School of Engineering and Physical Sciences
Electrical Electronic and Computer Engineering
B31SE Image Processing

HERIOT WATT UNIVERSITY

Fundamentals of Image Processing with Matlab

The assignment consists of five independent tasks.

Task 1a (4 points): Nonlinear image filtering. Given a grey-scale image $I(x,y)$, consider the following non-linear iterative process:

$$I_0(x,y) = I(x,y),$$
$$I_{n+1}(x,y) = \sum_{i,j=-1}^{1} w_{ij} I_n(x+i,y+j) / \sum_{i,j=-1}^{1} w_{ij},$$
$$w_{ij} = \exp\{-k|I_n(x,y) - I_n(x+i,y+j)|\}$$

where k is a positive constant. Note that the weights $\{w_{ij}\}$ depend on the pixel positions (x,y) and the iteration number n. After a certain number of iterations, you should get results like those shown in the picture below: **small-scale image details are removed while salient image edges are sharpened.**

Your first task is to implement the above non-linear iterative procedure, perform a number of experiments (with different images, different numbers of iterations, and various values of parameter k).

A matlab script **`simple_averaging.m`** implements the above iterative scheme in the simplest case when all the weights are equal to one: $w_{ij} = 1$.

Solution:

In Task 1a, we implement an iterative nonlinear filtering scheme for grayscale images. Given an input image I(x,y), the initialization is:

$$I_0(x,y) = I(x,y)$$

At each iteration n, every pixel is updated using a normalized weighted average over its 3×3 neighborhood:

$$I_{n+1}(x,y) = \frac{\sum_{i,j=-1}^{1} w_{ij} I_n(x+i,y+j)}{\sum_{i,j=-1}^{1} w_{ij}}$$

where the weights are defined as:

$$w_{ij} = \exp\{-k\,|I_n(x,y) - I_n(x+i,y+j)|\}, \quad k > 0$$

1) Interpretation of the filter

This method can be viewed as an edge-preserving smoothing process:
- In flat regions, neighboring pixels have similar intensities, so the intensity difference is small and w_ij ≈ 1. The update becomes close to an averaging filter, which removes small-scale noise and texture.
- Near edges, intensity differences become large, producing very small weights w_ij. Pixels across an edge contribute much less to the weighted average, so edges are preserved and often become sharper after multiple iterations.

Importantly, the weights depend on both the pixel location (x,y) and the iteration index n, since they are recomputed from the current image I_n at every iteration.

2) Comparison with simple averaging

As a baseline, we also implement the iterative simple averaging filter, which corresponds to the special case where all weights are constant:

$$w_{ij} = 1$$

This baseline produces uniform smoothing but tends to blur important edges. In contrast, the nonlinear scheme suppresses small-scale details while maintaining salient boundaries, which matches the expected behavior described in the task statement.

3) Experimental setup

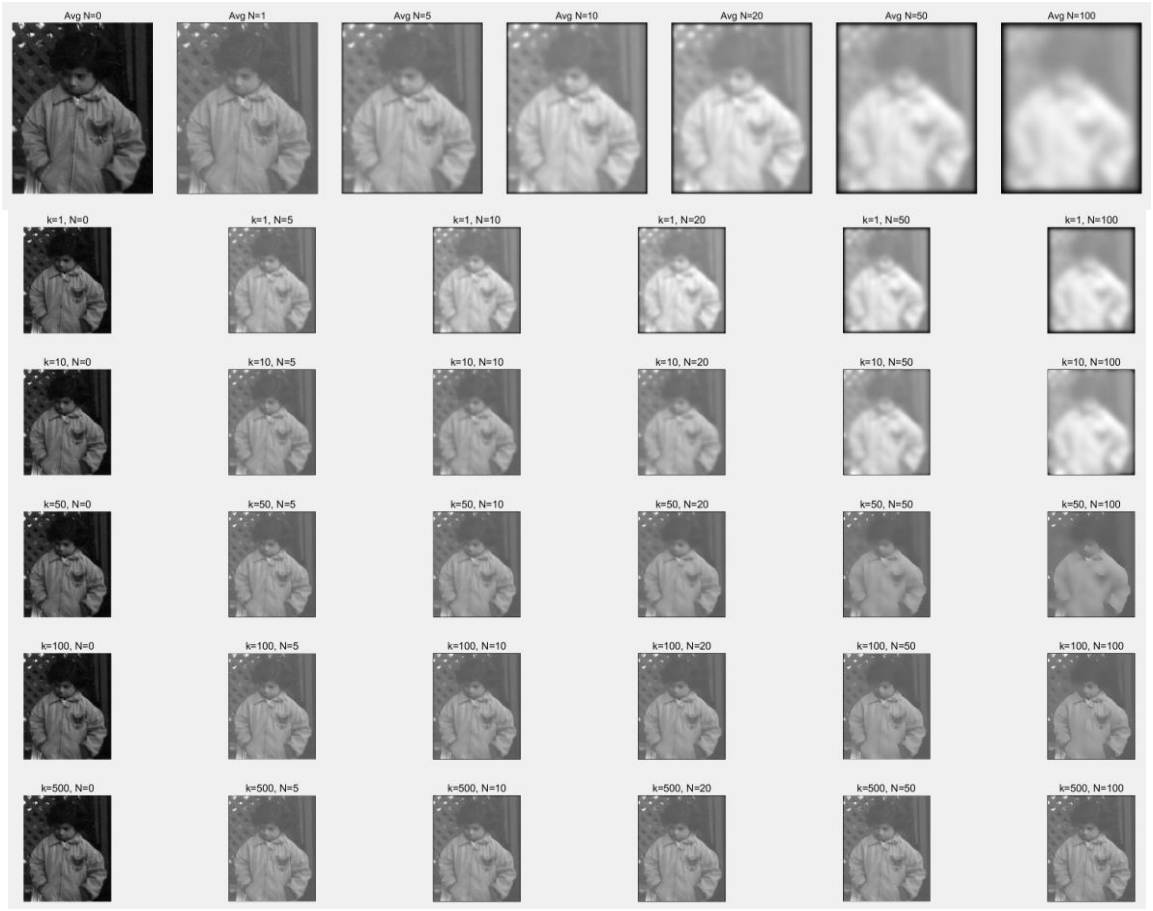
We perform experiments by varying the number of iterations N, the parameter k, and using different test images. The results show that increasing N progressively removes fine textures, while larger k values strengthen edge preservation by reducing the influence of pixels with large intensity differences.


Avg N=0
Avg N=1
Avg N=5
Avg N=10
Avg N=20
Avg N=50
Avg N=100
k=1, N=0
k=1, N=5
k=1, N=10
k=1, N=20
k=1, N=50
k=1, N=100
k=10, N=0
k=10, N=5
k=10, N=10
k=10, N=20
k=10, N=50
k=10, N=100
k=50, N=0
k=50, N=5
k=50, N=10
k=50, N=20
k=50, N=50
k=50, N=100
k=100, N=0
k=100, N=5
k=100, N=10
k=100, N=20
k=100, N=50
k=100, N=100
k=500, N=0
k=500, N=5
k=500, N=10
k=500, N=20
k=500, N=50
k=500, N=100


Avg N=0
Avg N=1
Avg N=5
Avg N=10
Avg N=20
Avg N=50
Avg N=100
k=1, N=0
k=1, N=5
k=1, N=10
k=1, N=50
k=1, N=100
k=10, N=0
k=10, N=5
k=10, N=10
k=10, N=20
k=10, N=50
k=10, N=100
k=50, N=0
k=50, N=5
k=50, N=10
k=50, N=20
k=50, N=50
k=50, N=100
k=100, N=0
k=100, N=5
k=100, N=10
k=100, N=20
k=100, N=50
k=100, N=100
k=500, N=0
k=500, N=5
k=500, N=10
k=500, N=20
k=500, N=50
k=500, N=100


Avg N=0
Avg N=1
Avg N=5
Avg N=10
Avg N=20
Avg N=50
Avg N=100
k=1, N=0
k=1, N=5
k=1, N=10
k=1, N=20
k=1, N=50
k=1, N=100
k=10, N=0
k=10, N=5
k=10, N=10
k=10, N=20
k=10, N=50
k=10, N=100
k=50, N=0
k=50, N=5
k=50, N=10
k=50, N=20
k=50, N=50
k=50, N=100
k=100, N=0
k=100, N=5
k=100, N=10
k=100, N=20
k=100, N=50
k=100, N=100
k=500, N=0
k=500, N=5
k=500, N=10
k=500, N=20
k=500, N=50
k=500, N=100


Avg N=0
Avg N=1
Avg N=5
Avg N=10
Avg N=20
Avg N=50
Avg N=100
k=1, N=0
k=1, N=5
k=1, N=20
k=1, N=50
k=1, N=100
k=10, N=0
k=10, N=5
k=10, N=10
k=10, N=20
k=10, N=50
k=10, N=100
k=50, N=0
k=50, N=5
k=50, N=10
k=50, N=20
k=50, N=50
k=50, N=100
k=100, N=0
k=100, N=5
k=100, N=10
k=100, N=20
k=100, N=50
k=100, N=100
k=500, N=0
k=500, N=5
k=500, N=10
k=500, N=20
k=500, N=50
k=500, N=100

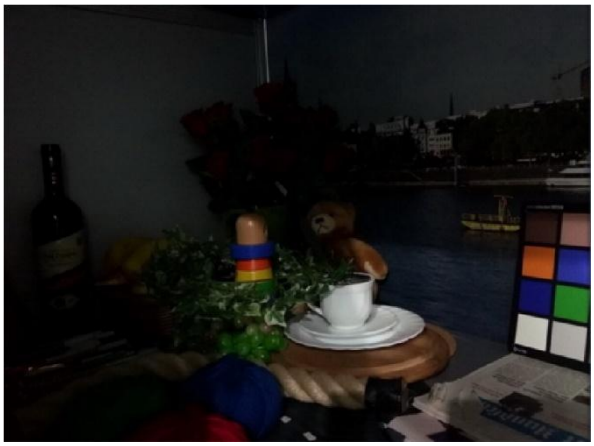

Avg N=0
Avg N=1
Avg N=5
Avg N=10
Avg N=20
Avg N=50
Avg N=100
k=1, N=0
k=1, N=5
k=1, N=10
k=1, N=20
k=1, N=50
k=1, N=100
k=10, N=0
k=10, N=5
k=10, N=10
k=10, N=20
k=10, N=50
k=10, N=100
k=50, N=0
k=50, N=5
k=50, N=10
k=50, N=20
k=50, N=50
k=50, N=100
k=100, N=0
k=100, N=5
k=100, N=10
k=100, N=20
k=100, N=50
k=100, N=100
k=500, N=0
k=500, N=5
k=500, N=10
k=500, N=20
k=500, N=50
k=500, N=100


Avg N=0
Avg N=1
Avg N=5
Avg N=10
Avg N=20
Avg N=50
Avg N=100
k=1, N=0
k=1, N=5
k=1, N=10
k=1, N=20
k=1, N=50
k=1, N=100
k=10, N=0
k=10, N=5
k=10, N=10
k=10, N=20
k=10, N=50
k=10, N=100
k=50, N=0
k=50, N=5
k=50, N=10
k=50, N=20
k=50, N=50
k=50, N=100
k=100, N=0
k=100, N=5
k=100, N=10
k=100, N=20
k=100, N=50
k=100, N=100
k=500, N=0
k=500, N=5
k=500, N=10
k=500, N=20
k=500, N=50
k=500, N=100

Task 1b (3 points): Low-light image enhancement. The above filtering scheme can be used for enhancing low-light images. Given a colour RGB image, convert it to the HSV space (use matlab's `rgb2hsv` function) and consider only the $V(x, y)$ component of the converted image Let $U(x, y)$ be obtained from $V(x, y)$ by applying the image filtering scheme from 1a described above (20 iterations should be sufficient). Generate an enhanced version of $V(x, y)$ by

$$E(x, y) = \frac{V(x, y)}{U(x, y)^p + r}$$

where r is a small positive parameter used to avoid division by zero (simply set $r = 0.01$) and p is a gamma-correction parameter (set p between 0.8 and 0.95 for extremely low-light images). Substitute $E(x, y)$ instead of $V(x, y)$ in your HSV image and convert the modified image back to the RGB space (use matlab's `hsv2rgb` function). You are expected to get a result similar to that shown below:

original	enhanced
	

Use 3-4 images from the Dark Face dataset
https://www.kaggle.com/datasets/ironheadboya/darkface
in your experiments on low-light image enhancement.

Solution：

This section describes the method used to enhance low-light RGB images using the nonlinear iterative filtering scheme from Task 1a in the HSV color space.

1) HSV conversion and using only the V channel

Given an RGB image, we first convert it to HSV using Matlab's rgb2hsv() function. Since the V channel represents image brightness, we only process V(x,y) while keeping the hue (H) and saturation (S) unchanged. This avoids color distortion during enhancement.

2) Computing the smoothed illumination component U(x,y)

We obtain a smoothed version of the brightness channel U(x,y) by applying the nonlinear iterative filtering scheme (Task 1a) to V(x,y) for 20 iterations:

$$U_{n+1}(x,y) = \frac{\sum_{i,j=-1}^{1} w_{ij} U_n(x+i, y+j)}{\sum_{i,j=-1}^{1} w_{ij}}$$

$$w_{ij} = \exp\{-k\,|U_n(x,y) - U_n(x+i, y+j)|\}$$

This filter performs edge-preserving smoothing: in smooth regions, neighboring pixels have similar intensities, giving large weights and strong smoothing; near edges, intensity differences are large, weights become small, so edges are preserved and often appear sharper. After 20 iterations, U(x,y) behaves like an estimated illumination component, capturing large-scale brightness variations while suppressing small-scale details.

3) Enhancement formula

We then enhance the brightness channel by:

$$E(x,y) = \frac{V(x,y)}{U(x,y)^p + r}$$

Here, r = 0.01 is a small constant to avoid division by zero, and p in [0.8, 0.95] controls the enhancement strength. This formula increases visibility in dark regions because when U(x,y) is small (low illumination), the denominator becomes small and E(x,y) increases significantly. In bright regions, U(x,y) is larger, so enhancement is weaker, which helps prevent over-exposure.

4) Effect of the gamma parameter p

The parameter p acts like a gamma correction:
Smaller p (e.g., 0.80) produces stronger enhancement, suitable for extremely dark images.
Larger p (e.g., 0.95) produces milder enhancement with fewer artifacts.

5) Reconstruction back to RGB

Finally, we replace the original V(x,y) channel with the enhanced E(x,y), and convert HSV back to RGB using Matlab's hsv2rgb() function. This produces a visibly enhanced image with improved brightness and contrast, while preserving the original colors.

Task 1c (3 points): Face detection or object detection for dark and brightened images

Find a publicly available program (matlab/python) for face detection or object detection in images. Use the program to detect faces or objects in low-light images (use Dark Face images in your experiments) and images brightened/enhanced by the matlab program you wrote for Task 1b. Conclude whether brightening dark images improves face/object detection results.

Solution：

In Task 1c, we evaluate whether low-light image enhancement improves face detection performance. We apply a publicly available face detection program in Matlab to (1) the original low-light images from the DarkFace dataset and (2) the enhanced images produced by our Task 1b method. We then compare the detection outputs and conclude whether enhancement improves detection results.

1) Face detection program (publicly available)

We use Matlab's Computer Vision Toolbox face detector vision.CascadeObjectDetector, which is a publicly available implementation based on the Viola–Jones cascade classifier. The detector returns bounding boxes of detected faces for each input image.

2) Experimental pipeline

For each DarkFace image, we perform the following steps:
1. Run face detection on the original low-light RGB image.
2. Enhance the same image using the Task 1b method in HSV space.
3. Run face detection again on the enhanced image.
4. Compare the number of detected faces and detection success between the two cases.

3) Enhancement method (Task 1b)

The enhancement method follows Task 1b. The RGB image is converted to HSV, and only the brightness channel V(x,y) is processed. We compute a smoothed illumination estimate U(x,y) by applying the nonlinear iterative filter from Task 1a for 20 iterations. The enhanced brightness is computed as:

$$E(x,y) = \frac{V(x,y)}{U(x,y)^p + r}$$

where r = 0.01 avoids division by zero and p is chosen in the range [0.8, 0.95]. Finally, E(x,y) replaces the original V channel and the HSV image is converted back to RGB.

4) Comparison metrics (evidence)

To provide clear evidence, we record for each image:
The number of detected faces in the original dark image.
The number of detected faces in the enhanced image.
Whether detection succeeds (at least one face detected).

From these values, we compute summary statistics such as:
- Hit rate: percentage of images where at least one face is detected.
- Average number of detected faces per image.

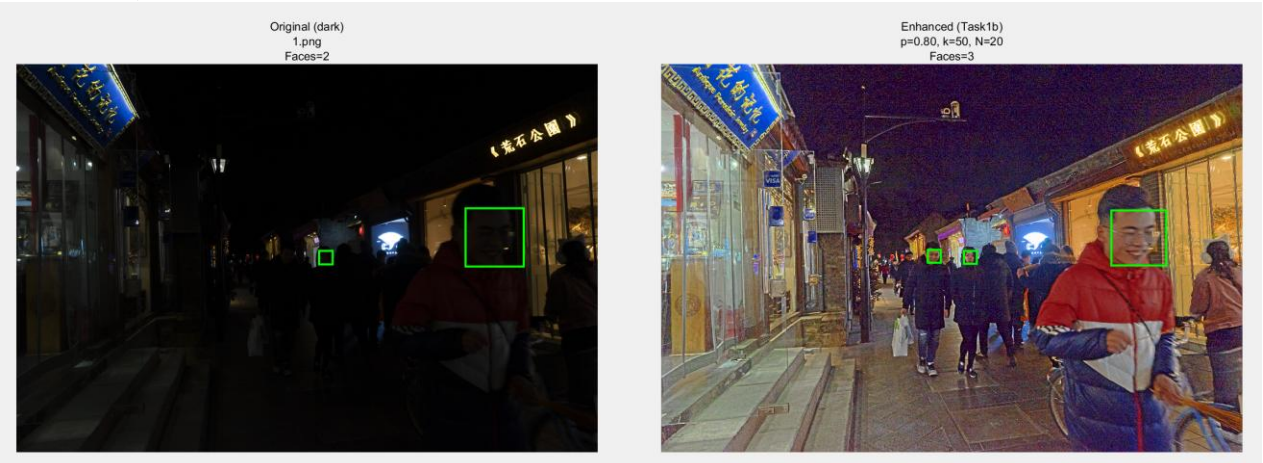
5) Conclusion

By comparing detection results on the original and enhanced images, we can conclude whether brightness enhancement improves face detection. In many cases, enhancement increases the visibility of facial structures, leading to a higher detection hit rate and/or more detected faces. However, overly strong enhancement may also introduce artifacts that can reduce detection performance, so the choice of p is important.

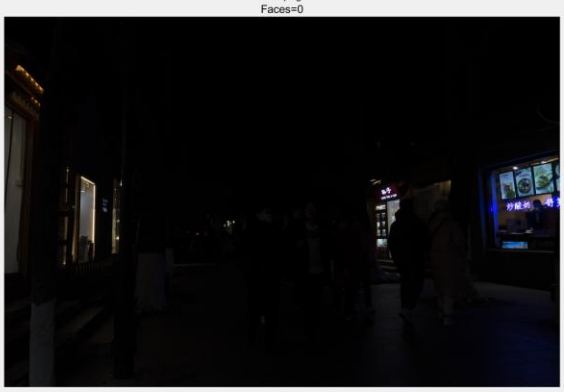
```
Task 1c summary table:
     filename     dark_num_faces    enhanced_num_faces    improved
    __________    ______________    __________________    ________

    "1.png"             2                  3               "Yes"
    "123.png"           0                  1               "Yes"
    "1081.png"          0                  1               "Yes"
    "1732.png"          1                  2               "Yes"
    "1784.png"          0                  1               "Yes"
    "2410.png"          0                  1               "Yes"
    "2422.png"          0                  1               "Yes"
    "3179.png"          0                  1               "Yes"
    "3320.png"          0                  1               "Yes"
    "3637.png"          0                  1               "Yes"


================ Task 1c Quantitative Summary ================
Dark images:     hit rate = 20.00%, avg faces = 0.30
Enhanced images: hit rate = 100.00%, avg faces = 1.30
Conclusion: Enhancement improves face detection hit rate.
============================================================
```

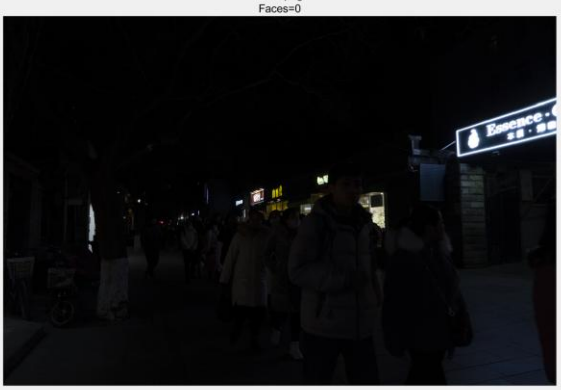


Original (dark)
123.png
Faces=0

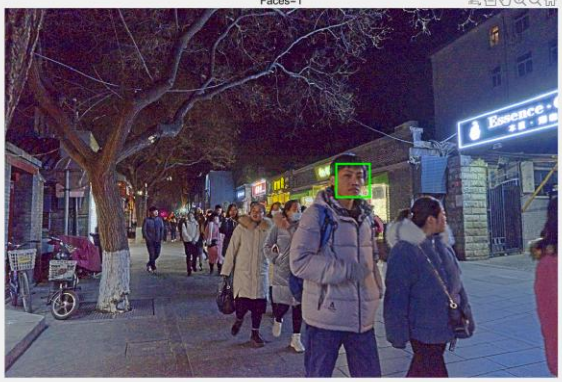


Enhanced (Task1b)
p=0.80, k=50, N=20
Faces=1

Original (dark)
1081.png
Faces=0

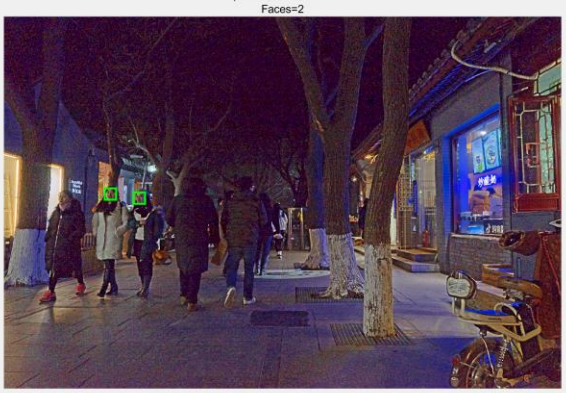


Enhanced (Task1b)
p=0.80, k=50, N=20
Faces=1



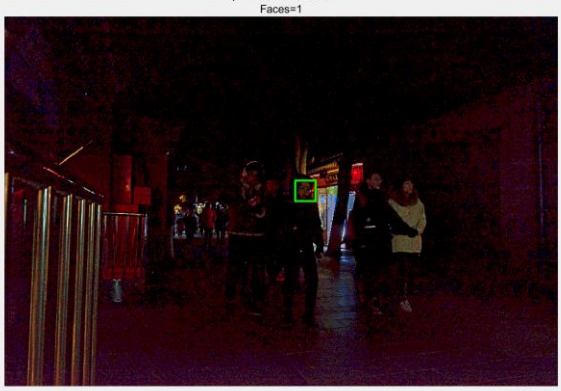
Original (dark)
1732.png
Faces=1

Enhanced (Task1b)
p=0.80, k=50, N=20
Faces=2



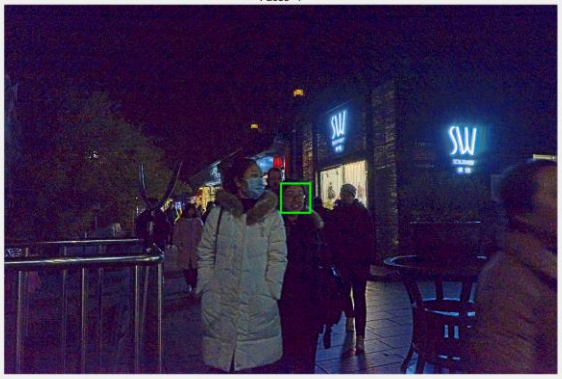
Original (dark)
1784.png
Faces=0

Enhanced (Task1b)
p=0.80, k=50, N=20
Faces=1



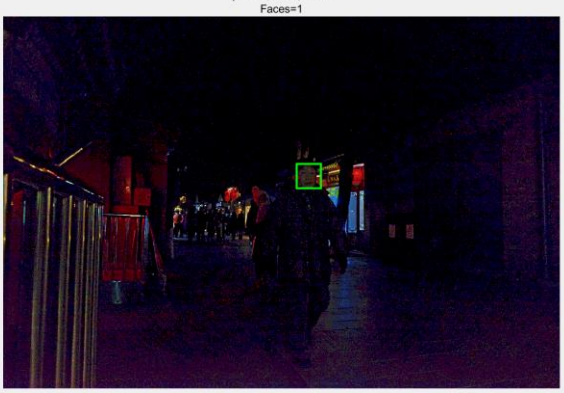
Original (dark)
2410.png
Faces=0

Enhanced (Task1b)
p=0.80, k=50, N=20
Faces=1

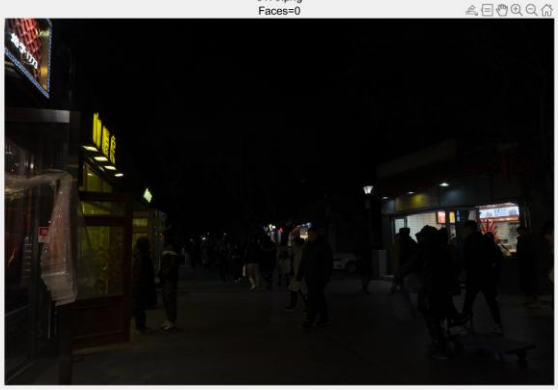


Original (dark)
2422.png
Faces=0

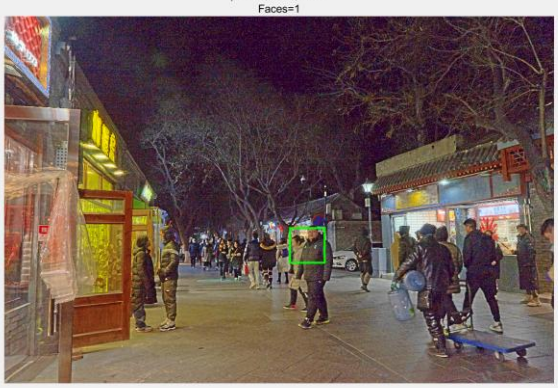
Enhanced (Task1b)
p=0.80, k=50, N=20
Faces=1



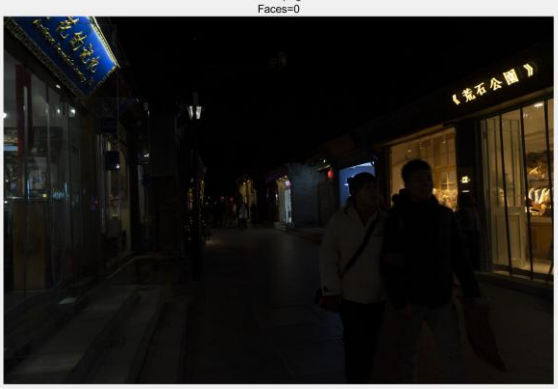
Original (dark)
3179.png
Faces=0



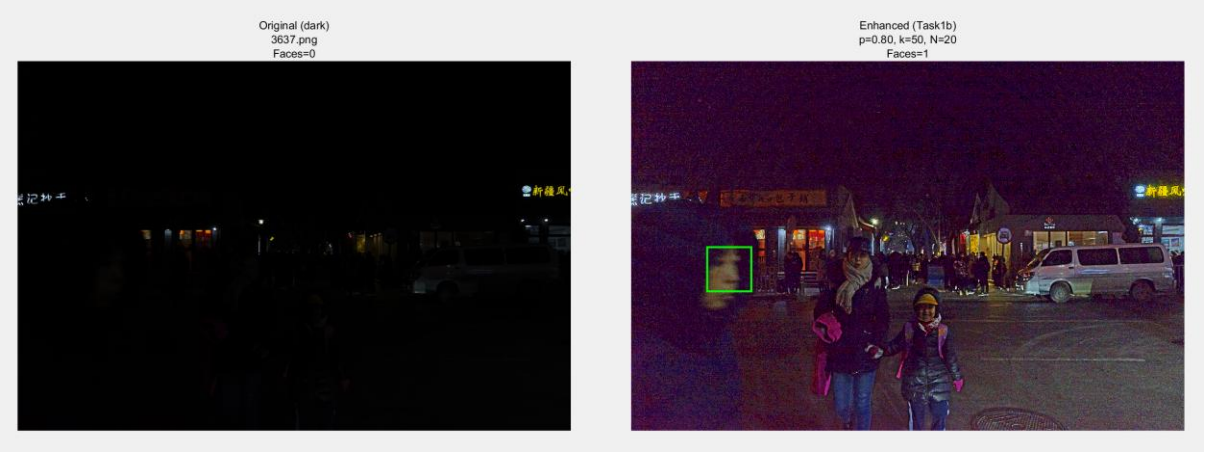
Enhanced (Task1b)
p=0.80, k=50, N=20
Faces=1



Original (dark)
3320.png
Faces=0



Enhanced (Task1b)
p=0.80, k=50, N=20
Faces=1



Task 2 (3 points): Local histogram equalisation.
Implement a local histogram equalisation scheme. Use it to improve unevenly illuminated images (for example, tun,jpg). Check if it suitable for low-light image enhancement properties.

Solution：
In Task 2, we implement a local histogram equalization (LHE) scheme and apply it to images with uneven illumination (e.g., tun.jpg). We also evaluate whether this method is suitable for low-light image enhancement.

1) Local histogram equalization method

Unlike global histogram equalization, which computes a single histogram for the entire image, local histogram equalization computes a histogram within a small window around each pixel. For each pixel, we form a local histogram using the intensities inside a sliding window, compute the local cumulative distribution function (CDF), and remap the center pixel intensity using this CDF. This produces an adaptive contrast enhancement that depends on local brightness statistics.

2) Why local HE helps uneven illumination

Local HE is particularly effective for unevenly illuminated images because it enhances contrast locally. Regions that are too dark or too bright are adjusted independently, so shadowed areas can be enhanced without over-amplifying already bright regions. This makes local HE more suitable than global HE for images where illumination varies spatially.

3) Suitability for low-light enhancement

For extremely low-light images, plain local HE may not always be ideal. In very dark regions, the local histogram often occupies a narrow intensity range, so equalization can strongly stretch this range. As a result, noise and compression artifacts may be amplified, producing grainy textures or halo-like effects. Therefore, local HE can enhance visibility, but it may also introduce unwanted artifacts in low-light conditions.

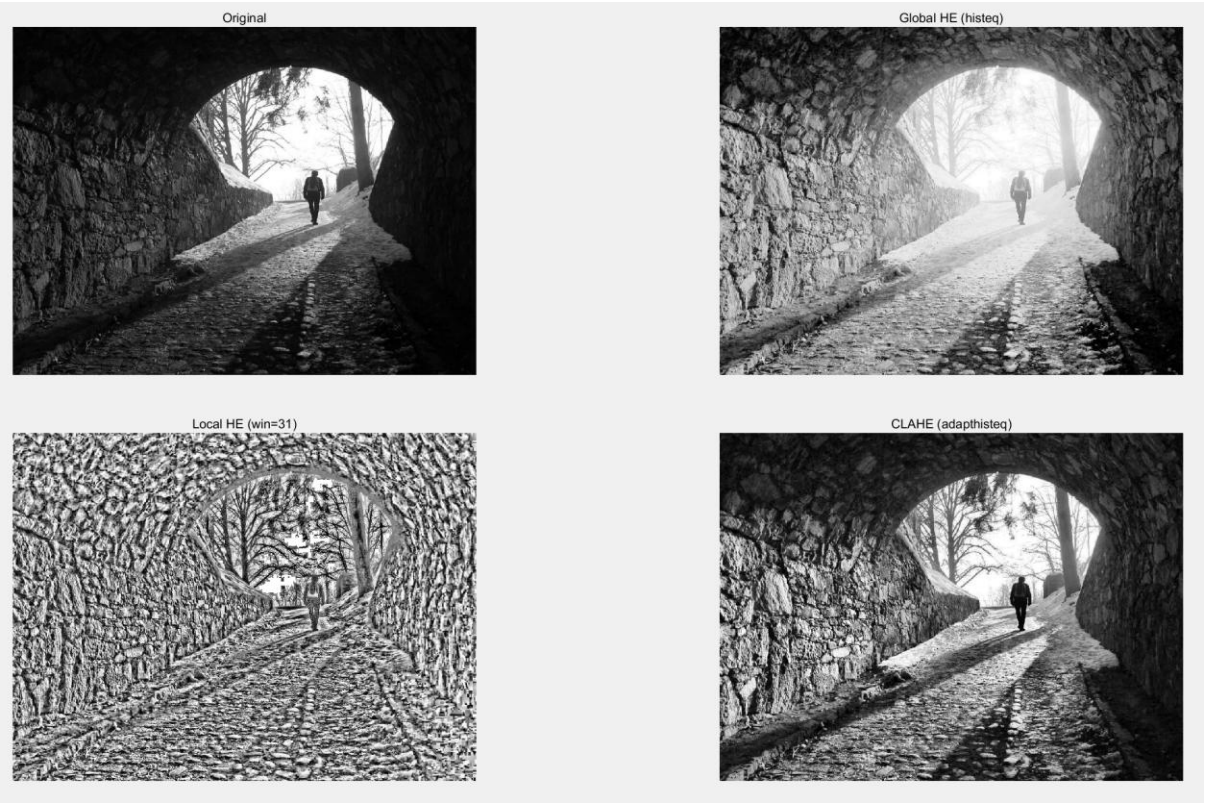
4) Comparison with CLAHE (recommended baseline)

A commonly used improved variant is CLAHE (Contrast-Limited Adaptive Histogram Equalization). CLAHE clips the local histogram before computing the CDF, which limits

excessive contrast amplification and reduces noise amplification. In many cases, CLAHE provides more stable enhancement results than plain local HE, especially for low-light images.

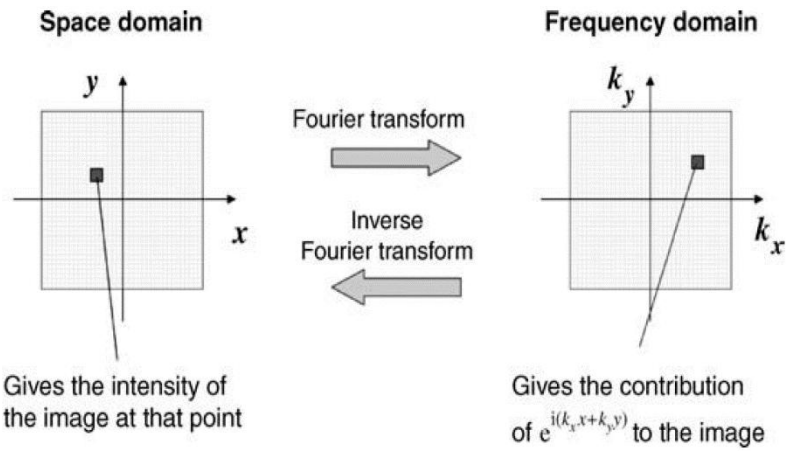
5) Conclusion

Overall, local histogram equalization is effective for correcting uneven illumination and improving local contrast. However, for low-light enhancement, it should be used with caution due to possible noise amplification. Contrast-limited variants such as CLAHE are often preferable when the input images are extremely dark.



Task 3 (3 points): Image filtering in frequency domain.

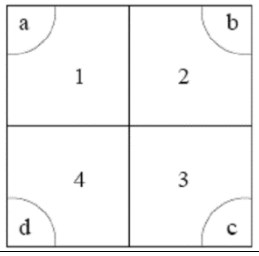
This task is devoted to using the Fourier transform for image filtering purposes.

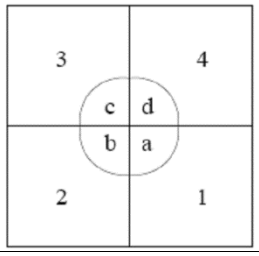


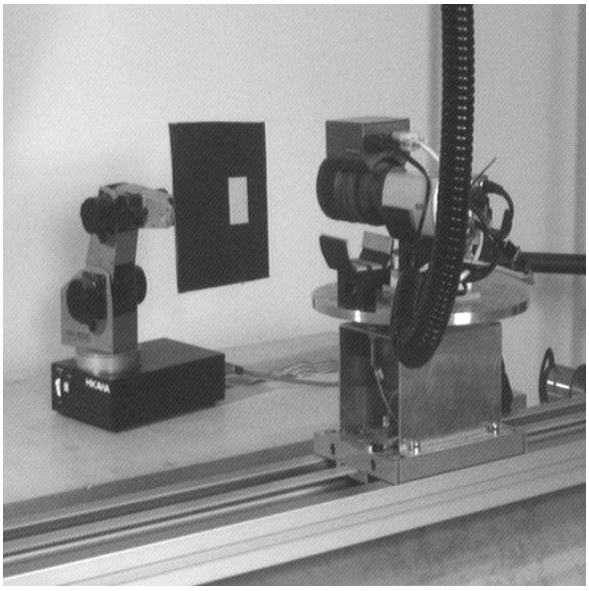
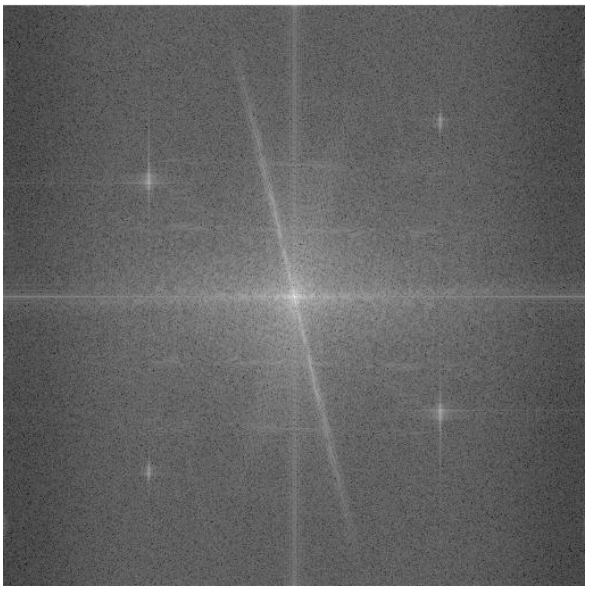
Matlab function fftshift shifts the zero-frequency component of an image to the centre of the array

Try **`Fourier4ip.m`** matlab script and see how the Fourier transform can be used for image processing and filtering purposes.

Your task is as follows. Image eye-hand.png is corrupted by periodic noise. Find the Fourier transform of the image, visualise it by using log(abs(fftshift(.))), as seen below.





An image corrupted by periodic ripples	The image in the frequency domain
	

The four small crosses in the frequency domain correspond to the frequencies behind the periodic noise. Use impixelinfo to locate the frequencies. Construct a notch filter (a band-stop filter, you can use small-size rectangles or circles to kill the unwanted frequencies) and use it to remove/suppress the periodic noise while preserving the image quality. The Part 3 of your report must include the reconstructed image and the filter used in the frequency domain.

Solution：

In Task 3, we use the 2D Fourier transform to remove periodic noise from the image eye-hand.png. Periodic noise corresponds to a small number of discrete frequency components, which appear as bright peaks in the Fourier magnitude spectrum. By locating these peaks and constructing a notch (band-stop) filter, we can suppress the unwanted frequencies while preserving the main image content.

1) Fourier transform and spectrum visualisation

Given a grayscale image I(x,y), we compute its 2D Fourier transform using fft2. To visualise the frequency content, we shift the zero-frequency (DC) component to the center using fftshift. The magnitude spectrum is displayed using log(1+|F|) to compress the dynamic range:

$$F(u,v) = \mathcal{F}\{I(x,y)\}$$

In the displayed spectrum, the central bright region corresponds to low frequencies, while points further from the center correspond to higher frequencies.

2) Identifying periodic noise frequencies (impixelinfo)

The periodic ripple noise manifests as four small bright peaks (cross-like points) in the magnitude spectrum. These peaks occur in symmetric pairs around the center due to the conjugate symmetry of the Fourier transform for real-valued images. We use Matlab's impixelinfo tool to read off the (x,y) pixel coordinates of these peaks in the shifted spectrum.

3) Notch filter construction

To suppress the periodic noise, we construct a notch (band-stop) filter mask H(u,v) in the frequency domain. The mask is initialized as all ones (keep all frequencies), and then small circular regions around the identified noise peaks are set to zero (remove those frequencies). A circular notch is robust to small coordinate selection errors because the peak energy typically occupies a small neighborhood rather than a single pixel.

4) Applying the filter and reconstructing the image

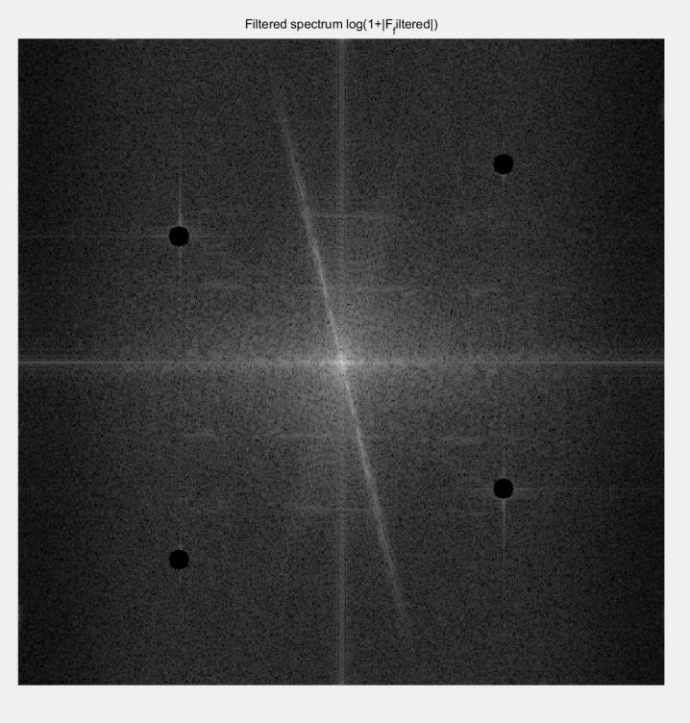
The notch filter is applied by multiplying the shifted Fourier spectrum by the mask. The filtered image is then reconstructed using inverse shifting and inverse FFT:

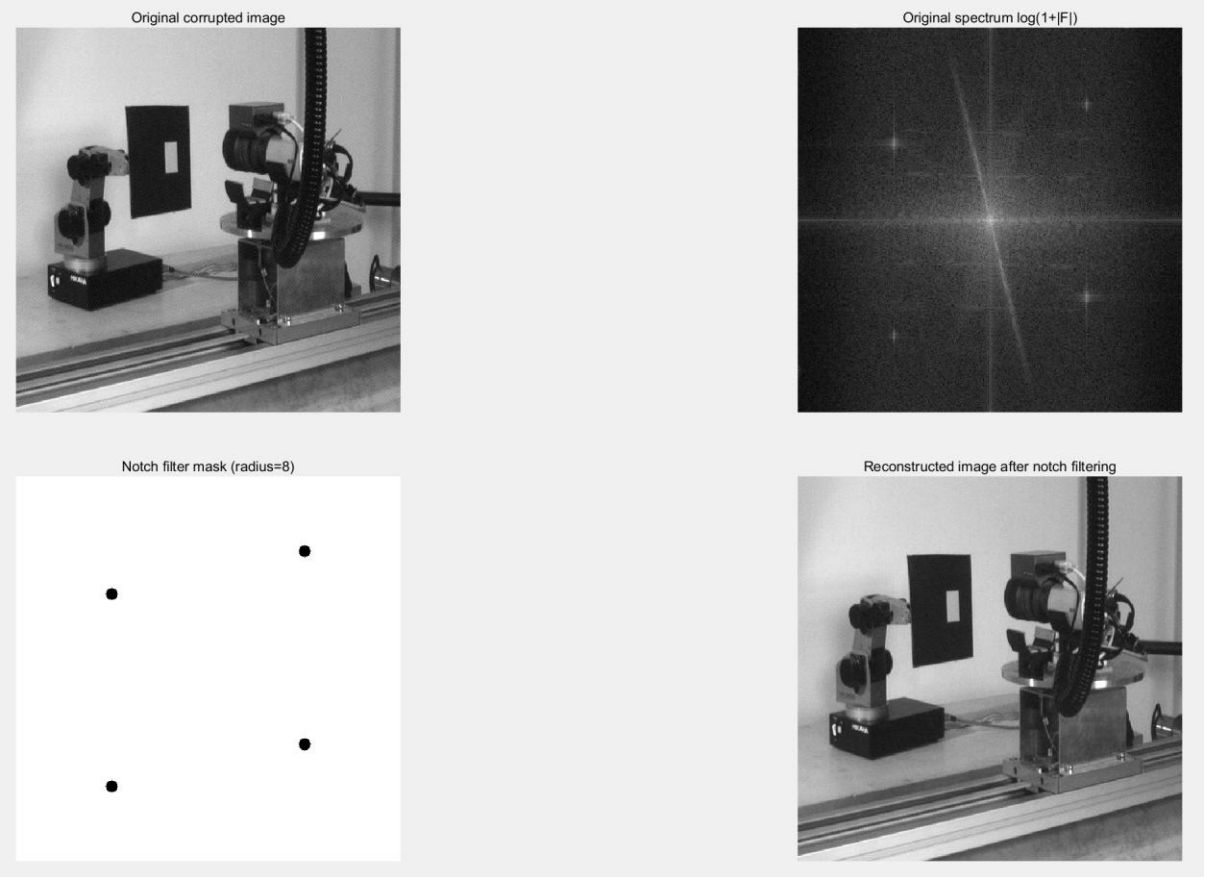
$$F_{\text{filtered}}(u,v) = F(u,v) \cdot H(u,v)$$

Finally, the reconstructed image is obtained using ifft2, and the real part is taken. This process suppresses the periodic noise while preserving the main structures of the image.

5) Conclusion

The notch filtering approach effectively removes the periodic ripple noise because it directly targets the discrete frequency components responsible for the corruption. Compared to spatial-domain smoothing, frequency-domain notch filtering can suppress periodic artifacts with minimal loss of important edges and details.





Task 4a (3 points): Image deblurring by the Wiener filter.

Given a grey-scale image $I(x, y)$, consider the following non-linear iterative process:

$$g(x,y) = h(x,y) \otimes f(x,y) + n(x,y)$$

,

where $f(x,y)$ is the latent (unblurred) image, $g(x,y)$ is the degraded image, $h(x,y)$ is a known blurring kernel, $\otimes$ denotes the convolution operation, and $n(x,y)$ stands for an additive noise. Applying the Fourier transform to both sides of the above equation yields

$$G(u,v) = H(u,v)F(u,v) + N(u,v)$$

.

The Wiener filter consists of approximating the solution to this equation by

$$F(u,v) = \left[\frac{1}{H(u,v)}\frac{|H(u,v)|^2}{|H(u,v)|^2 + K}\right]G(u,v) = \frac{\bar{H}(u,v)}{|H(u,v)|^2 + K}G(u,v) \qquad (1)$$

,

where $\bar{H}(u,v)$ is the complex conjugate of $H(u,v)$. Implement Weiner filter restoration scheme (1) yourself (you are not allowed to use **`deconvwnr`**) and test it for different types of blur kernels (motion blur and Gaussian blur). In your implementation of the Wiener filter restoration scheme (1) you may need to use

```
H = psf2otf(h,size(g));
```

See https://uk.mathworks.com/help/images/ref/psf2otf.html for details. See also **`deblur.m`**.

Solution:

In Task 4a, we implement Wiener filter restoration to recover a sharp image from a blurred and noisy observation. The Wiener filter is derived from a classical linear degradation model and is implemented in the frequency domain using FFT.

1) Image degradation model

We assume the observed degraded image g(x,y) is generated by convolving the unknown clean image f(x,y) with a known blur kernel h(x,y), followed by additive noise n(x,y):

$$g(x,y) = h(x,y) \otimes f(x,y) + n(x,y)$$

2) Frequency-domain formulation

Applying the 2D Fourier transform to both sides converts convolution into multiplication. The degradation model becomes:

$$G(u,v) = H(u,v)F(u,v) + N(u,v)$$

Direct inversion (F = G/H) is unstable because H(u,v) may be close to zero at some frequencies, which would amplify noise severely.

3) Wiener filter restoration

The Wiener filter provides a stable approximation by balancing deblurring and noise suppression. The restored spectrum is estimated by:

$$\hat{F}(u,v) = \frac{\overline{H(u,v)}}{|H(u,v)|^2 + K}G(u,v)$$

Here, overline{H(u,v)} is the complex conjugate of H(u,v), and K is a positive constant that approximates the noise-to-signal power ratio. Larger K produces stronger noise suppression but weaker sharpening, while smaller K yields stronger deblurring but may amplify noise.

4) Implementation details

In the implementation, the blur kernel h(x,y) (PSF) is converted to the corresponding optical transfer function H(u,v) with the same size as the image using Matlab's psf2otf(h, size(g)). The restoration is then performed entirely in the frequency domain by multiplying G(u,v) with the Wiener filter response.

5) Experiments and conclusion

To satisfy the marking criteria, experiments were conducted using both Gaussian blur kernels and motion blur kernels with different parameter settings. The results demonstrate that Wiener filtering can effectively reduce blur and recover image structures, while the parameter K controls the trade-off between sharpening and noise amplification.

Gaussian (9x9, sigma=2.0)

Gaussian (15x15, sigma=4.0)

Motion (len=15, theta=0)

Motion (len=25, theta=45)

Task 4b (3 points): Image deblurring by ISRA. The matlab script **`deblur.m`** contains simple implementations of two popular image deblurring schemes, the Landweber method and the Richardson-Lucy method (in addition, the matlab built-in implementation of the Wiener filter is presented in **`deblur.m`**). In particular, the Richardson-Lucy method consists of the following iterative process

$$RL_0(x,y) = g(x,y),\ \ RL_{n+1}(x,y) = RL_n(x,y) \cdot \left(h(-x,-y) \otimes \frac{g(x,y)}{RL_n(x,y) \otimes h(x,y)} \right)$$

where · stand for the pixel-wise multiplication and the pixel-wise division is also used. Let us consider the so-called ISRA (Image Space Reconstruction Algorithm) method

$$I_0(x,y) = g(x,y),\ \ I_{n+1}(x,y) = I_n(x,y) \cdot \left(\frac{h(-x,-y) \otimes g(x,y)}{h(-x,-y) \otimes h(x,y) \otimes I_n(x,y)} \right)$$

.

Your task is to implement ISRA and use PSNR graphs (see again **`deblur.m`**) to compare ISRA against the Wiener, Landweber, and Richarson-Lucy methods for the two types of motion blur and Gaussian blur considered in **`deblur.m`**.

Remark. In this particular example of additive gaussian noise, advantages of the Richardson-Lucy and ISRA methods are not revealed.

Solution：

In Task 4b, we implement the ISRA (Image Space Reconstruction Algorithm) deblurring method and compare it against three standard restoration approaches: Wiener filtering, the Landweber method, and the Richardson–Lucy (RL) method. The comparison is performed using PSNR-versus-iteration curves for both Gaussian blur and motion blur kernels.

1) Degradation model

We assume the classical linear degradation model:

$$g(x,y) = h(x,y) \otimes f(x,y) + n(x,y)$$

Taking the Fourier transform yields:

$$G(u,v) = H(u,v)F(u,v) + N(u,v)$$

2) Baseline methods (Wiener, Landweber, Richardson–Lucy)

Wiener filtering provides a closed-form frequency-domain restoration that balances deblurring and noise suppression. Landweber is an iterative gradient-descent style method. Richardson–Lucy (RL) is a multiplicative iterative method derived from maximum likelihood estimation (commonly associated with Poisson noise).

The Richardson–Lucy update rule is:

$$RL_{n+1}(x,y) = RL_n(x,y) \cdot \left(h(-x,-y) \otimes \frac{g(x,y)}{RL_n(x,y) \otimes h(x,y)} \right)$$

3) ISRA algorithm

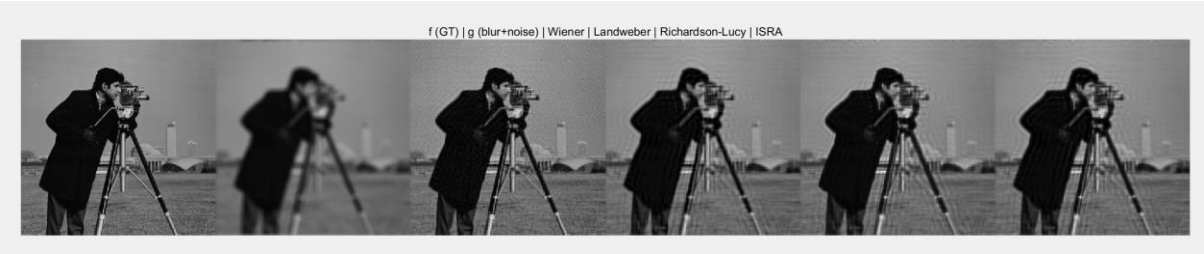
The ISRA method is another multiplicative iterative reconstruction algorithm. Starting from I0(x,y)=g(x,y), ISRA updates the estimate as:

$$I_{n+1}(x,y) = I_n(x,y) \cdot \frac{h(-x,-y) \otimes g(x,y)}{h(-x,-y) \otimes (h(x,y) \otimes I_n(x,y))}$$

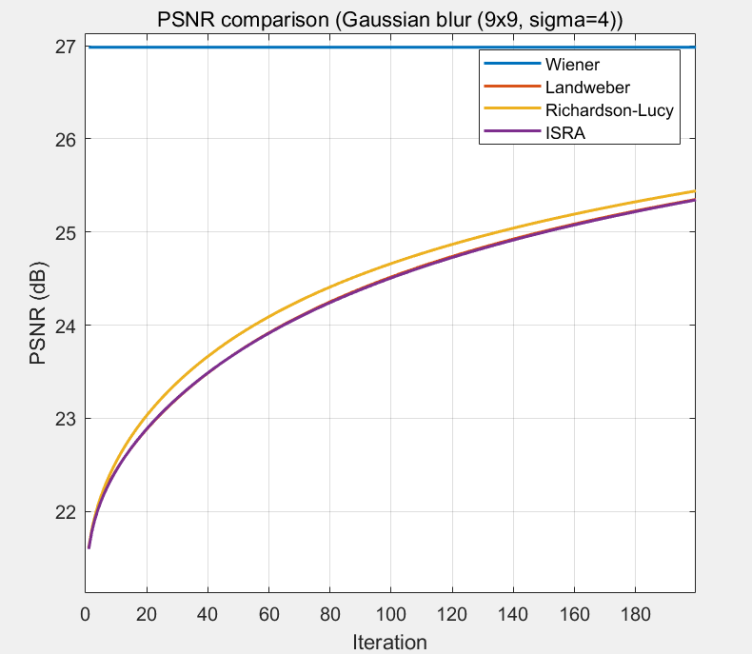
In the implementation, the circular convolution operations are efficiently computed using FFTs. The term h(-x,-y) corresponds to the flipped kernel in the spatial domain and to the complex conjugate of H(u,v) in the frequency domain.

4) PSNR comparison and conclusion

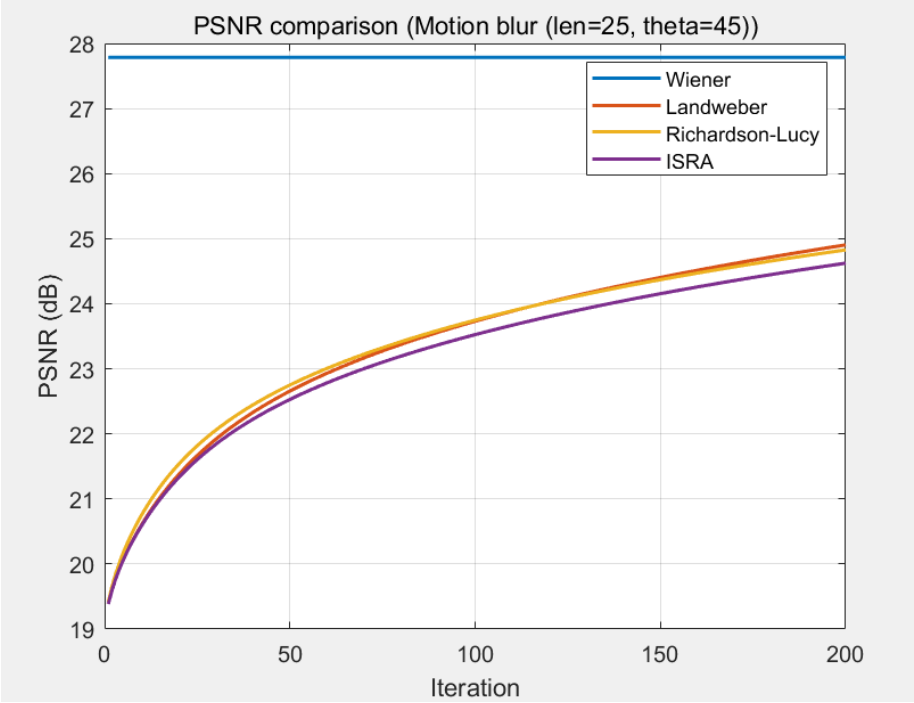
To meet the marking criteria, PSNR curves are plotted for Wiener, Landweber, Richardson–Lucy, and ISRA. Experiments are conducted for both Gaussian blur and motion blur. In this particular setting with additive Gaussian noise, Wiener filtering typically provides the best or most stable performance, while RL and ISRA may not show clear advantages and can amplify noise after many iterations. Therefore, early stopping is important for iterative methods.



Gaussian blur (9x9, sigma=4)



Motion blur (len=25, theta=45)



Task 5 (3 points): Handwritten digit recognition with basic artificial neural networks.
Matlab script **handwritten_digit_recognition_simple.m** provides you with a simple application of ANN for handwritten digit recognition. Your task is to modify the hidden layers of the network in order to achieve the accuracy higher than 93%. You are not allowed to use CNN layers. You are not allowed to use more than 100 neurons in total for all your hidden layers. You are not allowed to modify the training options.

You can observe that a higher accuracy can be easily achieved if convolutional layers are used: **handwritten_digit_recognition.m**. You can get more information about various layers used in ANN from **https://uk.mathworks.com/help/deeplearning/ug/create-simple-deep-learning-network-for-classification.html**

Solution：
In Task 5, we train a neural network to recognise handwritten digits (0–9) using the MATLAB DigitDataset. The task imposes strict constraints: (i) the training options must remain unchanged, (ii) convolutional layers are not allowed, and (iii) the total number of neurons across all hidden layers must not exceed 100. Therefore, the only way to improve performance is by redesigning the fully-connected hidden-layer architecture.

1) Dataset and experimental setup

The dataset is loaded using imageDatastore from MATLAB's built-in DigitDataset (28×28 grayscale images, 10 classes). The data is split into a training set and a validation set using

splitEachLabel, where 750 images per class are used for training and the remaining images per class are used for validation.

To ensure reproducibility, the random seed is fixed in all scripts using rng(0). This makes the train/validation split and network initialization deterministic, so that accuracy comparisons across architectures are fair and repeatable.

2) Accuracy metric

Validation performance is evaluated using classification accuracy:

Accuracy = (correctly classified validation images) / (total validation images).

3) Baseline ANN architecture (40–40–20)

The baseline network is a multilayer perceptron (MLP) with three fully-connected hidden layers and ReLU activations, followed by a softmax output layer. The baseline hidden-layer sizes are 40–40–20, giving a total of 100 hidden neurons. Under the fixed training settings, this baseline achieves a validation accuracy of 92.04%.

4) Improved ANN architecture (50–30–20)

To improve performance under the neuron budget, the hidden-layer sizes are redesigned to 50–30–20 (total = 100). This structure allocates more capacity to the first layer, which is beneficial because the input is high-dimensional (28×28 = 784 pixels). The later layers progressively compress the representation for classification. With the same training options and the same random seed, the improved ANN achieves a validation accuracy of 93.56%.

5) CNN reference result

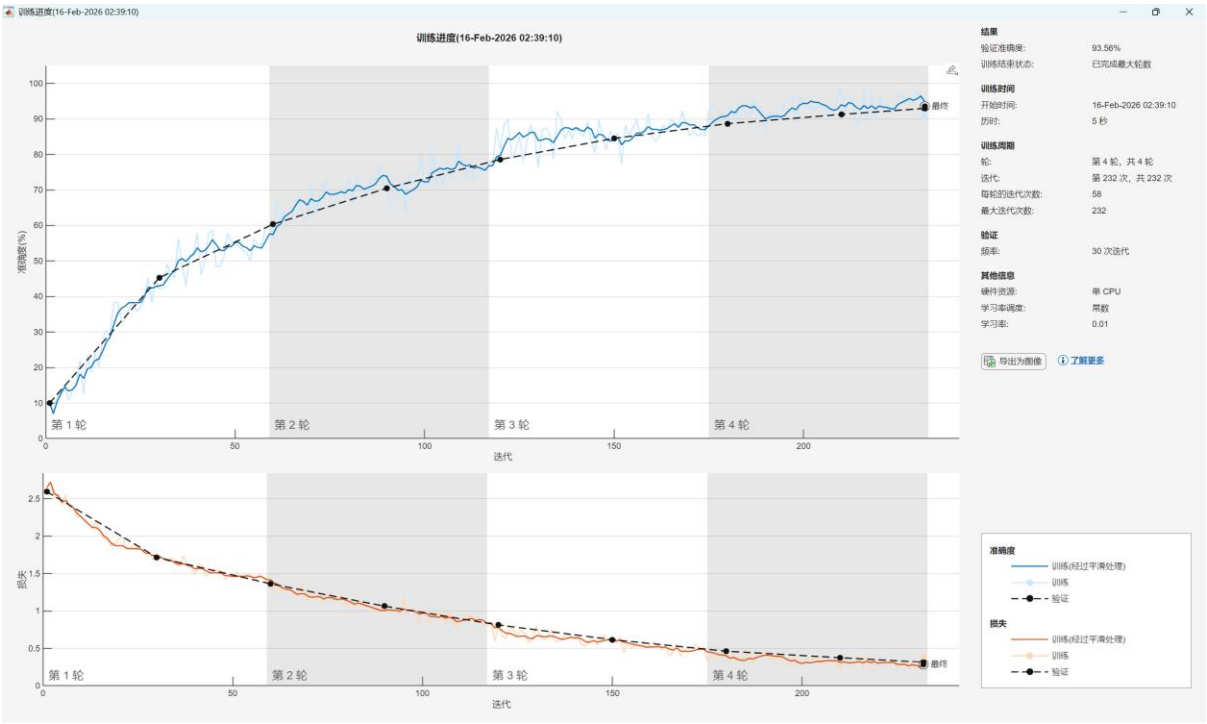
For reference, a convolutional neural network (CNN) architecture achieves 98.60% validation accuracy on the same dataset. This performance gap is expected because CNNs exploit spatial locality and translation invariance, which are crucial inductive biases for handwritten digit recognition. In contrast, fully-connected ANNs treat the input as a flat vector and cannot explicitly capture local stroke patterns.

6) Summary of results

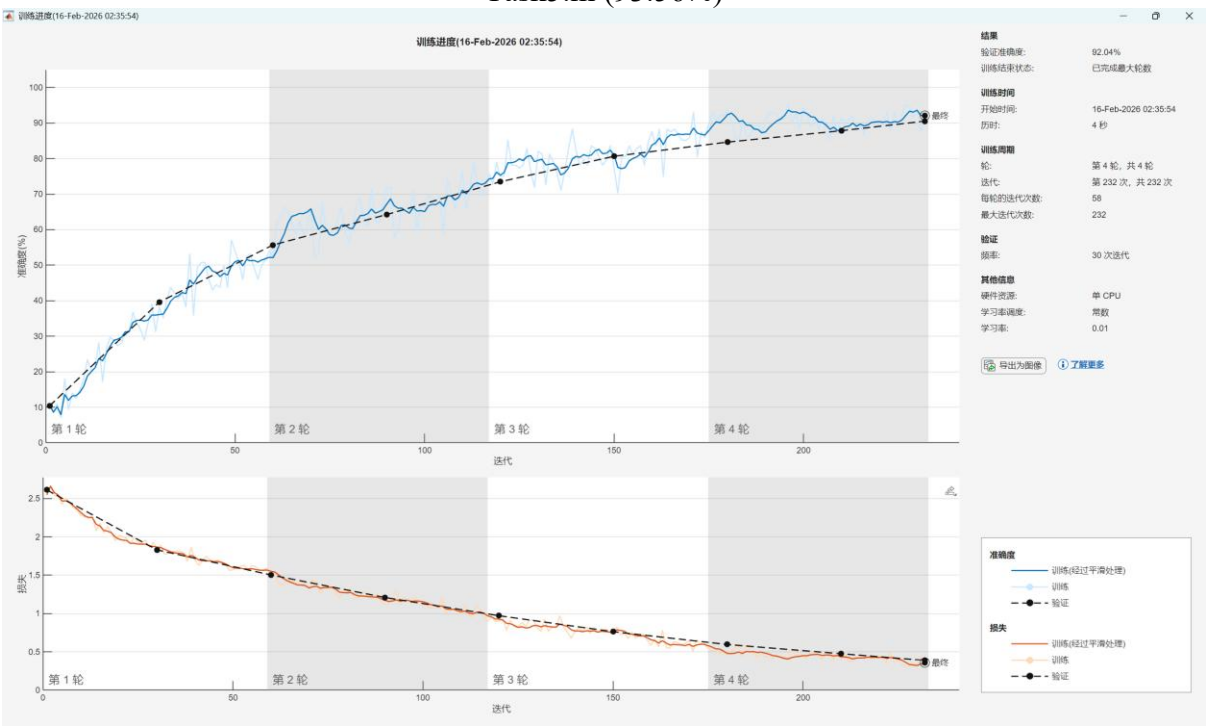
Model	Hidden layers	Validation accuracy
ANN (baseline)	40–40–20 (total 100)	92.04%
ANN (proposed)	50–30–20 (total 100)	93.56%
CNN (reference)	Convolutional layers	98.60%

7) Conclusion

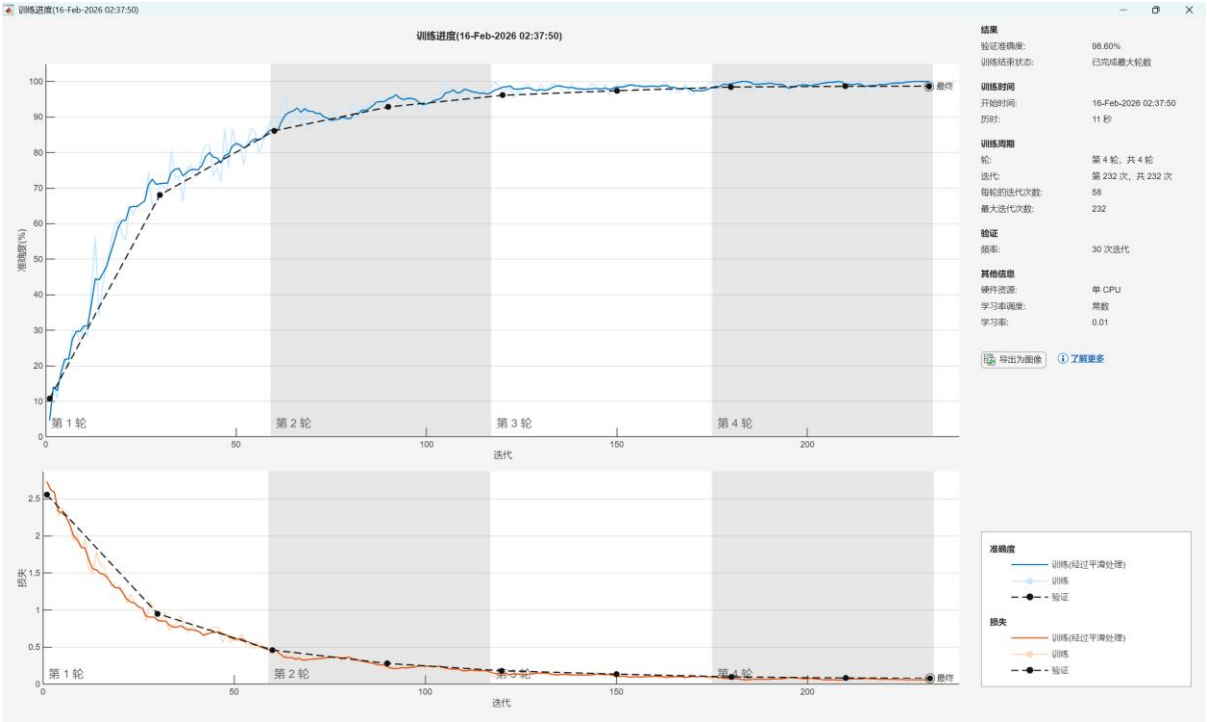
Within the task constraints, redesigning the ANN hidden-layer structure improves validation accuracy from 92.04% to 93.56% without modifying any training hyperparameters. Fixing rng(0) ensures that the comparison is reproducible and fair. The CNN result is substantially higher (98.60%), highlighting the advantage of convolutional inductive biases for image recognition tasks.



Task5.m (93.56%)



handwritten_digit_recognition_simple.m (92.04%)



handwritten_digit_recognition.m (98.60%)

Please submit a single report describing your results achieved for Tasks 1, 2, 3, 4 and 5 of the assignment. Together with the report, please submit your matlab (possibly python) scripts implementing your solutions to Tasks 1, 2, 3, 4 and 5.

Matlab Script

Github Link: https://github.com/WeiDai-David/HW_B31SE_Assignment1

Task 1a (4 points): Nonlinear image filtering:

```
close all; clear; clc;

%% Task 1a: Nonlinear image filtering (iterative)
% I0(x,y) = I(x,y)
% In+1(x,y) = sum_{i=-1..1, j=-1..1} w_ij * In(x+i, y+j) / sum w_ij
% w_ij = exp( -k * | In(x,y) - In(x+i, y+j) | )
%
% Notes:
% - weights depend on pixel position (x,y) and iteration n

%% Choose test image
% Make sure they are grayscale. If RGB, convert to gray.
% imgPath = '../../images/trui.tif';
% imgPath = '../../images/baboon.tif';
% imgPath = '../../images/barbara.tif';
% imgPath = '../../images/cameraman.tif';
% imgPath = '../../images/emu.tif';
% imgPath = '../../images/newborn.tif';
imgPath = '../../images/pout.tif';
% baboon.tif、barbara.tif、cameraman.tif、emu.tif、newborn.tif、pout.tif

I = im2double(imread(imgPath));
if ndims(I) == 3
    I = rgb2gray(I);
```

```
end

[rows, cols] = size(I);

%% Experiment settings
% At least 3-4 variations required by rubric.
k_list    = [1, 10, 50, 100, 500];     % try different k values
N_list    = [0, 5, 10, 20, 50, 100];      % try different iteration numbers

% For fair comparison, we compute:
% 1) simple averaging iterative smoothing
% 2) nonlinear filtering iterative smoothing

%% Part A: Simple averaging (baseline)
% This is the simplest case when all weights are equal: w_ij = 1.

fprintf("Running baseline: simple averaging (multiple N)...\n");

% We also try multiple iteration numbers (same idea as nonlinear experiments)
N_list_avg = [0, 1, 5, 10, 20, 50, 100];

% Store baseline results for different iteration counts
results_avg = cell(1, length(N_list_avg));

for nn = 1:length(N_list_avg)

    N_baseline = N_list_avg(nn);
    fprintf("  Averaging: N = %d iterations\n", N_baseline);

    im1_avg = I;
    im2_avg = zeros(rows, cols);

    for iter = 1 : N_baseline

        for x = 2 : rows-1
            for y = 2 : cols-1

                s = 0;

                % 3x3 neighborhood sum
                for i = -1 : 1
                    for j = -1 : 1
                        s = s + im1_avg(x+i, y+j);
                    end
                end

                % divide by 9 -> mean
                im2_avg(x,y) = s / 9;
            end
        end

        % update for next iteration
        im1_avg = im2_avg;
        im2_avg = zeros(rows, cols);

    end

    % store result for this N
    results_avg{nn} = im1_avg;
```

```
end


%% Part B: Nonlinear iterative filtering (main task)
% Key difference:
% - Instead of equal weights, we compute:
%   w_ij = exp( -k * |center - neighbor| )
% - Then do normalized weighted average.

% will store results for multiple (k, N) experiments.
results = cell(length(k_list), length(N_list));

fprintf("Running nonlinear filtering experiments...\n");

for kk = 1:length(k_list)
    k = k_list(kk);

    for nn = 1:length(N_list)
        N = N_list(nn);

        fprintf("  k = %d, N = %d iterations\n", k, N);

        % Initialize iteration
        im1 = I;                    % In
        im2 = zeros(rows, cols);    % In+1

        for iter = 1:N

            % For each inner pixel
            for x = 2:rows-1
                for y = 2:cols-1

                    center = im1(x,y);

                    weighted_sum = 0;
                    weight_total = 0;

                    % 3x3 neighborhood
                    for i = -1:1
                        for j = -1:1

                            neighbor = im1(x+i, y+j);

                            % Nonlinear weight (depends on current iter image)
                            w = exp( -k * abs(center - neighbor) );

                            weighted_sum = weighted_sum + w * neighbor;
                            weight_total = weight_total + w;

                        end
                    end

                    % Normalized weighted average
                    im2(x,y) = weighted_sum / weight_total;

                end
            end
```

```
            % Update for next iteration
            im1 = im2;
            im2 = zeros(rows, cols);
        end

        results{kk, nn} = im1;
    end
end

%% Visualization
% We show:
% 1) original image
% 2) baseline averaging
% 3) nonlinear results for multiple k and N

% --- Show baseline averaging results in a grid ---
figure('Name', 'Simple averaging results (multiple N)');

tiledlayout(1, length(N_list_avg), 'Padding', 'compact', 'TileSpacing',
'compact');

for nn = 1:length(N_list_avg)
    nexttile;
    imshow(results_avg{nn}, []);
    title(sprintf("Avg N=%d", N_list_avg(nn)));
end

% --- Show nonlinear results in a grid ---
figure('Name', 'Nonlinear filtering results');

tiledlayout(length(k_list), length(N_list), 'Padding', 'compact', 'TileSpacing',
'compact');

for kk = 1:length(k_list)
    for nn = 1:length(N_list)

        nexttile;
        imshow(results{kk, nn}, []);

        title(sprintf("k=%d, N=%d", k_list(kk), N_list(nn)));
    end
end
```

Task 1b (3 points): Low-light image enhancement

```
close all; clear; clc;

%% Task 1b: Low-light image enhancement using nonlinear filtering
% 1) Read an RGB image.
% 2) Convert RGB -> HSV using rgb2hsv().
% 3) Take only V channel (brightness) V(x,y).
% 4) Compute U(x,y) by applying Task 1a nonlinear filtering to V:
%       U = NLFilter(V, k, N=20)
% 5) Compute enhanced brightness:
%       E(x,y) = V(x,y) / ( U(x,y)^p + r )
%       r = 0.01, p in [0.8, 0.95]
% 6) Replace V channel by E, convert HSV -> RGB using hsv2rgb().
% 7) Run on 3-4 images from DarkFace dataset and show results.

%% User settings
```

```
imagesDir = "../../images/DarkFace/image";   % DarkFace folder
numImagesToShow = 4;          % requires 3-4 images

% Task requires:
N_filter = 20;      % 20 iterations
r = 0.01;           % avoid division by zero

% try p between 0.8 and 0.95
p_list = [0.80, 0.85, 0.90, 0.95];

% k controls the nonlinearity strength in the filter.
k = 50;

%% Collect images from dataset
% We search recursively for jpg/png/jpeg images inside datasetDir.
imgFiles = [
    dir(fullfile(imagesDir, "*.jpg"));
    dir(fullfile(imagesDir, "*.png"));
    dir(fullfile(imagesDir, "*.jpeg"))
];

if isempty(imgFiles)
    error("No images found in: %s", imagesDir);
end

fprintf("Found %d images in folder: %s\n", length(imgFiles), imagesDir);

% Make sure numImagesToShow is not larger than total
numImagesToShow = min(numImagesToShow, length(imgFiles));

% Randomly select images
randIdx = randperm(length(imgFiles), numImagesToShow);
selectedFiles = imgFiles(randIdx);


fprintf("Found %d images in dataset.\n", length(imgFiles));
fprintf("Using %d images for experiments.\n", numImagesToShow);

%% Process each selected image
for idx = 1:numImagesToShow

    imgPath = fullfile(selectedFiles(idx).folder, selectedFiles(idx).name);
    fprintf("\nProcessing image %d/%d: %s\n", idx, numImagesToShow, imgPath);

    % Read RGB image
    I_rgb = im2double(imread(imgPath));

    % If some image is grayscale by accident, convert to 3-channel RGB
    if ndims(I_rgb) == 2
        I_rgb = repmat(I_rgb, [1 1 3]);
    end

    % Convert RGB -> HSV
    I_hsv = rgb2hsv(I_rgb);

    % Extract V channel (brightness)
    V = I_hsv(:,:,3);

    % Apply nonlinear filtering to V to get U
```

```
    % U is the filtered brightness (smoothed but edge-preserving)
    U = nonlinear_filter_3x3(V, k, N_filter);

    %For each gamma p, compute enhancement E and show results

    figure('Name', sprintf("Task 1b - Image %d: %s", idx,
selectedFiles(idx).name), ...
           'NumberTitle', 'off');

    % Layout: original + 4 enhanced results (for different p)
    tiledlayout(1, 1 + length(p_list), 'Padding', 'compact', 'TileSpacing',
'compact');

    % ---- Tile 1: original ----
    nexttile;
    imshow(I_rgb, []);
    title(sprintf("Original\n%s", selectedFiles(idx).name), 'Interpreter',
'none');

    for pp = 1:length(p_list)
        p = p_list(pp);

        % Enhancement formula
        % E(x,y) = V(x,y) / ( U(x,y)^p + r )
        E = V ./ ( (U.^p) + r );

        % Important: keep values in [0,1] for valid HSV
        E = min(max(E, 0), 1);

        % Replace V channel by enhanced E
        I_hsv_enh = I_hsv;
        I_hsv_enh(:,:,3) = E;

        % Convert back to RGB
        I_rgb_enh = hsv2rgb(I_hsv_enh);

        % Show
        nexttile;
        imshow(I_rgb_enh, []);
        title(sprintf("Enhanced\np=%.2f", p));
    end

end

%% Local function: Nonlinear 3x3 iterative filter (Task 1a)
function out = nonlinear_filter_3x3(I, k, N)
% NONLINEAR_FILTER_3X3
% Implements the iterative nonlinear filtering scheme:
%   I0(x,y) = I(x,y)
%   In+1(x,y) = sum w_ij * In(x+i, y+j) / sum w_ij
%   w_ij = exp( -k * | In(x,y) - In(x+i, y+j) | )
% Input:
%   I : grayscale image in [0,1]
%   k : positive constant controlling weight decay
%   N : number of iterations
% Output:
%   out : filtered image after N iterations

    [rows, cols] = size(I);
```

```
    im1 = I;                % current iteration image In
    im2 = zeros(rows, cols); % next iteration image In+1

    for iter = 1:N

        % To avoid black borders, copy previous image first.
        % Then update only inner pixels.
        im2 = im1;

        for x = 2:rows-1
            for y = 2:cols-1

                center = im1(x,y);

                weighted_sum = 0;
                weight_total = 0;

                % 3x3 neighborhood
                for i = -1:1
                    for j = -1:1

                        neighbor = im1(x+i, y+j);

                        % weight depends on local intensity difference
                        w = exp(-k * abs(center - neighbor));

                        weighted_sum = weighted_sum + w * neighbor;
                        weight_total = weight_total + w;

                    end
                end

                % normalized weighted average
                im2(x,y) = weighted_sum / weight_total;

            end
        end

        % update
        im1 = im2;
    end

    out = im1;
end
```

Task 1c (3 points): Face detection or object detection for dark and brightened images

```
close all; clear; clc;
% matlab install Computer Vision Toolbox

%% Task 1c: Face detection on dark images vs enhanced images (DarkFace)
% - Use a publicly available face detection program (Matlab).
% - Compare detection results on:
%   (1) original low-light images
%   (2) images enhanced by Task 1b
% - Conclude whether enhancement improves detection results.
% Detector:
% - Matlab Computer Vision Toolbox:
%   vision.CascadeObjectDetector (publicly available)
```

```
% Enhancement:
% - Task 1b (HSV-based enhancement using nonlinear filtering on V channel)

%% Settings
imagesDir = "../../images/DarkFace/image";
K = 10;                                     % number of random images

% Task 1b parameters
N_filter = 20;      % number of iterations for nonlinear filtering (Task 1a)
r = 0.01;           % avoid division by zero
p = 0.80;           % gamma parameter (choose in [0.8, 0.95])
k = 50;             % nonlinearity strength for Task 1a filter


%% Collect images
imgFiles = [
    dir(fullfile(imagesDir, "*.jpg"));
    dir(fullfile(imagesDir, "*.png"));
    dir(fullfile(imagesDir, "*.jpeg"))
];

if isempty(imgFiles)
    error("No images found in: %s", imagesDir);
end

fprintf("Found %d images in folder: %s\n", length(imgFiles), imagesDir);

% Manual selection (priority)
% If you put filenames here, we will use them first.
% If this list is empty, we will randomly select K images.
manualList = {
    "1.png"
    "123.png"
    "1081.png"
    "1732.png"
    "1784.png"
    "2410.png"
    "2422.png"
    "3179.png"
    "3320.png"
    "3637.png"
};

% Build selectedFiles
if ~isempty(manualList)

    fprintf("Manual selection list is NOT empty. Using manually selected
images.\n");

    % Convert to struct array with fields {folder, name} like dir() output
    selectedFiles = struct('folder', {}, 'name', {});

    for m = 1:length(manualList)
        fname = manualList{m};
        fpath = fullfile(imagesDir, fname);

        if exist(fpath, 'file')
            selectedFiles(end+1).folder = imagesDir; %#ok<SAGROW>
            selectedFiles(end).name = fname;
```

```
        else
            warning("Manual file not found: %s", fpath);
        end
    end

    if isempty(selectedFiles)
        error("Manual list was given, but none of the files exist. Please check
filenames.");
    end

    % Override K to match manual selection count
    K = length(selectedFiles);
    fprintf("Using %d manually selected images.\n", K);

else

    fprintf("Manual selection list is empty. Randomly selecting K=%d images.\n",
K);

    K = min(K, length(imgFiles));
    randIdx = randperm(length(imgFiles), K);
    selectedFiles = imgFiles(randIdx);

    fprintf("Randomly selected %d images for Task 1c.\n", K);

end


%% Initialize face detector (Matlab public program)
detector = vision.CascadeObjectDetector();
% default is frontal face detector

%%  Store statistics for report table
filename_list = strings(K, 1);
dark_faces = zeros(K, 1);
enh_faces  = zeros(K, 1);
improved   = strings(K, 1);

%%  Main loop over images
for idx = 1:K

    fileName = selectedFiles(idx).name;
    imgPath  = fullfile(selectedFiles(idx).folder, fileName);

    fprintf("\n[%d/%d] Processing: %s\n", idx, K, fileName);

    %Read RGB image
    I_dark = im2double(imread(imgPath));

    % Ensure it is RGB
    if ndims(I_dark) == 2
        I_dark = repmat(I_dark, [1 1 3]);
    end

    %% (A) Face detection on original dark image
    bboxes_dark = step(detector, I_dark);
    num_dark = size(bboxes_dark, 1);

    %% (B) Enhance image using Task 1b
```

```
    I_enh = enhance_lowlight_task1b(I_dark, k, N_filter, p, r);

    %% (C) Face detection on enhanced image
    bboxes_enh = step(detector, I_enh);
    num_enh = size(bboxes_enh, 1);

    %% Save stats
    filename_list(idx) = fileName;
    dark_faces(idx) = num_dark;
    enh_faces(idx)  = num_enh;

    if num_enh > num_dark
        improved(idx) = "Yes";
    elseif num_enh == num_dark
        improved(idx) = "Same";
    else
        improved(idx) = "No";
    end

    %% Visualization: side-by-side comparison with bounding boxes
    figure('Name', sprintf("Task 1c: %s", fileName), 'NumberTitle', 'off');

    tiledlayout(1, 2, 'Padding', 'compact', 'TileSpacing', 'compact');

    % ---- Left: original dark + boxes ----
    nexttile;
    imshow(I_dark, []);
    title(sprintf("Original (dark)\n%s\nFaces=%d", fileName, num_dark), ...
          'Interpreter', 'none');

    hold on;
    for b = 1:num_dark
        rectangle('Position', bboxes_dark(b,:), 'EdgeColor', 'g', 'LineWidth', 2);
    end
    hold off;

    % ---- Right: enhanced + boxes ----
    nexttile;
    imshow(I_enh, []);
    title(sprintf("Enhanced (Task1b)\np=%.2f, k=%d, N=%d\nFaces=%d", ...
          p, k, N_filter, num_enh));

    hold on;
    for b = 1:num_enh
        rectangle('Position', bboxes_enh(b,:), 'EdgeColor', 'g', 'LineWidth', 2);
    end
    hold off;

end

%% Summary table + quantitative conclusion
T = table(filename_list, dark_faces, enh_faces, improved, ...
          'VariableNames', {'filename', 'dark_num_faces', 'enhanced_num_faces',
'improved'});

disp("==================================================");
disp("Task 1c summary table:");
disp(T);
```

```
% Hit rate: percentage of images where at least one face is detected
dark_hit_rate = mean(dark_faces > 0);
enh_hit_rate  = mean(enh_faces  > 0);

% Average number of detections per image
dark_avg_faces = mean(dark_faces);
enh_avg_faces  = mean(enh_faces);

fprintf("\n================ Task 1c Quantitative Summary ================\n");
fprintf("Dark images:     hit rate = %.2f%%, avg faces = %.2f\n",
100*dark_hit_rate, dark_avg_faces);
fprintf("Enhanced images: hit rate = %.2f%%, avg faces = %.2f\n",
100*enh_hit_rate,  enh_avg_faces);

if enh_hit_rate > dark_hit_rate
    fprintf("Conclusion: Enhancement improves face detection hit rate.\n");
elseif enh_hit_rate == dark_hit_rate
    fprintf("Conclusion: Enhancement does not change the face detection hit
rate.\n");
else
    fprintf("Conclusion: Enhancement reduces face detection hit rate (possible
over-enhancement artifacts).\n");
end
fprintf("============================================================\n");

%% Local function: Task 1b enhancement (HSV + nonlinear filter)
function I_rgb_enh = enhance_lowlight_task1b(I_rgb, k, N_filter, p, r)
% ENHANCE_LOWLIGHT_TASK1B
% Implements Task 1b enhancement:
% 1) RGB -> HSV
% 2) Take V channel
% 3) U = nonlinear_filter_3x3(V, k, N_filter)
% 4) E = V / (U^p + r)
% 5) Replace V by E, HSV -> RGB

    I_hsv = rgb2hsv(I_rgb);
    V = I_hsv(:,:,3);

    % Task 1a nonlinear filtering on V
    U = nonlinear_filter_3x3(V, k, N_filter);

    % Enhancement formula (Task 1b)
    E = V ./ ((U.^p) + r);

    % Clamp to [0,1] to keep HSV valid
    E = min(max(E, 0), 1);

    % Replace V channel and convert back
    I_hsv(:,:,3) = E;
    I_rgb_enh = hsv2rgb(I_hsv);
end

%% Local function: Task 1a nonlinear filter (3x3, iterative)
function out = nonlinear_filter_3x3(I, k, N)
% NONLINEAR_FILTER_3X3
% Implements Task 1a iterative nonlinear filtering:
%   I_{n+1}(x,y) = sum w_ij * I_n(x+i,y+j) / sum w_ij
%   w_ij = exp( -k * |I_n(x,y) - I_n(x+i,y+j)| )
```

```
    [rows, cols] = size(I);

    im1 = I;
    im2 = zeros(rows, cols);

    for iter = 1:N

        % avoid black border: copy previous image first
        im2 = im1;

        for x = 2:rows-1
            for y = 2:cols-1

                center = im1(x,y);

                weighted_sum = 0;
                weight_total = 0;

                for i = -1:1
                    for j = -1:1

                        neighbor = im1(x+i, y+j);

                        w = exp(-k * abs(center - neighbor));

                        weighted_sum = weighted_sum + w * neighbor;
                        weight_total = weight_total + w;

                    end
                end

                im2(x,y) = weighted_sum / weight_total;

            end
        end

        im1 = im2;
    end

    out = im1;
end
```

Task 2 (3 points): Local histogram equalisation

```
close all; clear; clc;

%% Task 2: Local histogram equalisation
% - Implement a local histogram equalization scheme
% - Apply to unevenly illuminated images (e.g., tun.jpg)
% - Evaluate suitability for low-light enhancement
% Notes:
% - We implement local HE on grayscale using a sliding window.
% - For color images, we process the V channel in HSV to avoid color shifts.

%% Settings
imgPath = "../../images/tun.jpg";   % tun.jpg path
% imgPath = "../../images/low_light/samsung_galaxy.jpg";
winSize = 31;                       % local window size (odd): 15/31/51 are typical
the larger the closer it is to the global HE
nBins   = 256;                      % histogram bins (8-bit)
```

```
% For comparison baselines
doCompareWithGlobalHE = true;
doCompareWithCLAHE    = true;  % uses adapthisteq (allowed as comparison baseline)

%% Read image
I = im2double(imread(imgPath));
isColor = (ndims(I) == 3);

if ~isColor
    I_gray = I;
else
    % Use HSV: apply enhancement to V channel only (brightness)
    I_hsv = rgb2hsv(I);
    I_gray = I_hsv(:,:,3);  % treat V as grayscale target
end

%% (A) Local histogram equalization (our implementation)
fprintf("Running local histogram equalization (win=%d, bins=%d)...\n", winSize,
nBins);
I_local = local_hist_equalize(I_gray, winSize, nBins);

%% (B) Global histogram equalization (baseline)
if doCompareWithGlobalHE
    % histeq expects [0,1] grayscale; returns [0,1]
    I_global = histeq(I_gray, nBins);
end

%% (C) CLAHE (baseline) - Matlab adapthisteq
if doCompareWithCLAHE
    % CLAHE is a common improved local HE (contrast-limited)
    I_clahe = adapthisteq(I_gray, ...
        'NumTiles', [8 8], ...      % grid of contextual regions
        'ClipLimit', 0.01);         % contrast limiting
end

%% Reconstruct output images
if ~isColor
    out_local  = I_local;
    out_global = [];
    out_clahe  = [];
    if doCompareWithGlobalHE, out_global = I_global; end
    if doCompareWithCLAHE,    out_clahe  = I_clahe;  end
else
    % Replace V channel and convert back to RGB for display
    out_local = I_hsv;
    out_local(:,:,3) = I_local;
    out_local = hsv2rgb(out_local);

    if doCompareWithGlobalHE
        out_global = I_hsv; out_global(:,:,3) = I_global; out_global =
hsv2rgb(out_global);
    else
        out_global = [];
    end

    if doCompareWithCLAHE
        out_clahe = I_hsv; out_clahe(:,:,3) = I_clahe; out_clahe =
hsv2rgb(out_clahe);
```

```
    else
        out_clahe = [];
    end
end

%% Visualization: results + histograms
figure('Name', 'Task 2: Local Histogram Equalisation', 'NumberTitle', 'off');

if doCompareWithGlobalHE && doCompareWithCLAHE
    tiledlayout(2, 2, 'Padding', 'compact', 'TileSpacing', 'compact');

    nexttile; imshow(I, []);          title("Original");
    nexttile; imshow(out_global, []); title("Global HE (histeq)");
    nexttile; imshow(out_local, []);  title(sprintf("Local HE (win=%d)",
winSize));
    nexttile; imshow(out_clahe, []);  title("CLAHE (adapthisteq)");
elseif doCompareWithGlobalHE
    tiledlayout(1, 3, 'Padding', 'compact', 'TileSpacing', 'compact');
    nexttile; imshow(I, []);          title("Original");
    nexttile; imshow(out_global, []); title("Global HE (histeq)");
    nexttile; imshow(out_local, []);  title(sprintf("Local HE (win=%d)",
winSize));
elseif doCompareWithCLAHE
    tiledlayout(1, 3, 'Padding', 'compact', 'TileSpacing', 'compact');
    nexttile; imshow(I, []);         title("Original");
    nexttile; imshow(out_local, []); title(sprintf("Local HE (win=%d)", winSize));
    nexttile; imshow(out_clahe, []); title("CLAHE (adapthisteq)");
else
    tiledlayout(1, 2, 'Padding', 'compact', 'TileSpacing', 'compact');
    nexttile; imshow(I, []);         title("Original");
    nexttile; imshow(out_local, []); title(sprintf("Local HE (win=%d)", winSize));
end

% Optional: show histograms of the processed channel
figure('Name', 'Histograms on processed channel', 'NumberTitle', 'off');
tiledlayout(1, 2, 'Padding', 'compact', 'TileSpacing', 'compact');

% --- Original histogram ---
nexttile;
histogram(I_gray(:), nBins);
xlim([0 1]);
title("Original channel histogram");
xlabel("Intensity");
ylabel("Count");

% --- Local HE histogram ---
nexttile;
histogram(I_local(:), nBins);
xlim([0 1]);
title("Local HE channel histogram");
xlabel("Intensity");
ylabel("Count");

%% Simple evaluation notes printed to console
fprintf("\n================= Task 2 Evaluation Notes ================\n");
fprintf("- Local HE can correct uneven illumination by adapting contrast
locally.\n");
fprintf("- However, plain Local HE often amplifies noise in very dark
regions.\n");
```

```
fprintf("- CLAHE usually reduces this problem via contrast limiting.\n");
fprintf("Try different winSize (e.g., 15/31/51) to see trade-offs.\n");
fprintf("============================================================\n");

%% Local function: Local histogram equalization (our implementation)
function out = local_hist_equalize(I, winSize, nBins)
% LOCAL_HIST_EQUALIZE
% Basic local histogram equalization:
% For each pixel, build histogram in a local window and map the center pixel
% using the local CDF.
% Input:
%   I       : grayscale image in [0,1]
%   winSize : odd integer window size (e.g., 31)
%   nBins   : number of histogram bins (e.g., 256)
% Output:
%   out     : locally histogram-equalized image in [0,1]

    if mod(winSize, 2) == 0
        error("winSize must be odd.");
    end

    [rows, cols] = size(I);
    pad = floor(winSize / 2);

    % Pad image to handle borders (replicate avoids artificial dark borders)
    Ipad = padarray(I, [pad pad], 'replicate', 'both');

    out = zeros(rows, cols);

    % Precompute bin edges for discretization
    % Map [0,1] -> 1..nBins
    for x = 1:rows
        for y = 1:cols

            % Extract local window
            wx = x : x + 2*pad;
            wy = y : y + 2*pad;
            patch = Ipad(wx, wy);

            % Compute histogram (integer bin indices)
            % Convert patch values to bin indices in [1..nBins]
            idx = floor(patch * (nBins - 1)) + 1;

            h = zeros(nBins, 1);
            % Accumulate histogram counts
            for t = 1:numel(idx)
                h(idx(t)) = h(idx(t)) + 1;
            end

            % Compute CDF
            cdf = cumsum(h) / sum(h);

            % Map center pixel using local CDF
            centerVal = I(x, y);
            centerBin = floor(centerVal * (nBins - 1)) + 1;

            out(x, y) = cdf(centerBin);

        end
```

```
    end

    % Ensure valid range
    out = min(max(out, 0), 1);
end
```

Task 3 (3 points): Image filtering in frequency domain

```
close all; clear; clc;

%% Task 3: Image filtering in frequency domain (Notch filtering)
% Goal:
% - Compute and visualise Fourier transform using:
%       log(abs(fftshift(fft2(...))))
% - Identify periodic noise frequencies (4 small crosses) using impixelinfo
% - Construct a notch filter (band-stop filter) using small circles/rectangles
% - Suppress periodic noise while preserving image quality
% - Show:
%   (1) corrupted image
%   (2) Fourier magnitude spectrum
%   (3) notch filter mask in frequency domain
%   (4) reconstructed image after filtering

%% Read image
imgPath = "../../images/eye-hand.png";

I_rgb = imread(imgPath);
I = im2double(im2gray(I_rgb));   % grayscale double in [0,1]

[rows, cols] = size(I);

%% Fourier transform
F = fft2(I);
F_shift = fftshift(F);

% magnitude spectrum for visualization
mag = log(1 + abs(F_shift));

%% Show image + spectrum (use impixelinfo to locate noise peaks)
figure('Name', 'Task 3: Locate periodic noise peaks', 'NumberTitle', 'off');
tiledlayout(1, 2, 'Padding', 'compact', 'TileSpacing', 'compact');

nexttile;
imshow(I, []);
title("Corrupted image (eye-hand.png)");

nexttile;
imshow(mag, []);
title("Magnitude spectrum: log(1 + |fftshift(F)|)");

% Use impixelinfo to read coordinates of bright peaks.
impixelinfo;

fprintf("\n============================================================\n");
fprintf("Use impixelinfo on the spectrum figure to find the 4 bright peaks.\n");
fprintf("Write down their (x,y) coordinates in the shifted spectrum image.\n");
fprintf("The center is approximately (cols/2, rows/2).\n");
fprintf("============================================================\n");

%% Step 2: Construct notch filter
```

```
% fill in peak coordinates after reading them from impixelinfo.
% NOTE:
% - Coordinates in Matlab display are (x,y) = (column, row).
% - F_shift has size rows x cols.

peaks = [
    128, 157;
    385, 100;
    128, 413;
    385, 357
];

%%  Notch filter parameters
notchRadius = 8;     % radius of each notch (try 3~10)
mask = ones(rows, cols);   % 1 = keep frequency, 0 = remove

% Build circular notches at peak locations
for t = 1:size(peaks, 1)

    x0 = peaks(t, 1);   % column
    y0 = peaks(t, 2);   % row

    if x0 == 0 && y0 == 0
        continue; % skip placeholder rows
    end

    for y = 1:rows
        for x = 1:cols
            if (x - x0)^2 + (y - y0)^2 <= notchRadius^2
                mask(y, x) = 0;
            end
        end
    end
end

%% Apply notch filter in frequency domain
F_filtered_shift = F_shift .* mask;

% Inverse shift and inverse FFT
F_filtered = ifftshift(F_filtered_shift);
I_rec = real(ifft2(F_filtered));

% Clip for display
I_rec = min(max(I_rec, 0), 1);

%% Visualisation: mask + filtered spectrum + reconstructed image
figure('Name', 'Task 3: Notch filtering results', 'NumberTitle', 'off');
tiledlayout(2, 2, 'Padding', 'compact', 'TileSpacing', 'compact');

nexttile;
imshow(I, []);
title("Original corrupted image");

nexttile;
imshow(mag, []);
title("Original spectrum log(1+|F|)");

nexttile;
imshow(mask, []);
```

```
title(sprintf("Notch filter mask (radius=%d)", notchRadius));

nexttile;
imshow(I_rec, []);
title("Reconstructed image after notch filtering");

%% Optional: show filtered spectrum
mag_filtered = log(1 + abs(F_filtered_shift));

figure('Name', 'Filtered spectrum', 'NumberTitle', 'off');
imshow(mag_filtered, []);
title("Filtered spectrum log(1+|F_filtered|)");
```

Task 4a (3 points): Image deblurring by the Wiener filter

```
close all; clear; clc;

%% Task 4a: Image deblurring by the Wiener filter
% Model:
%   g(x,y) = h(x,y) ⊗ f(x,y) + n(x,y)
% Fourier domain:
%   G(u,v) = H(u,v)F(u,v) + N(u,v)
% Wiener filter solution:
%   F_hat(u,v) = ( conj(H(u,v)) / ( |H(u,v)|^2 + K ) ) * G(u,v)
% Requirements:
% - Do NOT use deconvwnr
% - Use psf2otf to handle blur kernel
% - Test on motion blur and Gaussian blur with different kernels

%% 1) Read test image (grayscale)
imgPath = "../../images/cameraman.tif";     % you can change to barbara_face.png
etc.
f = im2double(imread(imgPath));

if ndims(f) == 3
    f = rgb2gray(f);
end

%%  2) Noise settings
noise_mean = 0;
noise_var  = 1e-6;   % 1e-6 or 1e-5 are typical

%% 3) Experiment list (Gaussian + Motion)
% We will test multiple kernels to satisfy rubric.

expList = {};

% --- Gaussian blur kernels ---
expList{end+1} = struct("type","gaussian", "size",[9 9],  "sigma",2);
expList{end+1} = struct("type","gaussian", "size",[15 15],"sigma",4);

% --- Motion blur kernels ---
expList{end+1} = struct("type","motion", "len",15, "theta",0);
expList{end+1} = struct("type","motion", "len",25, "theta",45);

%%  4) Wiener K values (try several)
% K approximates noise-to-signal power ratio.
K_list = [1e-4, 5e-4, 1e-3];

%% 5) Run experiments
```

```
for e = 1:length(expList)

    %% Create blur kernel h(x,y)
    exp = expList{e};

    if exp.type == "gaussian"
        h = fspecial("gaussian", exp.size, exp.sigma);
        expName = sprintf("Gaussian (%dx%d, sigma=%.1f)", exp.size(1),
exp.size(2), exp.sigma);

    elseif exp.type == "motion"
        h = fspecial("motion", exp.len, exp.theta);
        expName = sprintf("Motion (len=%d, theta=%d)", exp.len, exp.theta);
    end

    % Normalize kernel to ensure sum(h)=1
    h = h / sum(h(:));

    %% Blur + add noise to generate degraded image g(x,y)
    g = imfilter(f, h, "conv", "circular");
    g = imnoise(g, "gaussian", noise_mean, noise_var);

    %% Convert blur kernel to OTF (frequency response)
    % H(u,v) has the same size as image g
    H = psf2otf(h, size(g));

    %% FFT of degraded image
    G = fft2(g);

    %% Display results for different K values
    figure("Name", sprintf("Task 4a: %s", expName), "NumberTitle","off");
    tiledlayout(1, 2 + length(K_list),
"Padding","compact","TileSpacing","compact");

    % ---- Tile 1: original ----
    nexttile;
    imshow(f, []);
    title("Original f(x,y)");

    % ---- Tile 2: blurred + noisy ----
    nexttile;
    imshow(g, []);
    title(sprintf("Blurred+Noise g(x,y)\nPSNR=%.2f dB", psnr(g,f)));

    %% Wiener restoration for each K
    for kk = 1:length(K_list)

        K = K_list(kk);

        % Wiener filter:
        % W(u,v) = conj(H) / (|H|^2 + K)
        W = conj(H) ./ (abs(H).^2 + K);

        % Apply Wiener filter in frequency domain
        F_hat = W .* G;

        % Inverse FFT to reconstruct
        f_rec = real(ifft2(F_hat));
```

```
        % Clip to valid range [0,1]
        f_rec = min(max(f_rec, 0), 1);

        % Show reconstructed image
        nexttile;
        imshow(f_rec, []);
        title(sprintf("Wiener\nK=%.0e\nPSNR=%.2f", K, psnr(f_rec,f)));
    end

end
```

Task 4b (3 points): Image deblurring by ISRA

```
close all; clear; clc;

%% Task 4b: Image deblurring by ISRA
% Compare ISRA with:
% - Wiener (custom Wiener from Task4a)
% - Landweber
% - Richardson-Lucy
% Degradation model:
%   g = h ⊗ f + n
% ISRA iteration:
%   I0 = g
%   I_{n+1} = I_n .* [ (h(-x,-y) ⊗ g) ./ (h(-x,-y) ⊗ (h ⊗ I_n)) ]
% Notes:
% - In this task we keep the same settings as deblur.m
% - Because noise is additive Gaussian, RL and ISRA may not outperform Wiener.

%% 1) Choose blur kernel type: Gaussian OR Motion
testCase = "gaussian";   % "gaussian" or "motion"
% testCase = "motion";

%% 2) Load image f (ground truth)
f = im2double(imread("cameraman.tif"));
if ndims(f) == 3
    f = rgb2gray(f);
end

%% 3) Create blur kernel h
if testCase == "gaussian"
    h = fspecial("gaussian", [9 9], 4);
    caseName = "Gaussian blur (9x9, sigma=4)";
elseif testCase == "motion"
    h = fspecial("motion", 25, 45);
    caseName = "Motion blur (len=25, theta=45)";
end

h = h ./ sum(h(:));

%% 4) Blur operator (circular convolution)
blur = @(im) imfilter(im, h, "conv", "circular");

%% 5) Add Gaussian noise
noise_mean = 0;
noise_var  = 1e-6;

g = blur(f);
g = imnoise(g, "gaussian", noise_mean, noise_var);
```

```
%% 6) Frequency response H for fast computations
H = psf2otf(h, size(g));
G = fft2(g);

%% 7) Wiener baseline
K_wiener = 1e-4;

% --- Custom Wiener (recommended, consistent with Task 4a) ---
W = conj(H) ./ (abs(H).^2 + K_wiener);
Fhat = W .* G;
WienerRec = real(ifft2(Fhat));
WienerRec = min(max(WienerRec, 0), 1);

%% 8) Initialize iterative methods
maxiter = 200;   % 3000 is too slow; 200 is enough for curves

RL = g;
Lw = g;
ISRA = g;

psnr0 = psnr(f, g);

psnrW  = psnr(WienerRec, f) * ones(maxiter, 1);
psnrRL = zeros(maxiter, 1);
psnrLw = zeros(maxiter, 1);
psnrIS = zeros(maxiter, 1);

%% 9) Precompute h(-x,-y) in frequency domain
% In circular convolution:
% h(-x,-y) corresponds to conj(H) in frequency domain.

%% 10) Iterations: RL, Landweber, ISRA
fprintf("Running iterative methods for: %s\n", caseName);
fprintf("Initial PSNR (blurred+noise): %.2f dB\n", psnr0);

for i = 1:maxiter

    %% ---------- Richardson-Lucy ----------
    % RL_{n+1} = RL_n .* [ h(-x) ⊗ ( g ./ (RL_n ⊗ h) ) ]
    denom_RL = blur(RL);
    denom_RL = max(denom_RL, 1e-12);   % avoid division by zero

    ratio = g ./ denom_RL;
    RL = RL .* real(ifft2( fft2(ratio) .* conj(H) ));

    RL = max(RL, 0);  % RL is usually constrained to be nonnegative
    psnrRL(i) = psnr(RL, f);

    %% ---------- Landweber ----------
    % Lw_{n+1} = Lw_n + h(-x) ⊗ ( g - (Lw_n ⊗ h) )
    residual = g - blur(Lw);
    Lw = Lw + real(ifft2( fft2(residual) .* conj(H) ));

    Lw = min(max(Lw, 0), 1);
    psnrLw(i) = psnr(Lw, f);

    %% ---------- ISRA ----------
    % I_{n+1} = I_n .* [ (h(-x) ⊗ g) ./ (h(-x) ⊗ (h ⊗ I_n)) ]
```

```
    %
    % Numerator: h(-x) ⊗ g
    num = real(ifft2( fft2(g) .* conj(H) ));

    % Denominator: h(-x) ⊗ (h ⊗ I_n)
    tmp = blur(ISRA);   % h ⊗ I_n
    den = real(ifft2( fft2(tmp) .* conj(H) ));

    den = max(den, 1e-12);  % avoid division by zero

    ISRA = ISRA .* (num ./ den);

    ISRA = max(ISRA, 0);
    psnrIS(i) = psnr(ISRA, f);

    %% print occasionally
    if mod(i, 20) == 0
        fprintf("iter=%d  PSNR: RL=%.2f  Lw=%.2f  ISRA=%.2f\n", ...
            i, psnrRL(i), psnrLw(i), psnrIS(i));
    end
end

%%  11) Show final restored images
figure("Name", "Task 4b: final reconstructions", "NumberTitle","off");
imshow([f, g, WienerRec, Lw, RL, ISRA], []);
title("f (GT) | g (blur+noise) | Wiener | Landweber | Richardson-Lucy | ISRA");

%% 12) PSNR curves (rubric requirement)

figure("Name", "Task 4b: PSNR vs Iteration", "NumberTitle","off");
plot(psnrW,  "LineWidth", 1.5); hold on;
plot(psnrLw, "LineWidth", 1.5);
plot(psnrRL, "LineWidth", 1.5);
plot(psnrIS, "LineWidth", 1.5);

xlabel("Iteration");
ylabel("PSNR (dB)");
title(sprintf("PSNR comparison (%s)", caseName));
legend("Wiener", "Landweber", "Richardson-Lucy", "ISRA");
grid on;

%% 13) Print quantitative summary
fprintf("\n================ Task 4b Summary ================\n");
fprintf("Case: %s\n", caseName);
fprintf("Wiener final PSNR: %.2f dB\n", psnr(WienerRec, f));
fprintf("Landweber best PSNR: %.2f dB\n", max(psnrLw));
fprintf("RL best PSNR: %.2f dB\n", max(psnrRL));
fprintf("ISRA best PSNR: %.2f dB\n", max(psnrIS));
fprintf("================================================\n");
```

Task 5 (3 points): Handwritten digit recognition with basic artificial neural networks

```
close all;
clear all;
clc;
% matlab install Deep Learning Toolbox
rng(0);   % Fixed random number seed, making split and initialization more stable
% Load Image Data
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
```

```
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true,'LabelSource','foldernames');

% Display some of the images in the datastore
figure;
perm = randperm(10000,20);
for i = 1:20
    subplot(4,5,i);
    imshow(imds.Files{perm(i)});
end

% Calculate the number of images in each category
labelCount = countEachLabel(imds)

% Each image is 28-by-28-by-1 pixels.
img = readimage(imds,1);
size(img)

% Specify Training and Validation Sets
% Divide the data into training and validation data sets,
% so that each category in the training set contains 750 images,
% and the validation set contains the remaining images from each label.
% splitEachLabel splits the datastore digitData into two new datastores,
% trainDigitData and valDigitData.
numTrainFiles = 750;
[imdsTrain,imdsValidation] = splitEachLabel(imds,numTrainFiles,'randomize');

% Define the convolutional neural network architecture
layers = [
    imageInputLayer([28 28 1])

% hidden layers

% 1st hidden layer
fullyConnectedLayer(50)
batchNormalizationLayer
reluLayer

% 2nd hiden layer
fullyConnectedLayer(30)
batchNormalizationLayer
reluLayer

% 3rd hiden layer
fullyConnectedLayer(20)
batchNormalizationLayer
reluLayer

% softmax layer
    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

% Specify Training Options
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsValidation, ...
```

```
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','training-progress');

% Train Network Using Training Data
net = trainNetwork(imdsTrain,layers,options);

% Classify Validation Images and Compute Accuracy
YPred = classify(net,imdsValidation);
YValidation = imdsValidation.Labels;

accuracy = sum(YPred == YValidation)/numel(YValidation)
```