1. **Please use Inverse Filter and Wiener Filter to correct the image corrupted severe turbulence of the assigned image, 'Fig5.25.jpg', and print out the source code and the corrected image? (40)**

Original Picture:

Inverse Filtering Code:

```
clc;
clear all;
close all;
image=im2gray(imread('Fig5.25.jpg'));
image=imresize(image,[256 256])

x=fspecial('gaussian',260,2);
y=(imfilter(image,x,'circular'));

F = zeros(size(image));
G = fftshift(fft2(y));

H = fftshift(fft2(x));

R=55;

for i=1:size(image,2)
    for j=1:size(image,1)
        di = i - size(image,2)/2;
        dj = j - size(image,1)/2;
        if di^2 + dj^2 <= R^2;
        F(j,i) = G(j,i)./H(j,i);
        end
    end
end

inverse_filtered = abs(ifftshift(ifft2(F)));
figure,imshow(inverse_filtered, []);
```

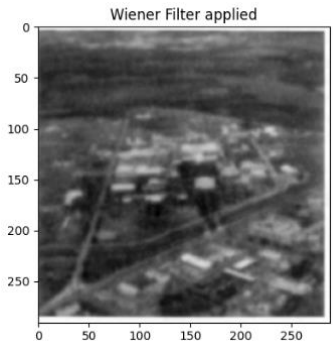
Inverse Filter Code Result:

Wiener Filtering Code:

```python
import os
import numpy as np
from numpy.fft import fft2, ifft2
from scipy.signal import gaussian, convolve2d
import matplotlib.pyplot as plt
import cv2

def blur(img, kernel_size=3):
    dummy = np.copy(img)
    h = np.eye(kernel_size) / kernel_size
    dummy = convolve2d(dummy, h, mode='valid')
    return dummy

def add_gaussian_noise(img, sigma):
    gauss = np.random.normal(0, sigma, np.shape(img))
    noisy_img = img + gauss
    noisy_img[noisy_img < 0] = 0
    noisy_img[noisy_img > 255] = 255
    return noisy_img

def wiener_filter(img, kernel, K):
    kernel /= np.sum(kernel)
    dummy = np.copy(img)
    dummy = fft2(dummy)
    kernel = fft2(kernel, s=img.shape)
    kernel = np.conj(kernel) / (np.abs(kernel) ** 2 + K)
    dummy = dummy * kernel
    dummy = np.abs(ifft2(dummy))
    return dummy
```

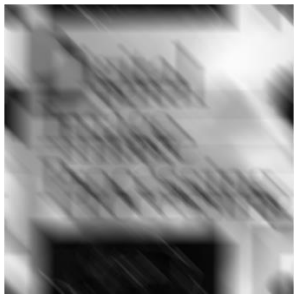
```python
def gaussian_kernel(kernel_size=3):
    h = gaussian(kernel_size, kernel_size / 3).reshape(kernel_size, 1)
    h = np.dot(h, h.transpose())
    h /= np.sum(h)
    return h

if __name__ == '__main__':
    # Load image and convert it to gray scale
    image = cv2.imread('Fig5.25.jpg', cv2.IMREAD_GRAYSCALE)

    # Apply Wiener Filter
    kernel = gaussian_kernel(3)
    filtered_img = wiener_filter(image, kernel, K=10)

    # Display results
    display = [image, filtered_img]
    label = ['Original Image', 'Wiener Filter applied']

    fig = plt.figure(figsize=(12, 10))

    for i in range(len(display)):
        fig.add_subplot(2, 2, i+1)
        plt.imshow(display[i], cmap='gray')
        plt.title(label[i])

    plt.show()
```

Wiener Filtering Result:



2. **Please use Inverse Filter and Wiener Filter to correct the image corrupted by motion blur of the assigned image, 'book-cover blurred.tif', and print out the source code and the corrected image? (40)**

Original Image:



```
import os
import numpy as np
from numpy.fft import fft2, ifft2
from scipy.signal import gaussian, convolve2d
import matplotlib.pyplot as plt
import cv2
import math

def motion_process(image_size, motion_angle, degree=15):
    center_position=(image_size[0]-1)/2
    slope_tan=math.tan(motion_angle*math.pi/180)
    slope_cot=1/slope_tan
    if slope_tan<=1:
        for i in range(degree):
            offset=round(i*slope_tan)
            PSF[int(center_position+offset),int(center_position-offset)]=1
        return PSF / PSF.sum()
    else:
        for i in range(degree):
            offset=round(i*slope_cot)
            PSF[int(center_position-offset),int(center_position+offset)]=1
        return PSF / PSF.sum()

def get_motion_dsf(image_size, motion_angle, motion_dis):

    PSF = np.zeros(image_size)
    x_center = (image_size[0] - 1) / 2
    y_center = (image_size[1] - 1) / 2

    sin_val = np.sin(motion_angle * np.pi / 180)
    cos_val = np.cos(motion_angle * np.pi / 180)
```

(variable) center_position: Any

```
    for i in range(motion_dis):
        x_offset = round(sin_val * i)
        y_offset = round(cos_val * i)
        PSF[int(x_center - x_offset), int(y_center + y_offset)] = 1

    # normalized
    return PSF / PSF.sum()

def make_blurred(input, PSF, eps):
    input_fft = np.fft.fft2(input)
    PSF_fft = np.fft.fft2(PSF)+ eps
    blurred = np.fft.ifft2(input_fft * PSF_fft)
    blurred = np.abs(np.fft.fftshift(blurred))
    return blurred

def inverse_filter(input, PSF, eps):

    input_fft = np.fft.fft2(input)
    PSF_fft = np.fft.fft2(PSF) + eps
    result = np.fft.ifft2(input_fft / PSF_fft)
    result = np.abs(np.fft.fftshift(result))
    return result
```

```python
def wiener_filter(input, PSF, eps, K=0.01):

    input_fft = np.fft.fft2(input)
    PSF_fft = np.fft.fft2(PSF) + eps
    PSF_fft_1 = np.conj(PSF_fft) / (np.abs(PSF_fft)**2 + K)
    result = np.fft.ifft2(input_fft * PSF_fft_1)
    result = np.abs(np.fft.fftshift(result))
    return result

if __name__ == "__main__":

    image = cv2.imread('book-cover-blurred.tif', 0)

    PSF = get_motion_dsf(image.shape[:2], -50, 100)
    blurred = make_blurred(image, PSF, 1e-3)

    plt.figure(figsize=(18, 8))
    plt.subplot(131), plt.imshow(blurred, 'gray'), plt.title("Original Image")
    plt.xticks([]), plt.yticks([])

    # Inverse Filtering
    result = inverse_filter(blurred, PSF, 1e-3)
    plt.subplot(132), plt.imshow(result, 'gray'), plt.title("inverse deblurred")
    plt.xticks([]), plt.yticks([])

    # Wiener FIltering
    result = wiener_filter(blurred, PSF, 1e-3, 0.000001)
    plt.subplot(133), plt.imshow(result, 'gray'), plt.title("wiener deblurred(k=0.000001)")
    plt.xticks([]), plt.yticks([])
```

```python
plt.tight_layout()
plt.show()
```

Inverse Filter Result:

Wiener Filter Result:

3. Please comment and compare your two design filters? (20)

Severe Turbulence Correction

The design frequency filters for severe turbulence correction involve the use of both Inverse Filter and Wiener Filter, each tailored to address distinct aspects of turbulence-induced distortions.

For the Inverse Filter, implemented in MATLAB, the frequency filter is designed to counteract the blurring effects caused by severe turbulence.

The choice of MATLAB is driven by its efficient handling of mathematical operations

and image processing tasks.

The frequency filter, formulated in the Fourier domain, seeks to invert the convolution process induced by turbulence.

However, this approach can be sensitive to noise, potentially leading to artifacts in the restored image.

The regularization term in the denominator is introduced to mitigate this sensitivity, balancing the restoration process.

Motion Blur Correction

The motion blur correction strategy involves the design of frequency filters using Inverse Filter and Wiener Filter techniques, implemented in Python.

The Inverse Filter for motion blur correction is designed in the frequency domain. It aims to reverse the convolution process caused by motion blur.

However, the Inverse Filter is known to be sensitive to noise, which necessitates careful consideration of noise levels during its application.

In contrast, the Wiener Filter is introduced to address the limitations of the Inverse Filter.

The Wiener Filter, also designed in the frequency domain, takes into account noise variance and image variance. By incorporating a regularization term (K), the Wiener Filter strikes a balance between noise suppression and detail preservation, making it particularly effective in scenarios with significant noise levels.

Comparison of Design Frequency Filters

Both Inverse Filters, applied in scenarios of severe turbulence and motion blur, aim to counteract the effects of convolution. However, their implementations vary to address the specific types of distortions present. The susceptibility of Inverse Filters to noise is evident in both cases, highlighting the need for regularization terms to stabilize their performance.

On the other hand, Wiener Filters are designed with a focus on minimizing noise sensitivity, demonstrating their flexibility across a wide range of image restoration tasks. By incorporating regularization, Wiener Filters can be fine-tuned according to

the characteristics of the noise, showcasing their adaptability and reliability.

In conclusion, the frequency domain filters integrate the strengths of Inverse Filters and Wiener Filters, selecting each based on its effectiveness for handling challenges like severe turbulence and motion blur. This comparison emphasizes the careful trade-offs made in image restoration, balancing noise suppression with the preservation of image details.