉的高頻較少，模糊的效果越不顯著。







* 1. **Homomorphic Filtering**

該部分利用 tkinter 的 frame 和 entry 功能，實現對使用者友好的介面，可讓使用者自訂 Homomorphic filter 的三個參數，rH、rL、D0。濾波的公式，則參照課本的公式，對平移後的頻譜相乘即可。

# User Input : GammaH of Homomorphic Filtering

gammaH\_frame = tk.Frame(window)

gammaH\_frame.place(x = 10, y = 310)

gammaH\_label = tk.Label(gammaH\_frame, text = 'gammaH   ')

gammaH\_label.pack(side = tk.LEFT)

gammaH\_ = tk.Entry(gammaH\_frame, width = 9)

gammaH\_.pack()

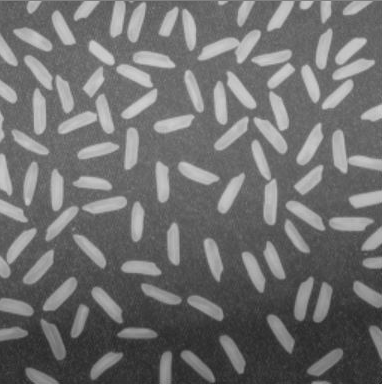
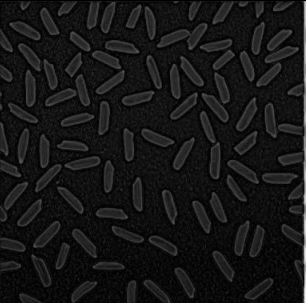
# Do gaussian highpass

for i in range(imgGray.shape[1]):

for j in range(imgGray.shape[0]):

fftShift\_homomorphic[j,i] = fftShift\_homomorphic[j,i] \* ((gammaH - gammaL)\*(1 - exp(-c\*((i-centerX)\*(i-centerX)+(j-centerY)\*(j-centerY))/ D0/D0)) + gammaL)

另外，查過課本和網路，若是調整 rL < 1，rH >= 1，則該濾波器能強化對比、動態範圍壓縮。進而達到銳化邊緣，將不均勻光平均化的效果。見下圖，經過 Homomorphic filter之後，背景的顏色變得單一，因此做後續的二值化會簡單許多。

   
rH = 2, rL = 0.2, D0 = 60 (C = 2)

rH = 2, rL = 0.2, D0 = 1 (C = 1)

* 1. **The restoration of Motion Blurred and Noise Image**

該題的 motion blur、inverse filter、wiener filter 皆是照課本公式作之。不過不知為何，motion blur 出來的結果和課本上的有所出入。或許是我對公式理解錯誤，或是實作時寫錯程式吧。見下方程式碼，我照課本(5-77)式，利用euler's formula 對 exponential 拆解後，只留下實數項，也就是cos項與剩餘公式相乘。另外，為防止公式除到零，在 i和 j等於零時，使 fftShift\_blur 保持原樣。

# Do Motion Blur

a = 0.1

b = 0.1

T = 1

for i in range(0, imgGray.shape[0]):

for j in range(0, imgGray.shape[1]):

if i == 0 and j == 0 :

fftShift\_blur[i,j] = fftShift\_blur[i,j]

else :

           motionBlur = sin(pi\*(i\*a + j\*b))\*T/(pi\*(i\*a + j\*b))\*cos(-pi\*

(i\*a + j\*b))

           fftShift\_blur[i,j] = fftShift\_blur[i,j] \* motionBlur

見下方結果圖，motion blur的結果和課本上的有所出入，但也有殘影及部分模糊的效果。對於只有加入motion blur干擾的圖片，inverse filter去motion blur的效果比wiener好。將經過濾波後的圖片相減，可以發現兩圖有所差異。差異主要呈現在模糊的效果上。另一方面，觀察有加入高斯躁點的圖片，wiener filter的還原效果比inverse filter好上許多。inverse filter的還原效果基本上是一團亂。將還原後的效果相減，可以得知兩圖主要差在斜線。猜測該斜線代表高斯躁點的訊息。查過網路和課本，wiener filter比inverse filter適合重建有躁點的圖片。

