1141　計算機視覺　54013

HW1 Threshold and Noise Reduction

S1254040　資工三　羅章弘

民國一一四年十月八日星期三

目錄

Environment

- Python 版本：3.12.7
- numpy 版本：2.2.6
- opencv-python 版本：4.12.0.88

Threshold

Local methods

A. Mean thresholding 平均閥值

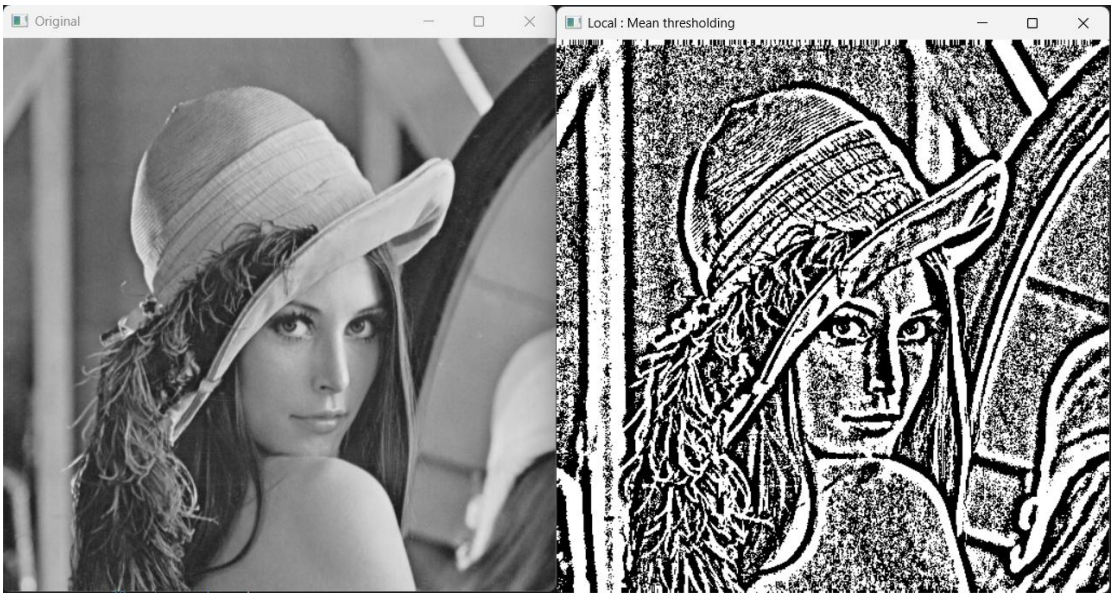
1. 直觀版本：逐像素取區域平均，時間較長
   時間複雜度：$O(h \times w \times \text{window size}^2)$

```python
def local_thresholding_naive(image, window_size=5):
    """
    使用局部均值法進行閾值化（直觀版本）
    image: 灰階影像（numpy array）
    window_size: 鄰域大小（必須為奇數）
    """
    h, w = image.shape      # 影像高度與寬度
    r = window_size // 2    # 半徑
    binary_image = np.zeros_like(image, dtype=np.uint8) # 初始化二值影像

    for y in range(h):
        for x in range(w):
            # 鄰域範圍（防止超出邊界）
            y1, y2 = max(0, y - r), min(h, y + r + 1)   # y1, y2 是區域的上下邊界
            x1, x2 = max(0, x - r), min(w, x + r + 1)   # x1, x2 是區域的左右邊界

            region = image[y1:y2, x1:x2]                # 取出鄰域
            T = np.mean(region)                         # 計算均值作為閾值
            binary_image[y, x] = 255 if image[y, x] > T else 0 # 二值化

    return binary_image
```

2. 積分圖版本：演算法優化，快速平均值計算，時間較短
   時間複雜度：$O(h \times w)$

```python
def local_thresholding_integral(image, window_size=5):
    """
    使用積分圖加速的局部均值法閾值化
    image: 灰階影像 (numpy array)
    window_size: 鄰域大小 (必須為奇數)
    """
    h, w = image.shape      # 影像高度與寬度
    r = window_size // 2    # 半徑
    binary_image = np.zeros_like(image, dtype=np.uint8) # 初始化二值影像

    # 預先計算積分圖 (integral 輸出比原圖大一圈)
    # 積分圖其實就是二維前綴和 (2D prefix sum)
    integral = cv2.integral(image, sdepth=cv2.CV_64F) # 使用 64 位元浮點數以防溢位

    for y in range(h):
        for x in range(w):
            # 鄰域範圍 (使用 integral 圖座標多 +1)
            y1, y2 = max(0, y - r), min(h - 1, y + r)
            x1, x2 = max(0, x - r), min(w - 1, x + r)

            # 轉成積分圖索引 (+1)
            sum_region = ( # 計算區域和
                integral[y2 + 1, x2 + 1] - integral[y1, x2 + 1]
                - integral[y2 + 1, x1] + integral[y1, x1]
            )

            # 區域面積 = window_size^2 (邊界處會小於)
            area = (y2 - y1 + 1) * (x2 - x1 + 1)
            mean_val = sum_region / area    # 計算均值
            T = mean_val                    # 閾值
            binary_image[y, x] = 255 if image[y, x] > T else 0  # 二值化

    return binary_image
```

3. 結果(window size = 5)：

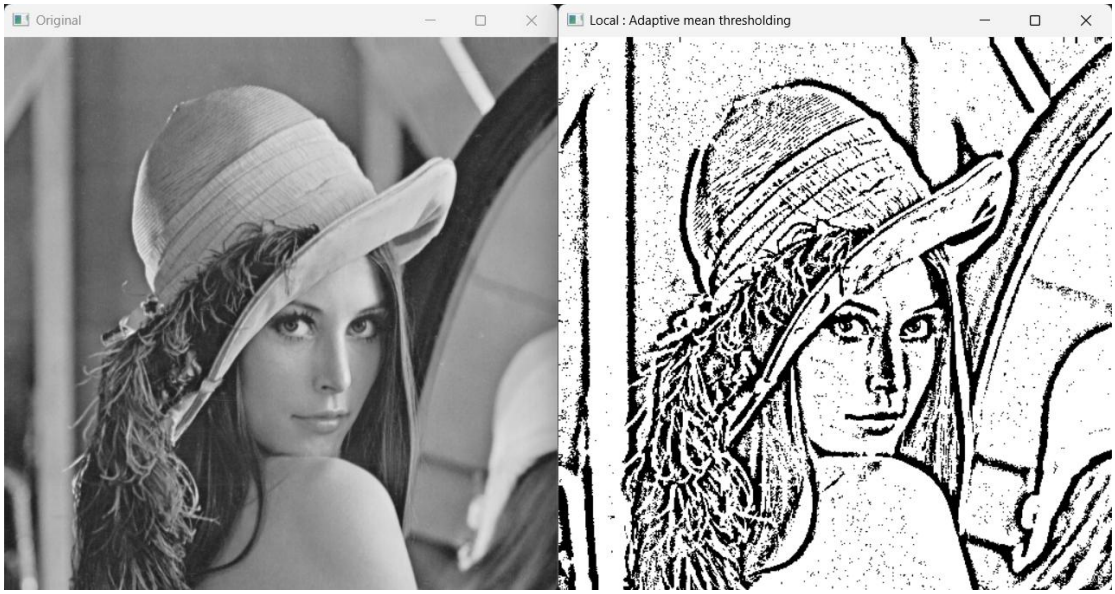


B. Adaptive mean thresholding 自適應平均閥值

單純在「Mean thresholding 平均閥值」的基礎上增加一個常數，將計算出來的局部平均值減去此常數。

```python
def local_thresholding_adaptive (image, window_size=5, c=5):
    """
    使用積分圖加速的局部均值法閾值化
    image: 灰階影像 (numpy array)
    window_size: 鄰域大小 (必須為奇數)
    c: 常數，從計算出的均值中減去的值
    """
    h, w = image.shape      # 影像高度與寬度
    r = window_size // 2    # 半徑
    binary_image = np.zeros_like(image, dtype=np.uint8) # 初始化二值影像

    # 預先計算積分圖 (integral 輸出比原圖大一圈)
    # 積分圖就是二維前綴和 (2D prefix sum)
    integral = cv2.integral(image, sdepth=cv2.CV_64F) # 使用 64 位元浮點數以防溢位

    for y in range(h):
        for x in range(w):
            # 鄰域範圍 (使用 integral 圖座標多 +1)
            y1, y2 = max(0, y - r), min(h - 1, y + r)
            x1, x2 = max(0, x - r), min(w - 1, x + r)

            # 轉成積分圖索引 (+1)
            sum_region = ( # 計算區域和
                integral[y2 + 1, x2 + 1] - integral[y1, x2 + 1]
                - integral[y2 + 1, x1] + integral[y1, x1]
            )

            # 區域面積 = window_size^2 (邊界處會小於)
            area = (y2 - y1 + 1) * (x2 - x1 + 1)
            mean_val = sum_region / area    # 計算均值
            T = mean_val - c                # 閾值
            binary_image[y, x] = 255 if image[y, x] > T else 0  # 二值化

    return binary_image
```

結果(window size = 5, c = 5)：



C. Niblack's method

公式：$T = \mu + k\sigma$，分別計算平均值 μ 與標準差 σ

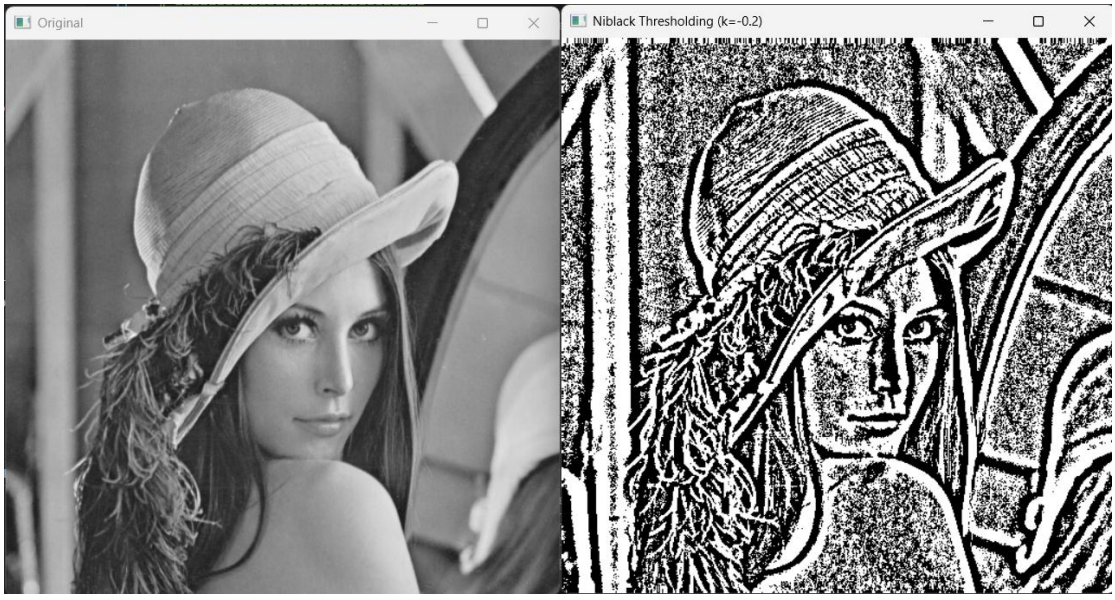
1. 平均值 $\mu = \frac{S}{area}$，S 為區域總和(從積分圖取得)，$area$ 為區域面積(中間處= window size^2，邊界處< window size^2)
2. 標準差 $\sigma = \sqrt{Var}$ ，Var 為變異數
   變異數 $Var = E[X^2] - (E[X])^2$ ，平方的平均減平均的平方
   區域平方和從平方積分圖取得

```
12  def local_thresholding_niblack(image, window_size=15, k=-0.2):
13      """
14      Niblack's local thresholding method
15      T = mean + k * std
16      image: 灰階影像 (numpy array)
17      window_size: 鄰域大小 (必須為奇數)
18      k: 常數，一般介於 [-0.5, 0.5]
19      """
20      h, w = image.shape      # 影像高度與寬度
21      r = window_size // 2    # 半徑
22      binary_image = np.zeros_like(image, dtype=np.uint8) # 初始化二值影像
23
24      # 建立積分圖與平方積分圖 (多一圈邊界)
25      integral = cv2.integral(image, sdepth=cv2.CV_64F) # 使用 64 位元浮點數以防溢位
26      integral_sq = cv2.integral(np.square(image), sdepth=cv2.CV_64F) # 平方積分圖
```

```
    for y in range(h):
        for x in range(w):
            # 區域邊界（不超出圖像）
            y1, y2 = max(0, y - r), min(h - 1, y + r)
            x1, x2 = max(0, x - r), min(w - 1, x + r)

            # 使用積分圖計算區域總和與平方和
            S  = (integral[y2 + 1, x2 + 1] - integral[y1, x2 + 1]
                - integral[y2 + 1, x1] + integral[y1, x1])
            S2 = (integral_sq[y2 + 1, x2 + 1] - integral_sq[y1, x2 + 1]
                - integral_sq[y2 + 1, x1] + integral_sq[y1, x1])
```

```

            # 使用積分圖計算區域總和與平方和
            S  = (integral[y2 + 1, x2 + 1] - integral[y1, x2 + 1]
                - integral[y2 + 1, x1] + integral[y1, x1])
            S2 = (integral_sq[y2 + 1, x2 + 1] - integral_sq[y1, x2 + 1]
                - integral_sq[y2 + 1, x1] + integral_sq[y1, x1])

            # 區域面積
            area = (y2 - y1 + 1) * (x2 - x1 + 1)

            # 均值與標準差
            mean = S / area                 # 均值
            var = (S2 / area) - (mean ** 2) # 變異數 = E[X^2] - (E[X])^2
            std = np.sqrt(max(var, 0))      # 避免浮點誤差造成負數

            # Niblack 閾值
            T = mean + k * std

            binary_image[y, x] = 255 if image[y, x] > T else 0

    return binary_image
```

3. 結果(window size = 15, k = -0.2)：

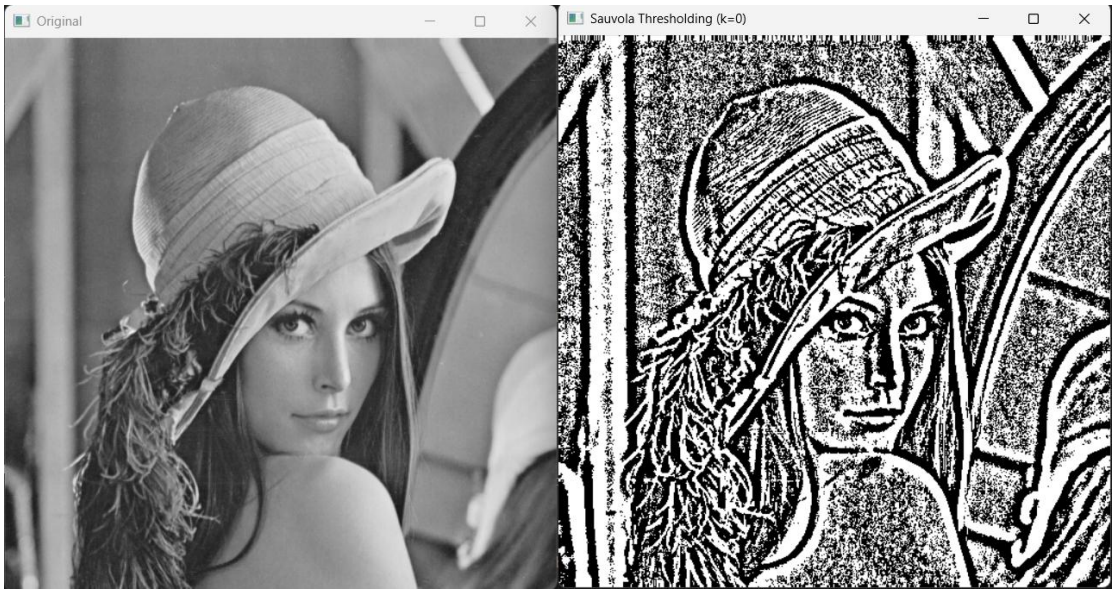


D. Sauvola's method

公式：$T = \mu\left[1 + k\left(\frac{\sigma}{R-1}\right)\right]$，比「Niblack's method」多一個參數而已

```python
def local_thresholding_sauvola(image, window_size=15, k=0.5, R=128):
    # R 是標準差的動態範圍（對於 8 位元影像通常為 128）
    h, w = image.shape      # 影像高度與寬度
    r = window_size // 2    # 半徑
    binary_image = np.zeros_like(image, dtype=np.uint8)

    # 積分圖與平方積分圖
    integral = cv2.integral(image, sdepth=cv2.CV_64F)
    integral_sq = cv2.integral(np.square(image), sdepth=cv2.CV_64F)

    for y in range(h):
        for x in range(w):
            # 區域邊界（不超出圖像）
            y1, y2 = max(0, y - r), min(h - 1, y + r)
            x1, x2 = max(0, x - r), min(w - 1, x + r)

            # 區域和與平方和
            S  = (integral[y2 + 1, x2 + 1] - integral[y1, x2 + 1]
                - integral[y2 + 1, x1] + integral[y1, x1])
            S2 = (integral_sq[y2 + 1, x2 + 1] - integral_sq[y1, x2 + 1]
                - integral_sq[y2 + 1, x1] + integral_sq[y1, x1])

            area = (y2 - y1 + 1) * (x2 - x1 + 1)    # 區域面積
            mean = S / area                         # 均值
            var = (S2 / area) - (mean ** 2)         # 變異數
            std = np.sqrt(max(var, 0))              # 標準差，避免負數

            # Sauvola 閾值公式
            T = mean * (1 + k * ((std / R) - 1))
            binary_image[y, x] = 255 if image[y, x] > T else 0

    return binary_image
```

結果(window size = 15, k = 0, R = 128)：



Global methods

A. Variance-based thresholding 基於變異數的閾值

```
6   def global_thresholding_otsu(image):
```

計算步驟：

1. 計算直方圖與機率分佈

```
13      L = 256 # 灰階級數
14      hist = cv2.calcHist([image], [0], None, [L], [0, L]).ravel()
15      hist_norm = hist / hist.sum() # 正規化直方圖，p_i
```

2. 計算累積機率 $P_1(k)$

```
44      P1 = np.cumsum(hist_norm)
45      # cumsum 是 numpy 的累積和函式，P1[k] = sum(P(i)) for i=0 to k
```

3. 計算累積平均 $m(k)$

```
48      intensity = np.arange(L) # arange 產生 0 到 255 的陣列
49      m = np.cumsum(intensity * hist_norm) # m[k] = sum(i * P(i)) for i=0 to k
```

4. 取全域平均 m_G

```
52      mG = m[-1] # mG = sum(i * P(i)) for i=0 to 255(k=L-1)
```

5. 計算類間方差 $\sigma_B^2(k)$

```
    numerator = (mG * P1 - m)**2
    denominator = P1 * (1 - P1)
    # 避免除以 0
    valid = denominator > 0          # 產生有效分母的布林陣列
    sigma_b2 = np.zeros_like(P1)     # 初始化類間方差陣列，全設為 0
    sigma_b2[valid] = numerator[valid] / denominator[valid] # 布林遮罩，只計算有效值
```

6. 找最大方差對應閾值

```
    max_val = np.max(sigma_b2)
    k_candidates = np.where(sigma_b2 == max_val)[0] # 找出所有最大值對應的 k，[0] 取出索引陣列
    k_star = int(np.mean(k_candidates))  # 若有多個最大值，取平均
```

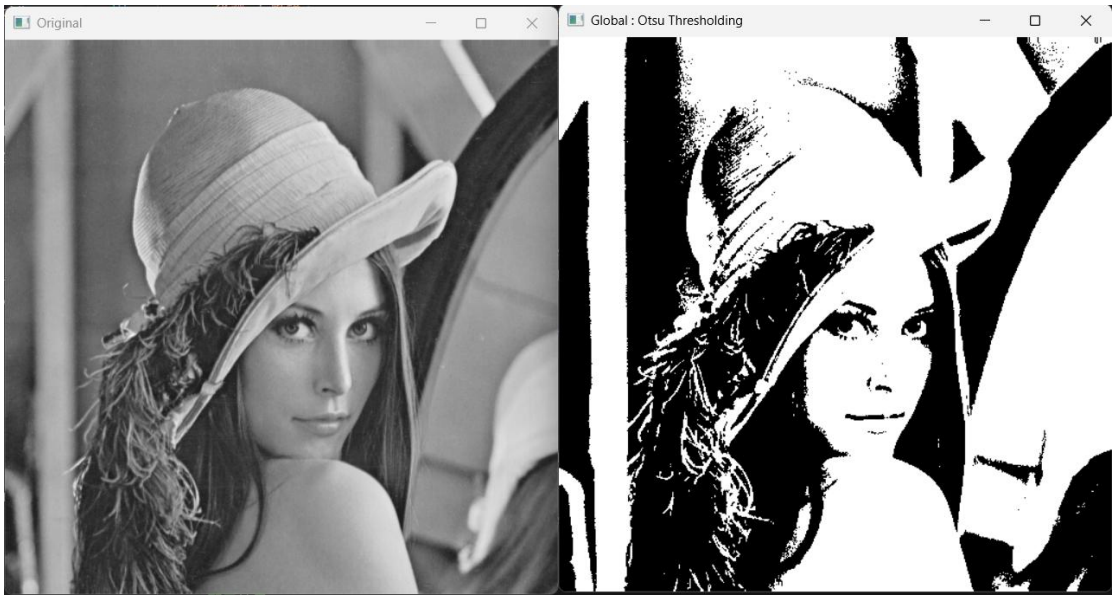
7. 計算類別分離度 η^*，越大表示分離效果越好

```
    sigma_g2 = np.sum(((intensity - mG)**2) * hist_norm)
    eta_star = sigma_b2[k_star] / sigma_g2 if sigma_g2 > 0 else 0
```

8. 產生二值圖像並回傳

```
    # 產生二值圖像
    binary = np.where(image > k_star, 255, 0).astype(np.uint8)

    return k_star, binary, eta_star
```

結果：




```
Otsu threshold = 117
Separability η* = 0.6992
```

B. Entropy-based thresholding 基於熵的閾值

程式碼：

```python
def global_thresholding_entropy(image):
    """
    使用 Entropy (最大熵法) 進行全域閾值處理
    image: 灰階影像 (numpy array)
    return：閾值、二值影像、最大熵值 H*
    """
    # 步驟 1. 計算直方圖與機率分佈
    L = 256
    hist = cv2.calcHist([image], [0], None, [L], [0, L]).ravel()
    hist_norm = hist / hist.sum()  # 正規化成機率分佈 P(i)

    # 步驟 2. 計算累積機率 (前景與背景)
    P1 = np.cumsum(hist_norm)           # 前景機率 P1(T)
    P2 = 1 - P1                         # 背景機率 P2(T)
```

```python

    # 小常數，防止 log(0) 錯誤
    eps = 1e-12

    # 步驟 3. 對每個灰階分界 T，計算前景與背景熵
    H1 = np.zeros(L)
    H2 = np.zeros(L)

    for T in range(L):
        # 前景區間 0 ~ T
        if P1[T] > 0:
            p1 = hist_norm[:T+1] / (P1[T]) # 前景的條件機率分佈，P(i|C1) = p(i)/P1(T), i in [0,T]
            H1[T] = -np.sum(p1 * np.log(p1 + eps))
        else:
            H1[T] = 0

        # 背景區間 T+1 ~ 255
        if P2[T] > 0:
            p2 = hist_norm[T+1:] / (P2[T]) # 背景的條件機率分佈，P(i|C2) = p(i)/P2(T), i in [T+1,255]
            H2[T] = -np.sum(p2 * np.log(p2 + eps))
        else:
            H2[T] = 0

    # 步驟 4. 找最大熵值
    H_total = H1 + H2
    max_val = np.max(H_total)
    k_candidates = np.where(H_total == max_val)[0]
    k_star = int(np.mean(k_candidates))  # 若多個最大值取平均

    # 步驟 5. 產生二值圖像
    binary = np.where(image > k_star, 255, 0).astype(np.uint8)

    return k_star, binary, max_val
```

解釋：

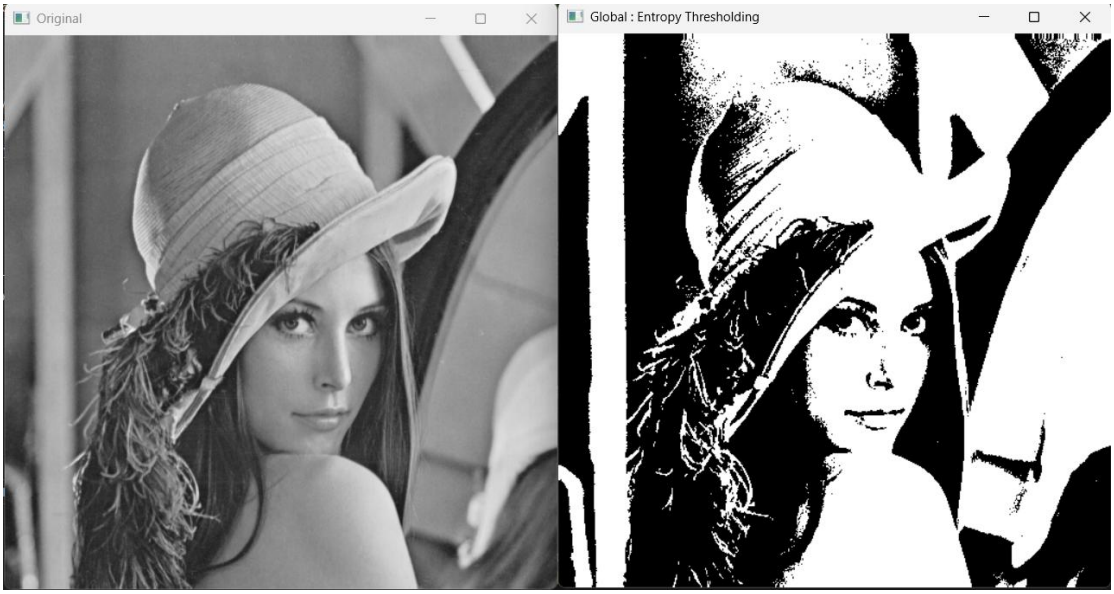
主要在第三步驟，為了在各自區域中計算熵，要對該區域的機率重新正規化，

比如前景的條件機率分布為： $P(i|C_1) = \frac{P(i)}{P_1(T)}, i = 0, 1, \dots, T$

所以前景熵為：$H_1(T) = -\sum_{i=0}^{T} P(i|C_1) \log P(i|C_1) = -\sum_{i=0}^{T} \frac{P(i)}{P_1(T)} \log \frac{P(i)}{P_1(T)}$

結果：

```
Entropy threshold = 122
Max entropy H* = 8.9413
```



C. Maximum likelihood thresholding 最大似然閾值

使用高斯分布（Gaussian distribution），比如前景的條件機率分布如下：

$$p(i|C_1) = \frac{1}{\sqrt{2\pi}\sigma_1} e^{-\frac{(i-\mu_1)^2}{2\sigma_1^2}}$$

對於每個可能的閾值 T：

1. 分成兩組：
   - 背景：灰階 $i \leq T$
   - 前景：灰階 $i > T$
2. 計算各組的平均與變異數：

$$\mu_1 = \frac{\sum_{i=0}^{T} i \cdot P(i)}{P_1(T)}, \quad \mu_2 = \frac{\sum_{i=T+1}^{L-1} i \cdot P(i)}{P_2(T)}$$

$$\sigma_1^2 = \frac{\sum_{i=0}^{T} (i-\mu_1)^2 \cdot P(i)}{P_1(T)}, \quad \sigma_2^2 = \frac{\sum_{i=T+1}^{L-1} (i-\mu_2)^2 \cdot P(i)}{P_2(T)}$$

3. 定義各類的**似然函數 (Likelihood)**：
   - 對整個類別而言，所有像素灰階出現的聯合機率：

$$L_1(T) = \prod_{i=0}^{T} p(i|C_1)^{P(i)}$$

$$L_2(T) = \prod_{i=T+1}^{L-1} p(i|C_2)^{P(i)}$$

實際運算會取對數，避免 underflow：

$$\log L(T) = \sum_{i=0}^{T} P(i) \log p(i|C_1) + \sum_{i=T+1}^{L-1} P(i) \log p(i|C_2)$$

4. **最大化聯合似然：**

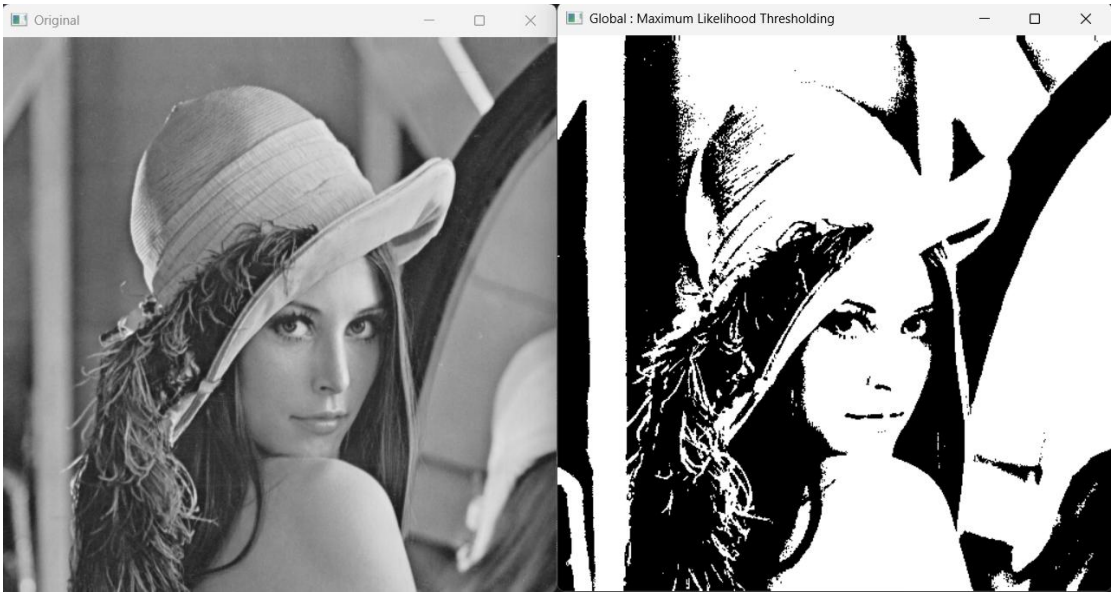
$$T^* = \arg\max_T \log L(T)$$

程式碼：

```python
def global_thresholding_mle(image):
    """
    使用 Maximum Likelihood (最大似然法) 進行全域閾值處理
    image: 灰階影像 (numpy array)
    return：閾值、二值影像、最大 log-likelihood
    """
    L = 256
    hist = cv2.calcHist([image], [0], None, [L], [0, L]).ravel()
    hist_norm = hist / hist.sum()  # 機率分佈 P(i)

    intensity = np.arange(L)
    P1 = np.cumsum(hist_norm)       # 前景累積機率
    P2 = 1 - P1                     # 背景累積機率
    eps = 1e-12                     # 防止除零與 log(0)
```

```python
    log_likelihood = np.full(L, -np.inf)

    for T in range(1, L-1):  # 0 與 255 不合理（全部屬於單一類）
        # 前景區間 0 ~ T
        if P1[T] > 0:
            mu1 = np.sum(intensity[:T+1] * hist_norm[:T+1]) / P1[T] # 前景均值 μ1
            sigma1_sq = np.sum(((intensity[:T+1] - mu1)**2) * hist_norm[:T+1]) / P1[T] # 前景變異數 σ1²
            sigma1_sq = max(sigma1_sq, eps)  # 防止除零
            p1 = (1 / np.sqrt(2 * np.pi * sigma1_sq)) * np.exp(-0.5 * ((intensity[:T+1] - mu1)**2) / sigma1_sq) # 代入高斯分布公式
            log_L1 = np.sum(hist_norm[:T+1] * np.log(p1 + eps)) # log domain 避免 underflow
        else:
            log_L1 = -np.inf # 無效值

        # 背景區間 T+1 ~ 255
        if P2[T] > 0:
            mu2 = np.sum(intensity[T+1:] * hist_norm[T+1:]) / P2[T] # 背景均值 μ2
            sigma2_sq = np.sum(((intensity[T+1:] - mu2)**2) * hist_norm[T+1:]) / P2[T] # 背景變異數 σ2²
            sigma2_sq = max(sigma2_sq, eps)
            p2 = (1 / np.sqrt(2 * np.pi * sigma2_sq)) * np.exp(-0.5 * ((intensity[T+1:] - mu2)**2) / sigma2_sq)
            log_L2 = np.sum(hist_norm[T+1:] * np.log(p2 + eps))
        else:
            log_L2 = -np.inf # 無效值

        # 聯合似然 (log domain)
        log_likelihood[T] = log_L1 + log_L2

    # 找最大似然對應的閾值
    max_val = np.max(log_likelihood)
    k_candidates = np.where(log_likelihood == max_val)[0]
    k_star = int(np.round(np.mean(k_candidates)))

    # 產生二值圖像
    binary = np.where(image > k_star, 255, 0).astype(np.uint8)

    return k_star, binary, max_val
```

結果：




```
ML threshold = 114
Max log-likelihood = -4.684070513470
```

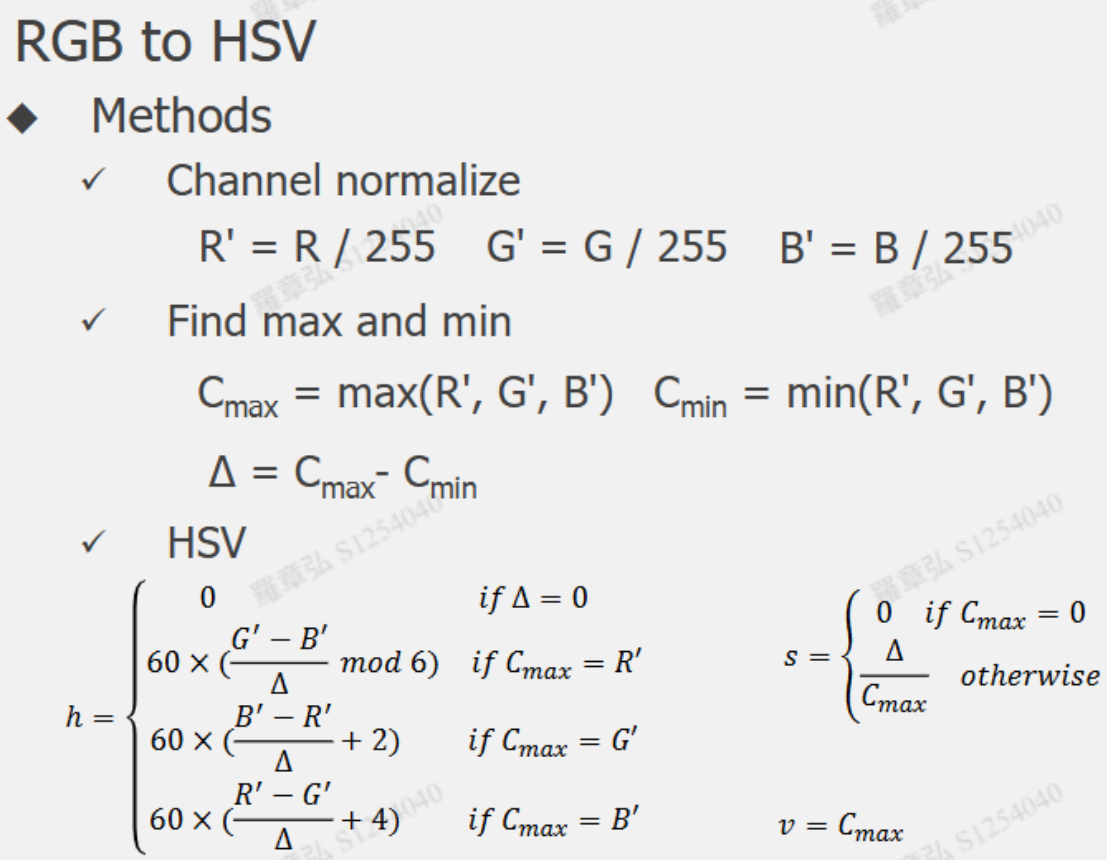
Noise reduction in color image

A. RGB to HSV

RGB -> 轉成 HSV -> 對 V 分量去噪(選擇) -> 轉回 RGB

1. 將 RGB 轉成 HSV

   依講義公式：

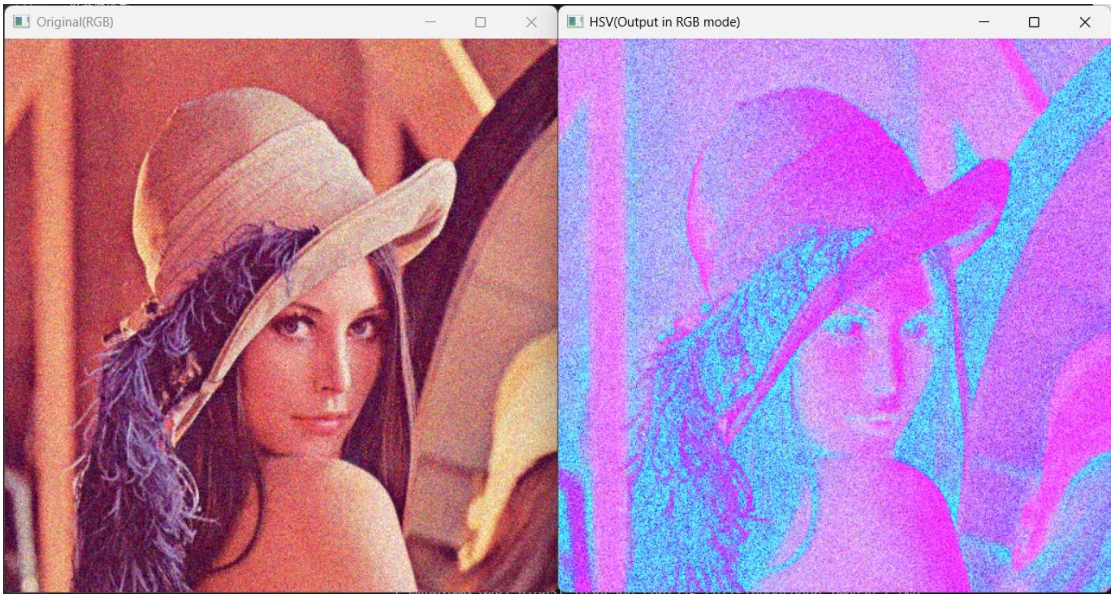


程式碼：

```
6   def rgb_to_hsv_manual(img_bgr):
7       """手動實作 RGB -> HSV 轉換"""
8       img = img_bgr.astype(np.float32) / 255.0  # normalize
9       B, G, R = cv2.split(img)
10
11      Cmax = np.maximum(np.maximum(R, G), B)
12      Cmin = np.minimum(np.minimum(R, G), B)
13      delta = Cmax - Cmin
14
15      # Hue
16      H = np.zeros_like(Cmax)
```

```
    mask = delta != 0
    # when Cmax == R
    idx = (Cmax == R) & mask
    H[idx] = 60 * (((G[idx] - B[idx]) / delta[idx]) % 6)
    # when Cmax == G
    idx = (Cmax == G) & mask
    H[idx] = 60 * (((B[idx] - R[idx]) / delta[idx]) + 2)
    # when Cmax == B
    idx = (Cmax == B) & mask
    H[idx] = 60 * (((R[idx] - G[idx]) / delta[idx]) + 4)
    H[~mask] = 0

    # Saturation
    S = np.zeros_like(Cmax)
    S[Cmax != 0] = delta[Cmax != 0] / Cmax[Cmax != 0]

    # Value
    V = Cmax

    hsv = cv2.merge([H, S, V])
    return hsv
```

結果(HSV 圖以 RGB 方式輸出)：



2. 選擇對 V 分量去噪方式
   a. 無濾波
      V 分量保持不變

b. 中值濾波（Median Filter）

概念：

- 不是用平均值，而是取鄰域中所有像素灰階的「中位數（median）」作為新的像素值。
- 對突發型雜訊（salt-and-pepper noise，黑白點狀雜訊）特別有效。

數學表示：

$$I'(x,y) = \text{median}\{I(i,j) \mid (i,j) \in N(x,y)\}$$

其中 $N(x,y)$ 為以 (x,y) 為中心的 $k \times k$ 鄰域。

特性：

- 能有效去除離群像素。
- 不會造成邊緣過度模糊（邊緣仍可保留）。
- 適合用於「椒鹽雜訊」。

OpenCV 程式：

```
v_denoised = cv2.medianBlur(v, 5)
```

代表以 5×5 視窗 取中位數。

c. 高斯濾波（Gaussian Filter）

概念：

- 將鄰近像素加權平均，距離中心越近、權重越高。
- 權重服從高斯分佈（常態分佈），因此平滑效果自然、不會產生明顯邊界。

數學表示：

$$I'(x,y) = \sum_{i,j} G(i,j) \cdot I(x+i, y+j)$$

$$G(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

特性：

- 適合消除高斯分佈雜訊（自然相機影像常見）。
- 平滑整體影像，但可能使邊緣模糊。
- 在影像處理中常用作「預處理」步驟。

OpenCV 程式：

```
v_denoised = cv2.GaussianBlur(v, (5, 5), 0)
```

(5,5) 是核大小，0 代表自動根據核大小計算 σ。

d. 雙邊濾波（Bilateral Filter）

概念：

- 結合空間距離與像素相似度兩種權重：
  - 距離近 -> 權重大
  - 顏色差小 -> 權重大
- 因此在平滑雜訊的同時能保留邊緣細節。

數學表示：

$$I'(x,y) = \frac{1}{W_p} \sum_{i,j} e^{-\frac{(x-i)^2+(y-j)^2}{2\sigma_s^2}} \cdot e^{-\frac{(I(x,y)-I(i,j))^2}{2\sigma_r^2}} \cdot I(i,j)$$

- σ_s：控制空間距離的權重
- σ_r：控制灰階差異的權重
- W_p：歸一化因子

特性：

- 對邊緣敏感，能有效保留邊界。
- 處理速度較慢（計算量大）。
- 適合自然照片或人臉影像去噪。

OpenCV 程式：

```
v_denoised = cv2.bilateralFilter(v, 9, 75, 75)
```

9 為鄰域直徑，75,75 控制空間與顏色域的 σ 值。

3. 將 HSV 轉回 RGB，依講義公式

依講義公式：

HSV to RGB

- Methods
  - ✓ Calculate the interval where the hue lies

    $C = v \times s$　　$H' = H / 60$　　$X = C \times (1-|(H' \bmod 2) -1|)$
  - ✓ Calculate the middle value of RGB

$$(R_1, G_1, B_1) = \begin{cases} (C, X, 0) & if\ 0 \le H' < 1 \\ (X, C, 0) & if\ 1 \le H' < 2 \\ (0, C, X) & if\ 2 \le H' < 3 \\ (0, X, C) & if\ 3 \le H' < 4 \\ (X, 0, C) & if\ 4 \le H' < 5 \\ (C, 0, X) & if\ 5 \le H' < 6 \end{cases}$$

  - ✓ Final RGB value

$m = v - c$

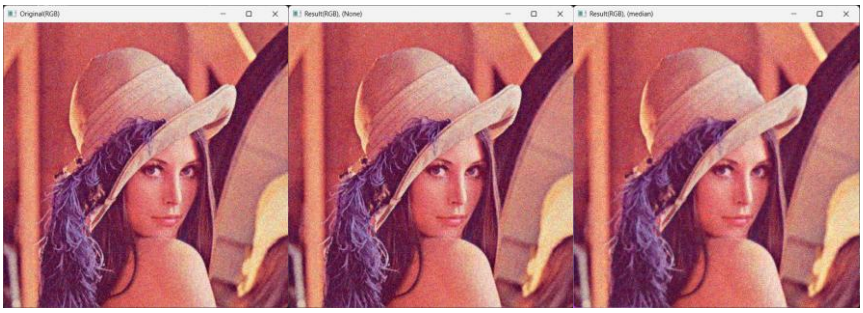
$(R, G, B) = (R_1+m, G_1+m, B_1+m)$

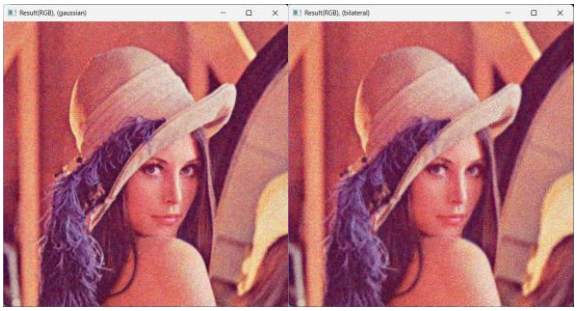
程式碼：

```
40    def hsv_to_rgb_manual(hsv):
41        """手動實作 HSV -> RGB 轉換"""
42        H, S, V = cv2.split(hsv)
43
44        C = V * S
45        H_ = H / 60.0
46        X = C * (1 - np.abs((H_ % 2) - 1))
47        m = V - C
48
49        # 建立空白輸出
50        R1 = np.zeros_like(H)
51        G1 = np.zeros_like(H)
52        B1 = np.zeros_like(H)
```

```
    # 根據區段條件給值
    conds = [
        (0 <= H_) & (H_ < 1),
        (1 <= H_) & (H_ < 2),
        (2 <= H_) & (H_ < 3),
        (3 <= H_) & (H_ < 4),
        (4 <= H_) & (H_ < 5),
        (5 <= H_) & (H_ < 6),
    ]
    Z = np.zeros_like(H) # 零陣列
    rgb_values = [
        (C, X, Z),
        (X, C, Z),
        (Z, C, X),
        (Z, X, C),
        (X, Z, C),
        (C, Z, X),
    ]
    for cond, (r, g, b) in zip(conds, rgb_values):
        R1[cond], G1[cond], B1[cond] = r[cond], g[cond], b[cond]

    # 加上 m 並轉回 0~255
    R = (R1 + m) * 255
    G = (G1 + m) * 255
    B = (B1 + m) * 255

    rgb = cv2.merge([B, G, R]).astype(np.uint8)
    return rgb
```

結果(原版、無濾波與三種濾波)：





B. Joint bilateral filtering 聯合雙邊濾波

使用灰階圖片 lena.bmp 作為導引影像(guide image)

公式：

- Spatial weight 空間權重 $\omega_s(p,q)$：由像素之間的歐氏距離決定

$$\omega_s(p,q) = \exp\left(-\frac{\|p-q\|^2}{2\sigma_s^2}\right)$$

- Intensity weight 強度權重 $\omega_r\big(G(p),G(q)\big)$：由像素之間的差異決定

$$\omega_r\big(G(p),G(q)\big) = \exp\left(-\frac{\big(G(p)-G(q)\big)^2}{2\sigma_r^2}\right)$$

- Joint weight 聯合權重：由空間權重和強度權重的乘積決定

$$\omega(p,q) = \omega_s(p,q) \times \omega_r\big(G(p),G(q)\big)$$

- Filtered pixel 濾波像素：由鄰域內所有像素的加權平均值決定

$$J(p) = \frac{\sum_{q\in N_p}\omega(p,q)\times I(p)}{\sum_{q\in N_p}\omega(p,q)}$$

- 空間權重標準差 σ_s 與強度權重標準差 σ_r 自訂，亦可自行計算

程式碼：

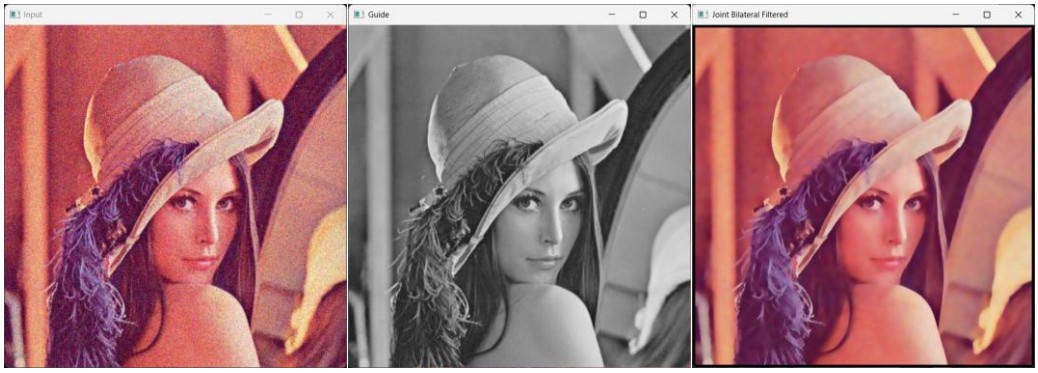
```
10  def joint_bilateral_filter(I, G, diameter=9, sigma_s=12, sigma_r=25):
11      """
12      實作：Joint Bilateral Filtering（聯合雙邊濾波）
13      Args:
14          I : ndarray     要被平滑處理的影像（彩色圖像）
15          G : ndarray     導引影像（guide image，灰階圖像）
16          diameter : int  濾波視窗大小（必須為奇數，如 3、5、7...）
17          sigma_s : float 空間權重標準差（σ_s），控制距離越遠權重衰減的速度
18          sigma_r : float 亮度差權重標準差（σ_r），控制顏色差距對權重的影響程度
19      Returns：經過 Joint Bilateral Filter 後的影像
20      """
21      # ---- Step 1: 資料型別處理 ----
22      # 若輸入是彩色圖像，保留三通道；若是灰階，擴維成三維方便運算
23      if I.ndim == 3:
24          I = I.astype(np.float32)
25      else:
```

```
        I = I[..., np.newaxis].astype(np.float32)

    # 導引影像轉成 float32，避免溢位
    G = G.astype(np.float32)

    # 視窗半徑與圖像長寬
    half = diameter // 2
    h, w = G.shape

    # 輸出影像初始化 (與輸入 I 同大小)
    result = np.zeros_like(I)
```

```
    # ---- Step 2: 建立空間權重 (spatial weight) ω_s(p, q) ----
    # 根據像素距離的高斯函數
    # ω_s(p,q) = exp(-||p-q||^2 / (2σ_s^2))
    x, y = np.meshgrid(np.arange(-half, half + 1), np.arange(-half, half + 1))
    spatial_weight = np.exp(-(x**2 + y**2) / (2 * sigma_s**2))

    # ---- Step 3: 對每一個像素進行濾波 ----
    # p 為中心像素座標
    for i in range(half, h - half):
        for j in range(half, w - half):

            # ---- Step 3.1: 擷取鄰域像素 ----
            # N_p 表示以 (i,j) 為中心的鄰域
            patch_I = I[i - half:i + half + 1, j - half:j + half + 1, :]  # 要濾波的彩色區塊
            patch_G = G[i - half:i + half + 1, j - half:j + half + 1]    # 導引影像的對應灰階區塊

            # ---- Step 3.2: 計算亮度權重 (range weight) ω_r ----
            # ω_r(G(p), G(q)) = exp(-( (G(p) - G(q))^2 / (2σ_r^2) ))
            diff = patch_G - G[i, j]
            range_weight = np.exp(-(diff**2) / (2 * sigma_r**2))

            # ---- Step 3.3: 計算聯合權重 (joint weight) ----
            # ω(p,q) = ω_s(p,q) × ω_r(G(p), G(q))
            weight = spatial_weight * range_weight

            # 擴張維度成 (k,k,1)，讓彩色三通道能同時乘上相同權重
            weight = weight[..., np.newaxis]
```

```
            # ---- Step 3.4: 計算加權平均 ----
            # J(p) = Σ ω(p,q) I(q) / Σ ω(p,q)
            numerator = np.sum(weight * patch_I, axis=(0, 1))  # 加權像素和
            denominator = np.sum(weight, axis=(0, 1))          # 總權重
            result[i, j, :] = numerator / denominator          # 加權平均值

    # ---- Step 4: 處理單通道情況 ----
    if result.shape[2] == 1:
        result = result.squeeze()

    # ---- Step 5: 回傳結果 ----
    return np.clip(result, 0, 255).astype(np.uint8)
```

原圖、導引圖與結果(diameter=9, sigma_s=12, sigma_r=25)：



C. Vector median filtering 向量中值濾波

- 每個像素是一個 RGB 向量 $v_i = (R_i, G_i, B_i)$
- 計算該向量與鄰域所有向量的距離總和 $D(v_i) = \sum_j d(v_i, v_j)$
- 取距離總和最小的那個像素作為輸出值
- 這種方法能有效處理彩色影像的脈衝雜訊，比單通道 median filter 更保留色彩一致性。
- 此實作為最直接版本，時間複雜度 $O(N \times k^4)$，當視窗很大時計算會變慢。可改用 KD-Tree、快速排序或近似方法。

程式碼：

```
7   def vector_median_filter(img, diameter=5):
8       """
9       實作：Vector Median Filtering
10      Args:
11          img : ndarray  彩色影像 (H, W, 3)
12          diameter : int  鄰域視窗大小 (奇數)
13      Returns:
14          result : ndarray 濾波後的影像
15      """
16      # 確保是 float32，方便計算
17      img = img.astype(np.float32)
18      half = diameter // 2
19      h, w, c = img.shape
20
21      # 輸出影像初始化
22      result = np.zeros_like(img)
23
24      # 對每個像素進行濾波
25      for i in range(half, h - half):
26          for j in range(half, w - half):
27
28              # Step 1: 擷取鄰域
29              patch = img[i - half:i + half + 1, j - half:j + half + 1, :]
30              pixels = patch.reshape(-1, 3)  # (k^2, 3) 將鄰域像素展平為向量集合。
```

```
            # Step 2: 計算兩兩間歐氏距離總和
            # d(v_i, v_j) = sqrt( (R_i-R_j)^2 + (G_i-G_j)^2 + (B_i-B_j)^2 )
            # D(v_i) = sum_j d(v_i, v_j)
            # 使用 broadcasting 加速，diffs 用廣播計算所有向量之間的距離。
            diffs = pixels[:, np.newaxis, :] - pixels[np.newaxis, :, :]  # shape (N, N, 3)
            dist = np.sqrt(np.sum(diffs ** 2, axis=2))  # shape (N, N)
            sum_dist = np.sum(dist, axis=1)  # shape (N,)，sum_dist：每一個向量與所有其他向量的距離總和。

            # Step 3: 找出距離總和最小的像素
            min_idx = np.argmin(sum_dist) # 找出最靠近中位值的向量。
            median_vector = pixels[min_idx]

            # Step 4: 設定輸出像素
            result[i, j, :] = median_vector

    return np.clip(result, 0, 255).astype(np.uint8)
```

原圖與結果：