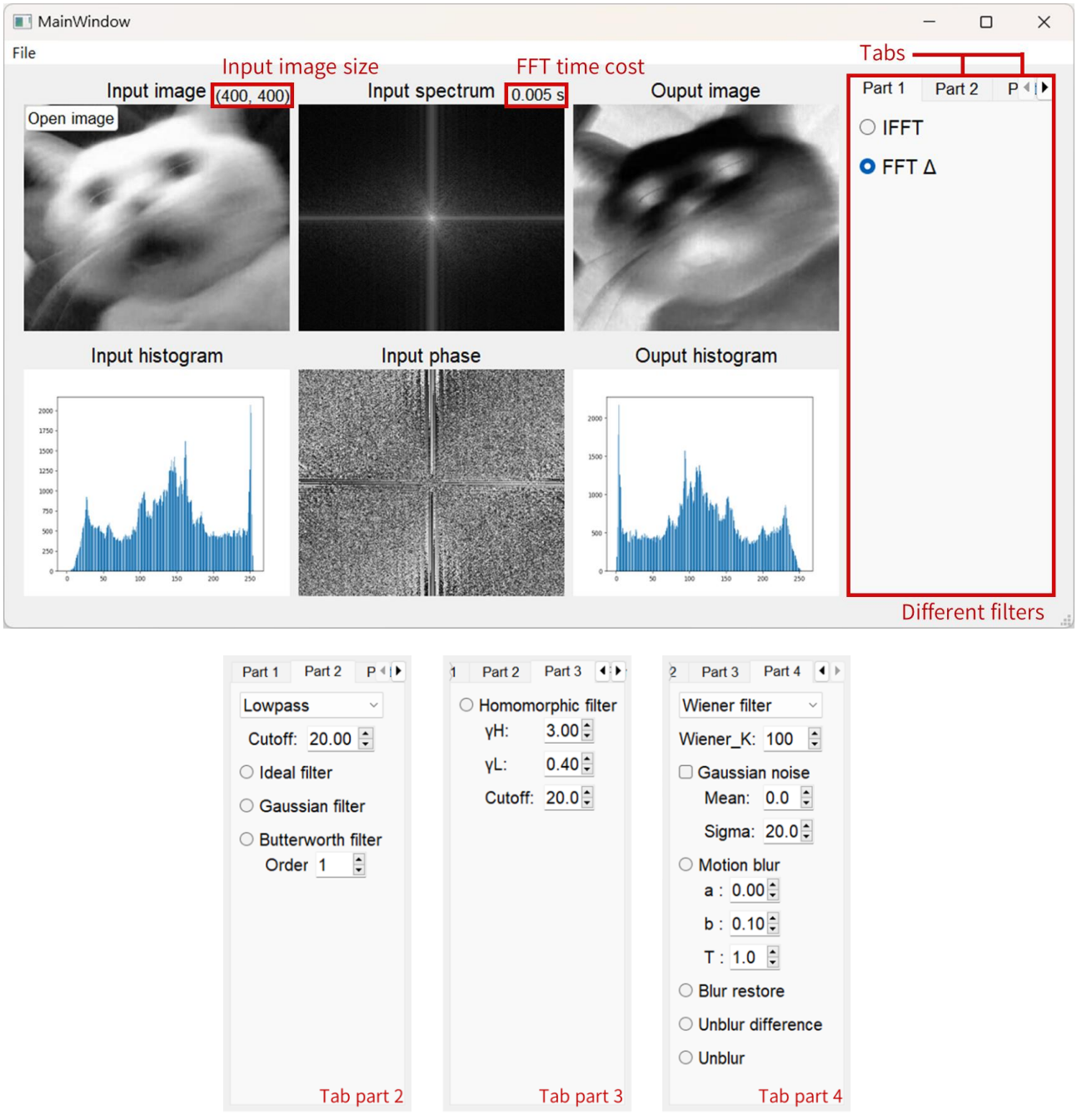
Principles and Applications of Digital Image Processing

Homework 4

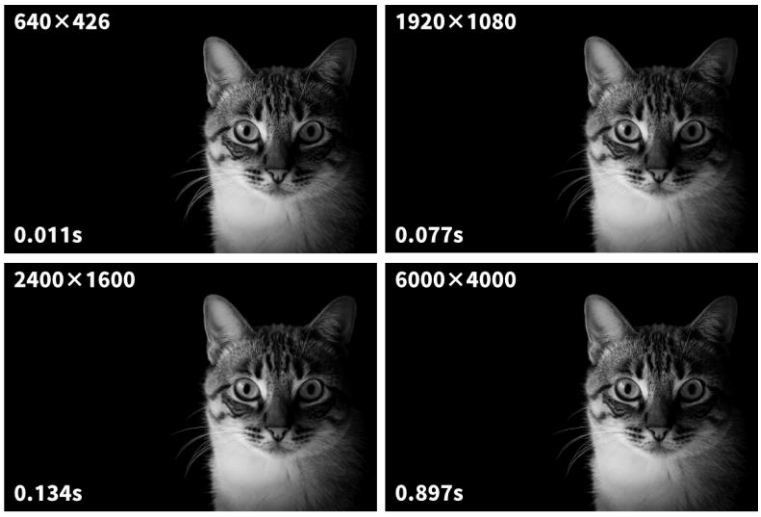
GUI



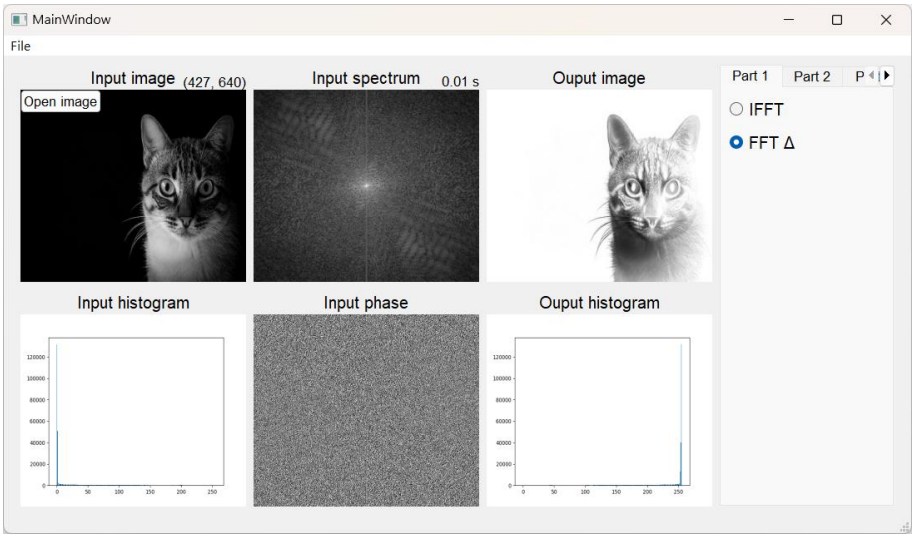
圖一、介面使用說明。

Part 1: (25%)

Make an analysis of the effect of image size on the processing time of your Fourier transform program.



圖二、不同大小影像作 FFT 所耗之時間。影像取自 Unsplash Pacto Visual。



圖三、原始影像減去經 FFT 和 IFFT 之影像。

執行檔在使用者輸入影像後，會自動顯示輸入影像的 spectrum 和 phase 圖。使用者可以透過 part 1 tab 使用 IFFT 和 IFFT difference 功能（圖一）。IFFT 是將 frequency domain 之影像反轉換的影像。IFFT difference 是將原始影像減去經 IFFT 的影像。影像經 FFT 成 spatial domain 的前後皆經過正歸化。觀察圖二可發現越大的影像作 FFT 所需的時間越多。此外，看圖三可發現經過 IFFT 的 FFT 影像與原始影像並不相同，雖然其看起來相似。實作如下：

```
def fft(img, img_path):
    time_start = time.time()
    fshift = fft2(img)
    fft_time = time.time()-time_start
    spectrum = np.abs(fshift)
    fmin = np.log(1+np.abs(spectrum.min()))
    fmax = np.log(1+np.abs(spectrum.max()))
    ynew = 255*(np.log(1+abs(spectrum))-(fmin))/(fmax-fmin)
    spectrum_path = save_output(ynew, img_path, 'spectrum')
```

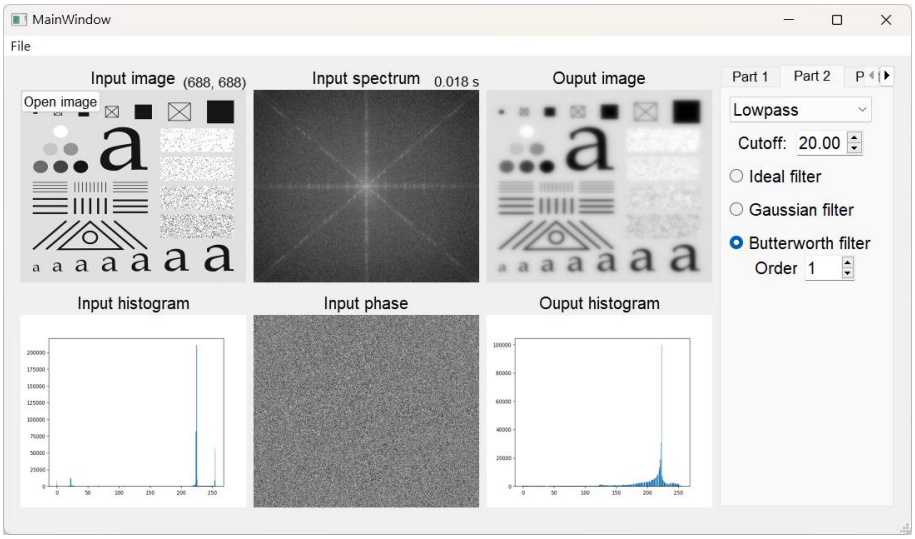
```
    phase = np.arctan(fshift.imag/fshift.real)
    phase_path = save_output(normalize2(phase), img_path, 'phase')
    return fshift, spectrum_path, phase_path, fft_time


def ifft(fshift, img_path='./', save=False):
    f_ishift = np.fft.ifftshift(fshift)
    img_back = np.fft.ifft2(f_ishift)
    img_back = normalize2(img_back)
    if save:
        save_path = save_output(img_back.real, img_path, 'back')
        return img_back.real, save_path
    return img_back.real


def fft_d(img_arr, fshift, img_path):
    img_back = ifft(fshift)
    img_d = img_arr - img_back.real
    img_d = normalize2(img_d)
    save_path = save_output(img_d, img_path, 'd')
    return img_d.real, save_path
```

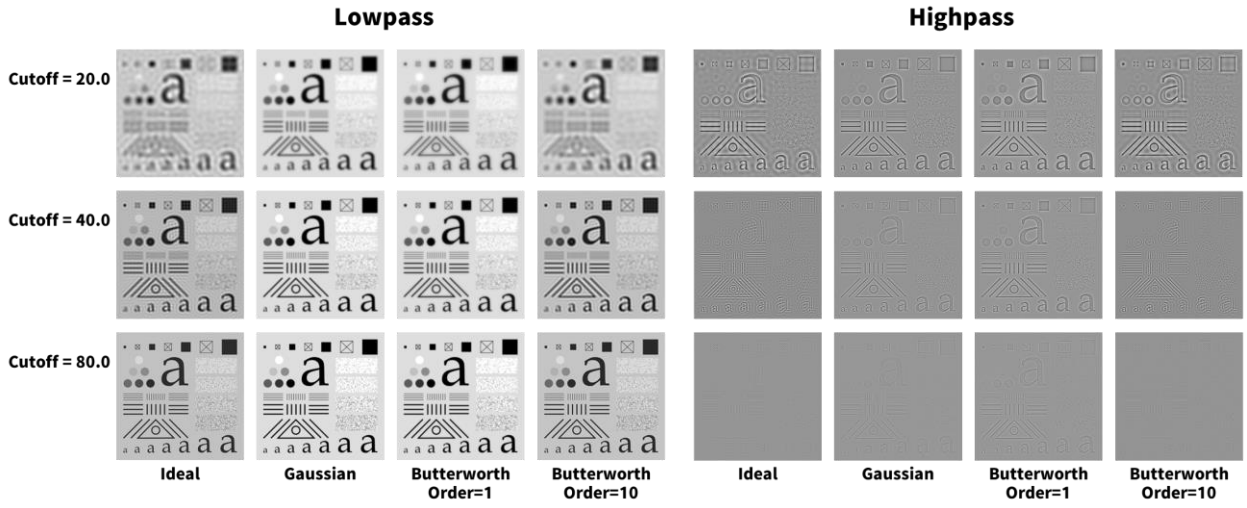
Part 2: (25%)

Design a program for highpass and lowpass filtering of images.



圖四、高/低通濾波影像之程式。

Discuss the effect of cut-off frequency on the processed image.



圖五、用不同 cutoff 的 lowpass/highpass filter 之影像。

看圖五，用 Ideal filter 可看見明顯的水紋狀，而 Gaussian 則較為平滑。Butterworth 兼具兩種特性，當 order 小時類似 Gaussian，order 大時類似 Ideal。再來觀察不同 cutoff 對影像濾波的影響。在低通濾波時，Cutoff 越小濾波效果越明顯，當 cutoff 大時則效果不彰。而在高通濾波，小的 cutoff 可以將影像中的線條、形狀良好的顯現，大的 cutoff 只會保留較粗、較明顯的線條。實作如下：

```
def ideal_filter(fshift, cutoff, f_pass, img_path):
    p, q = fshift.shape
    filter_h = np.zeros((p, q), dtype='complex_')
    # Lowpass filter
    if f_pass == 'Lowpass':
        for i in range(p):
            for j in range(q):
                distance = pow((pow(i-p/2, 2)+pow(j-q/2, 2)), 0.5)
```

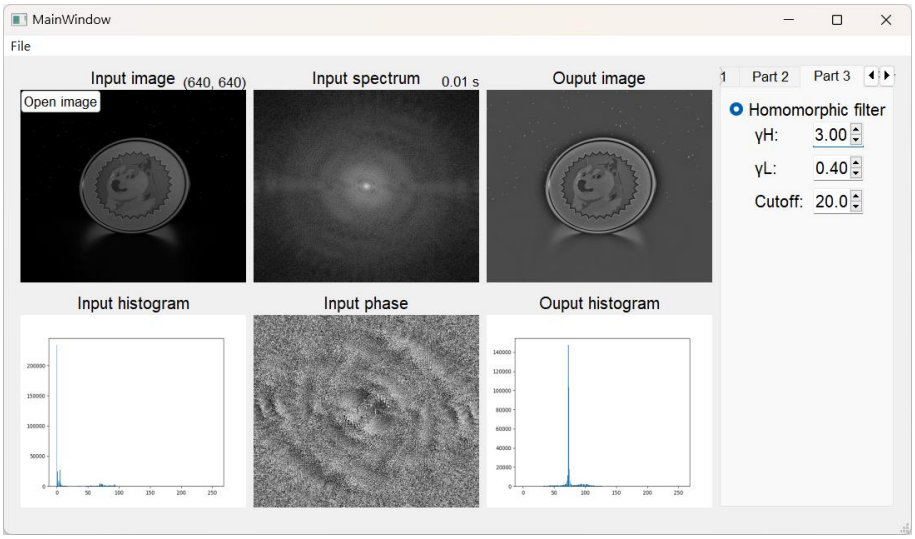
```
                if distance > cutoff:
                    filter_h[i][j] = 1
    gshift = fshift * filter_h
    img_back = ifft(gshift)
    img_back = normalize2(img_back)
    save_path = save_output(img_back, img_path, 'ideal')
    return img_back, save_path


def gaussian_filter(fshift, cutoff, f_pass, img_path):
    p, q = fshift.shape
    filter_h = np.zeros((p, q), dtype='complex_')
    # Lowpass filter
    for i in range(p):
        for j in range(q):
            distance = pow((pow(i-p/2, 2)+pow(j-q/2, 2)), 0.5)
            filter_h[i, j] = np.exp((-pow(distance, 2))/(2*pow(cutoff, 2))
    # Highpass filter
    if f_pass == 'Highpass':
        filter_h = 1-filter_h
    gshift = fshift * filter_h
    img_back = ifft(gshift)
    img_back = normalize2(img_back)
    save_path = save_output(img_back, img_path, 'gaussian')
    return img_back, save_path
```

```
def butterworth_filter(fshift, cutoff, n, f_pass, img_path):
    p, q = fshift.shape
    filter_h = np.zeros((p, q), dtype='complex_')
    # Lowpass filter
    for i in range(p):
        for j in range(q):
            distance = pow((pow(i-p/2, 2)+pow(j-q/2, 2)), 0.5)
            filter_h[i, j] = 1/(1+pow(distance/cutoff, 2*n))
    # Highpass filter
    if f_pass == 'Highpass':
        filter_h = 1-filter_h
    gshift = fshift * filter_h
    img_back = ifft(gshift)
    img_back = normalize2(img_back)
    save_path = save_output(img_back, img_path, 'butter')
    return img_back, save_path
```

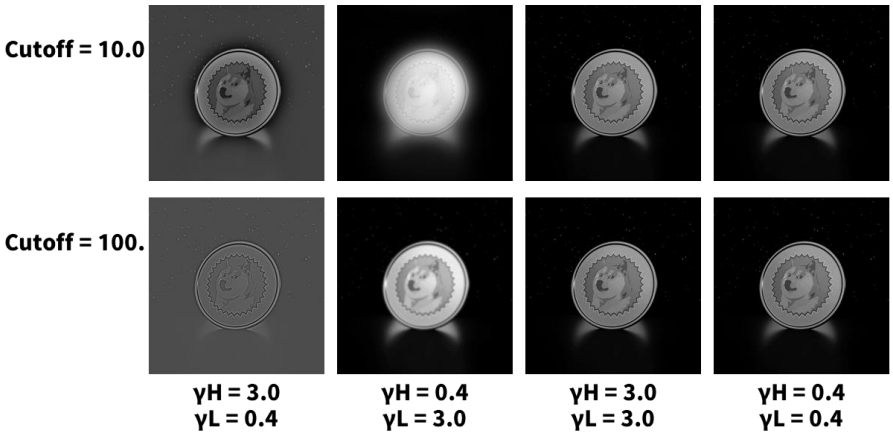
Part 3: (25%)

Design an image processing program for homomorphic filtering.



圖六、Homomorphic 濾波程式。

Discuss the effect of these parameters on the processed image in your report.

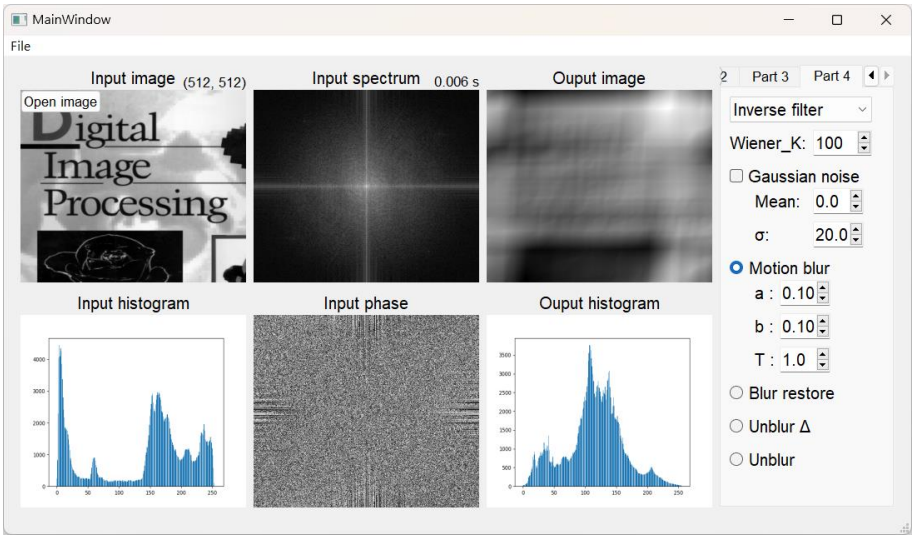


圖七、經 homomorphic filtering 之影像。

觀察圖七，可發現當 γ_H 大於一 γ_L 小於一時，濾波的效果與高通濾波相似，而濾波效果與 cutoff 大小相關。而當 γ_H 小於一 γ_L 大於一時，濾波的效果與低通濾波相似。當 γ_H 與 γ_L 相等時，看起來沒有濾波效果。其實作過程與 part 2 的 filter 雷同，區別在於公式不同，詳細程式碼可參考./code/img_processing.py: `def homo_filter()`。

Part 4: (25%)

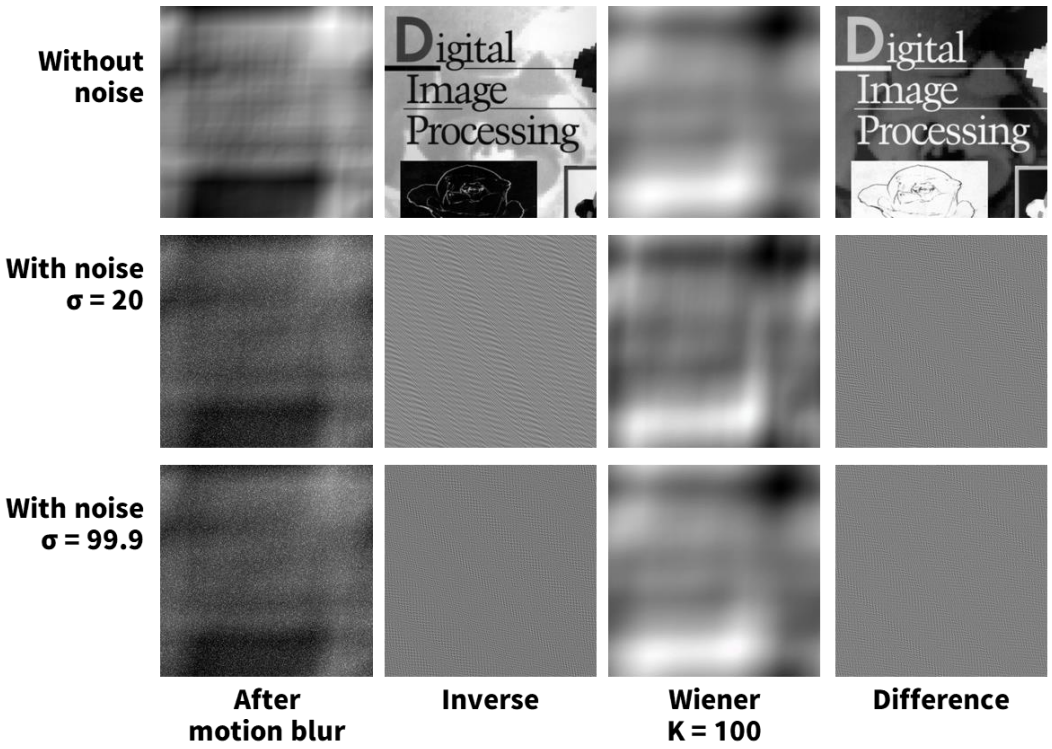
Design an image processing program to create a motion blurred.



圖八、Motion 濾波程式。

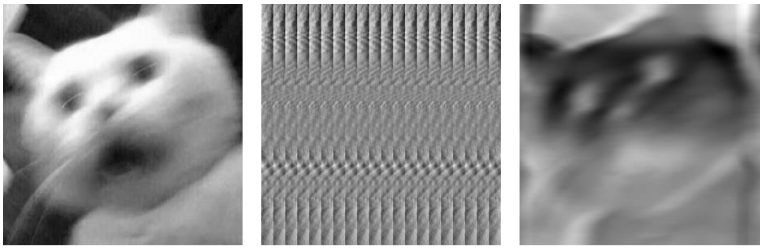
使用者可以使用右上角方形下拉選單選取 Inverse/Wiener filter。勾選 Gaussian noise 將影像加上 Gaussian noise。Motion blur 對影像做動態模糊。Blur restore 將原始影像做動態模糊後，用使用者選取的 filter 做復原影像。Unblur Δ 會計算經動態模糊之影像用 Inverse filter 和 Wiener filter 做 restore 的結果差。Unblur 用使用者選取得 filter 對原始影像做 restore。

Comment on the performance of the two filters for removing motion blur.



圖八、經不同動態濾波之影像。

Wiener filter 在 K=0 時與 inverse filter 相同，因此沒有用 K=0 之 Wiener。看圖八可了解當影像沒有雜訊時，Wiener 之 K 應該調整成 0，否則結果非常糟糕，呈現類似經過低通濾波的影像。將 Wiener 減去 Inverse 後得到似經高通濾波之影像。而當影像被賦予雜訊時，Inverse filter 變得完全不能使用。反觀 Wiener 尚可觀察到原始影像的粗略輪廓。此時 Wiener 減去 Inverse 後得到似雜訊的影像。當將雜訊的標準差調高時，Wiener 的結果變得模糊。為了取得去模糊，可以用更高的 K 值。



Original | **Inverse** a = 0, b = 0.05, T = 1 | **Wiener** K = 100, a = 0, b = 0.05, T = 1

圖九、經不同動態濾波之網路影像。

嘗試將網路上的影像用動態濾波，還原未晃動的影像。觀察圖九可發現原始網路影像在拍攝時包含雜訊，Inverse filter 難以還原貓的輪廓，而 Wiener filter 免強可得到貓的原樣。實作如下：

```python
def noise_filter(img_arr, mean, sigma, img_path, save=True):
    noise = np.random.normal(mean, sigma, img_arr.shape)
    noise = (noise-mean)/(2.5*sigma)
    gaussian_out = np.clip(noise, 0, 1)
    gaussian_out *= 255
    if save:
        img_back = img_arr+gaussian_out
        img_back = normalize2(img_back)
        save_path = save_output(img_back, img_path, 'noise')
        return img_back, save_path
    # img_arr is gshift if save=False
    gau_shift = fft2(gaussian_out)
    return (img_arr+gau_shift)


def motion_filter(fshift, a, b, t, img_path, save=True):
    p, q = fshift.shape
    filter_h = np.zeros((p, q), dtype='complex128')
    for u in range(p):
        for v in range(q):
            d = np.pi * (u*a+v*b)
            if d == 0:
                d = 1
            # d = np.pi * ((u+1)*a/10+(v+1)*b/30)
            filter_h[u, v] = (t/d) * np.sin(d)*np.exp(-1j*d)
    hshift = np.fft.fftshift(filter_h)
    gshift = fshift * hshift
    if save:
        img_back = ifft(gshift)
        img_back = normalize2(np.abs(img_back))
        save_path = save_output(img_back, img_path, 'motion')
        return img_back, save_path
    return gshift, hshift
```

```python
def unblur_filter(gshift, way, hshift, K, img_path):
    if way == 'Inverse filter':
        f_hat = gshift/hshift
    elif way == 'Wiener filter':
        h2 = (hshift.imag)*(hshift.real)
        f_hat = ((1/hshift)*h2/(h2+K))*gshift
    img_back = ifft(f_hat)
    img_back = normalize2(img_back)
    save_path = save_output(img_back, img_path, 'unblur')
    return img_back, save_path


def unblur_d_filter(gshift, hshift, K, img_path):
    f_inverse = gshift/hshift
    h2 = (hshift.imag)*(hshift.real)
    f_wiener = (1/hshift*h2/(h2+K))*gshift
    f_hat = f_wiener - f_inverse
    img_back = ifft(f_hat)
    img_back = normalize2(img_back)
    save_path = save_output(img_back, img_path, 'unblurD')
    return img_back, save_path


def motion_unblur_filter(gshift, a, b, t, way, K, img_path):
    _, hshift = motion_filter(gshift, a, b, t, img_path, save=False)
    if way == 'Inverse filter':
        f_hat = gshift/hshift
    elif way == 'Wiener filter':
        h2 = (hshift.imag)*(hshift.real)
        f_hat = (1/hshift*h2/(h2+K))*gshift
    img_back = ifft(f_hat)
    img_back = normalize2(img_back)
    save_path = save_output(img_back, img_path, 'restore')
    return img_back, save_path
```