Department of Electronic and Telecommunications Engineering
University of Moratuwa

EN3160
Image Processing and Machine Vision

Assignment 1 on Intensity Transformations and Neighborhood Filtering

Rathnayake R M T N B
220528X
https://github.com/TNRathnayake/EN3160-Assignment1

[0.8cm] **August 12, 2025**

Question 1: Intensity Transformation

A piecewise intensity mapping (Fig. 1a) was implemented as a 256-entry LUT and applied to the grayscale image in Fig. 1b. Levels 0–50 were unchanged, 51–150 were boosted linearly to 100–255, a jump was set at 150, and 151–255 were mapped linearly from 150 to 255.

```python
def build_q1_lut():
    xs = np.arange(256, dtype=np.float32)
    y = np.zeros_like(xs)
    y[0:50] = xs[0:50]
    y[50:150] = 100 + 1.55*(xs[50:150] - 50)
    y[150] = 150
    y[151:256] = xs[151:256]
    return np.clip(y, 0, 255).astype(np.uint8)

lut = build_q1_lut()
out = cv2.LUT(img, lut)
```

Listing 1: Q1: LUT construction and application

(a) Transformation function (LUT).

(b) Original (left) and transformed (right).

Figure 1: Q1: Intensity mapping and results.

Discussion: The midtone ramp ($\approx 1.55\times$ gain) enhances facial texture contrast while leaving shadows and highlights mostly unchanged. The jump at level 150 prevents over-brightening of the brightest tones but introduces slight *banding* in smooth gradients. A continuous curve could reduce such artifacts.

Question 2: Accentuating White/Gray Matter

A Gaussian pulse was used as an intensity transform to accentuate specific tissue bands in the proton-density slice. This provides a smooth (continuous) mapping, avoiding the banding artifacts of discontinuous piecewise curves.

For input intensity x, the LUT is

$$f(x) = b + A \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right),$$

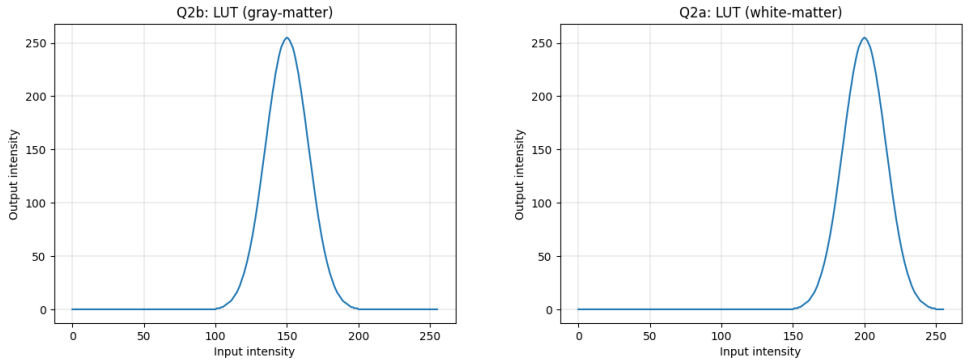
where μ centers the passband and σ controls selectivity. Parameters used: μ_{GM}=150, σ=15 (gray matter) and μ_{WM}=200, σ=15 (white matter).

```python
def gaussian_lut(mu, sigma, A=255, b=0):
    x = np.arange(256, dtype=np.float32)
    y = b + A * np.exp(-((x - mu)**2) / (2.0 * sigma**2))
    return np.clip(y, 0, 255).astype(np.uint8)

brain  = cv2.imread("a1images/brain_proton_density_slice.png", cv2.IMREAD_GRAYSCALE)
gm_lut = gaussian_lut(mu=150, sigma=15)   # gray matter
wm_lut = gaussian_lut(mu=200, sigma=15)   # white matter
gm_img = cv2.LUT(brain, gm_lut)
wm_img = cv2.LUT(brain, wm_lut)
```

Listing 2: Q2: Gaussian LUT and application (essential lines)

(a) Original (b) Gray matter (c) White matter

Figure 2: Q2 results: Gaussian band-pass LUTs applied to the brain slice.



(a) LUT for gray-matter (μ=150, σ=15). (b) LUT for white-matter (μ=200, σ=15).

Figure 3: Q2: Intensity transformation plots (required).

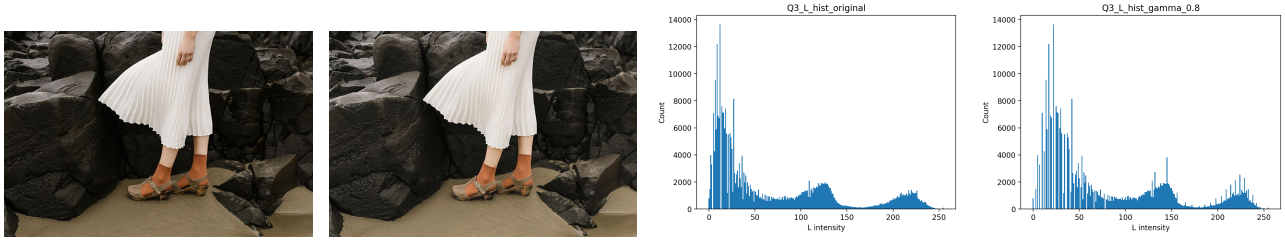
Discussion: With μ placed near the target tissue's intensity mode, in-band structures are brightened while others are suppressed. A smaller σ yields tighter selectivity but can dim non-target areas more aggressively. Noise is not removed by this point-wise transform; denoising would require neighborhood filtering. (If a monotonic look is preferred, a multiplicative bump $f(x) = x(1 + aG)$ can be used instead.)

Question 3: Gamma Correction on the L* Channel

Gamma correction was applied to the L* (lightness) channel in L*a*b* space to brighten shadows while preserving highlights. A value of $\gamma = 0.8$ was selected after visual tuning.

```
lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
L, a, b = cv2.split(lab)
gamma = 0.8
# LUT or direct power; here: direct
Lf = L.astype(np.float32) / 255.0
L_corr = np.clip((Lf ** gamma) * 255.0, 0, 255).astype(np.uint8)
img_corr = cv2.cvtColor(cv2.merge([L_corr, a, b]), cv2.COLOR_LAB2BGR)
```

Listing 3: Q3: Gamma on L* (essential lines)



(a) Original (b) Gamma ($\gamma = 0.8$) (c) L* hist (orig.) (d) L* hist (after)

Figure 4: Q3: Gamma on L* — visuals and histograms in one figure.

Discussion: Because $\gamma < 1$, darker L* values are expanded, brightening shadows while minimally affecting highlights. The non-linear mapping redistributes discrete intensities and can introduce slight quantization noise in histograms; however, perceptually it increases overall brightness without color shifts (chroma preserved in a^*, b^*).

Question 4: Vibrance via Saturation-Space Intensity Transform

The image was converted to HSV and only the *Saturation* plane was transformed using

$$f(x) = \min\Big\{ x + a \cdot 128 \exp\big(-\tfrac{(x-128)^2}{2\sigma^2}\big),\ 255\Big\}, \quad a \in [0,1],\ \sigma = 70.$$

A value of $a = 0.65$ was selected visually to increase colourfulness in mid-saturation regions while avoiding over-saturation; σ was fixed at 70. The modified S plane was recombined with H and V to obtain the final image.

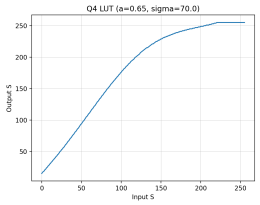


Figure 5: Q4: S-plane transform).

```python
# Split to HSV
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
H, S, V = cv2.split(hsv)

# Vibrance LUT: x + a * 128 * exp(-((x-128)^2)/(2*sigma^2))
def vibrance_lut(a=0.65, sigma=70.0):
    x = np.arange(256, dtype=np.float32)
    y = x + a * (128.0 * np.exp(-((x - 128.0)**2) / (2.0 * sigma**2)))
    return np.clip(y, 0, 255).astype(np.uint8)

lut = vibrance_lut(a=0.65, sigma=70.0)
S2  = cv2.LUT(S, lut)                                  # apply only on S
out = cv2.cvtColor(cv2.merge([H, S2, V]), cv2.COLOR_HSV2BGR)  # recombine
```

Listing 4: Q4: Vibrance on saturation (essential lines)

Figure 6: Q4(a): Hue, Saturation, and Value planes (displayed in grayscale).

(a) Original.

(b) Vibrance enhanced ($a = 0.65$).

Figure 7: Q4(b–e): Original & result

Discussion: The Gaussian bump centred at x=128 selectively increases mid-range saturation (vibrance) while leaving very low/high S values nearly unchanged; the $\min(\cdot, 255)$ cap prevents clipping. This enhances colourful regions without pushing already saturated areas into artefacts.

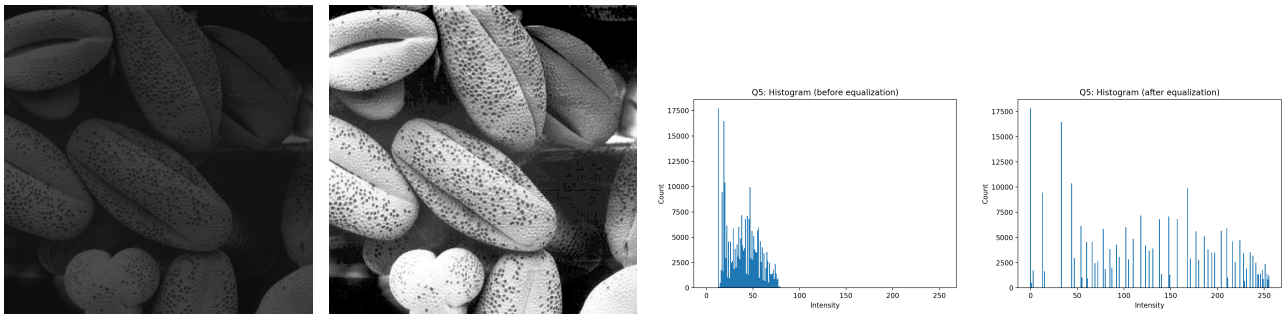
Question 5: Histogram Equalization

A custom histogram equalization was implemented by computing the histogram, its cumulative distribution (CDF), and forming a 256-entry LUT via the standard $\text{cdf}_{\min}$ formula. The method redistributes intensities to

spread the histogram and improve contrast. (If the source was a 16-bit `.tif`, it was converted to 8-bit prior to equalization.)

```python
def hist_equalize_gray(img: np.ndarray):
    """Equalize a uint8 grayscale image using CDF-based LUT."""
    if img.ndim != 2 or img.dtype != np.uint8:
        raise ValueError("Provide a 2D uint8 grayscale image")

    hist = np.bincount(img.ravel(), minlength=256).astype(np.int64)
    cdf  = hist.cumsum()
    nz   = np.flatnonzero(cdf)
    if nz.size == 0:  # empty
        return img.copy(), hist, hist.copy(), np.arange(256, dtype=np.uint8)

    cdf_min = cdf[nz[0]]; total = cdf[-1]
    if total == cdf_min:  # flat image
        return img.copy(), hist, hist.copy(), np.arange(256, dtype=np.uint8)

    lut = np.round((cdf - cdf_min) / (total - cdf_min) * 255.0)
    lut = np.clip(lut, 0, 255).astype(np.uint8)
    out = lut[img]
    hist_after = np.bincount(out.ravel(), minlength=256).astype(np.int64)
    return out, hist, hist_after, lut

# Example usage:
# orig = cv2.imread("a1images/shells.tif", cv2.IMREAD_GRAYSCALE)
# eq, h_before, h_after, lut = hist_equalize_gray(orig)
```

Listing 5: Q5: Custom histogram equalization (essential lines)



(a) Original (b) Equalized (c) Hist (before) (d) Hist (after)

Figure 8: Q5: Custom equalization — images and histograms (one-row layout).

Discussion. The equalized image shows increased global contrast, with the histogram spread more uniformly. Minor quantization noise can appear due to the discrete LUT mapping, but overall visibility of midtones and details is improved.

Question 6: Foreground-Only Histogram Equalization

The image was converted to HSV; the *Saturation* plane was thresholded (Otsu) to obtain a binary foreground mask. Histogram equalization was then applied *only* to the foreground intensities of the V plane using the CDF-based formula, and the equalized foreground was recombined with the original background.

```python
# Split and mask (S-plane, Otsu)
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
H, S, V = cv2.split(hsv)
_, mask = cv2.threshold(S, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN,  np.ones((3,3),np.uint8), iterations=1)
mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, np.ones((3,3),np.uint8), iterations=1)

# FG histogram and CDF on V
hist_fg = cv2.calcHist([V],[0],mask,[256],[0,256]).ravel()
cdf     = np.cumsum(hist_fg); cdf_min = cdf[cdf>0][0]; N = cdf[-1]
lut     = np.round((cdf - cdf_min) / max(1,(N - cdf_min)) * 255.0).clip(0,255).astype(np.uint8)

# Apply to V, compose FG(eq) + BG(orig), and merge back
V_eq_all = cv2.LUT(V, lut)
fg_eq    = cv2.bitwise_and(V_eq_all, V_eq_all, mask=mask)            # equalized FG
bg_orig  = cv2.bitwise_and(V,        V,        mask=cv2.bitwise_not(mask))
V_out    = cv2.add(fg_eq, bg_orig)
result   = cv2.cvtColor(cv2.merge([H, S, V_out]), cv2.COLOR_HSV2BGR)
```

Listing 6: Q6: Foreground-only equalization on V (essential lines)

(a) Hue, Saturation, and Value planes (grayscale).

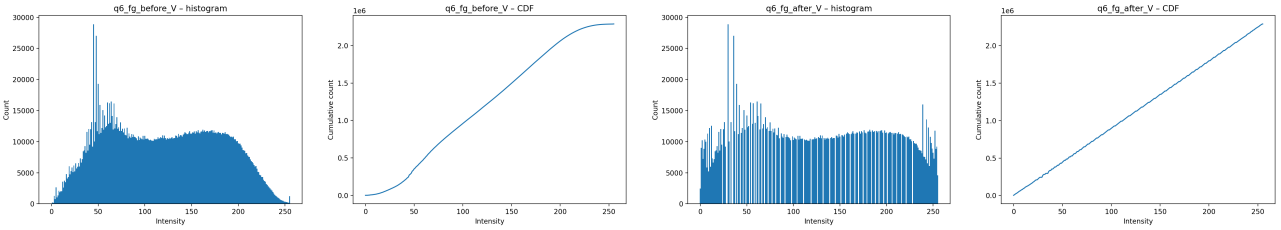
(b) Binary foreground mask from S

Figure 9: Q6(a–b): HSV split and foreground mask

(a) Original image.

(b) Foreground-equalized result.

Figure 10: Q6(f): Result after equalizing the foreground (background unchanged).



(a) FG hist (before) (b) FG CDF (before) (c) FG hist (after) (d) FG CDF (after)

Figure 11: Q6(c–e): Foreground-only histogram and CDF before/after equalization.

Discussion. Thresholding the S plane isolates colourful regions well; the FG-only CDF mapping spreads the foreground intensities on V, increasing contrast there while leaving the background intact. Slight edge transitions may appear at mask boundaries; these can be reduced with gentle mask feathering, though a hard recomposition was used to match the specification.

Question 7: Sobel Gradient (filter2D, custom, separable)

The horizontal/vertical Sobel gradients were computed with (a) `cv2.filter2D`, (b) a custom 2D convolver, and (c) a separable implementation using $K_x = \begin{bmatrix}1\\2\\1\end{bmatrix}\ [1\ 0\ -1]$ and $K_y = K_x^\top$. For visualization of signed responses, results were linearly mapped to $[0, 255]$ with zero $\mapsto$ 128.

```
# Sobel kernels
Kx = np.array([[1,0,-1],[2,0,-2],[1,0,-1]], np.float32)
Ky = Kx.T

# (a) OpenCV filter2D
gx_a = cv2.filter2D(img, cv2.CV_32F, Kx, borderType=cv2.BORDER_REFLECT)
gy_a = cv2.filter2D(img, cv2.CV_32F, Ky, borderType=cv2.BORDER_REFLECT)

# (b) Custom 3x3 convolution
def conv2d(img, k):
    pad = cv2.copyMakeBorder(img,1,1,1,1,cv2.BORDER_REFLECT)
    out = np.zeros_like(img, np.float32)
    for y in range(img.shape[0]):
```

```
        for x in range(img.shape[1]):
            out[y,x] = np.sum(pad[y:y+3, x:x+3].astype(np.float32) * k)
    return out
gx_b, gy_b = conv2d(img,Kx), conv2d(img,Ky)

# (c) Separable: Kx = a @ b, Ky = Kx^T
a = np.array([1,2,1], np.float32).reshape(-1,1)   # 3x1 smooth
b = np.array([1,0,-1], np.float32).reshape(1,-1)  # 1x3 derivative
dx = cv2.filter2D(img, cv2.CV_32F, b,   borderType=cv2.BORDER_REFLECT)  # d/dx
gx = cv2.filter2D(dx,  cv2.CV_32F, a,   borderType=cv2.BORDER_REFLECT)  # smooth_y
dy = cv2.filter2D(img, cv2.CV_32F, a,   borderType=cv2.BORDER_REFLECT)  # smooth_x
gy = cv2.filter2D(dy,  cv2.CV_32F, b.T, borderType=cv2.BORDER_REFLECT)  # d/dy

# (display: map signed float to 0..255 with 0 -> 128)
def to_u8_signed(g):
    m = float(np.max(np.abs(g))) + 1e-9
    return np.clip((g/(2*m) + 0.5)*255.0, 0, 255).astype(np.uint8)
```

Listing 7: Q7: Sobel via filter2D, custom 2D conv, and separable

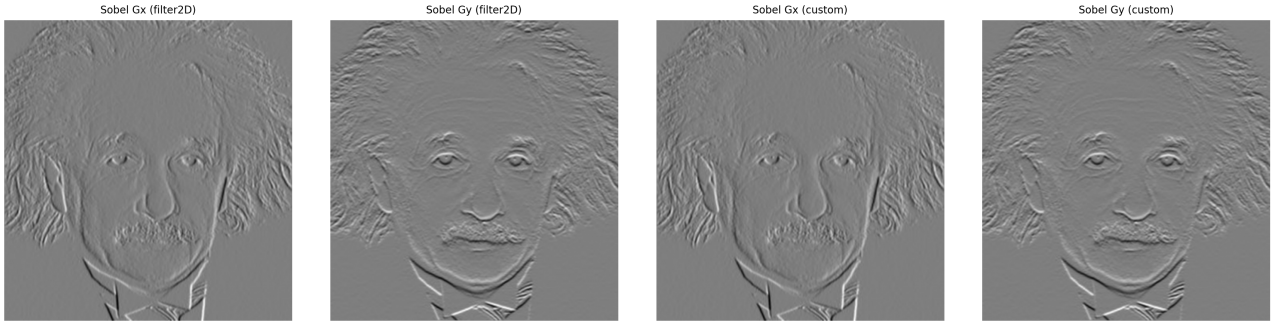


Figure 12: Q7(a,b): Sobel G_x, G_y via OpenCV `filter2D` and custom convolution.

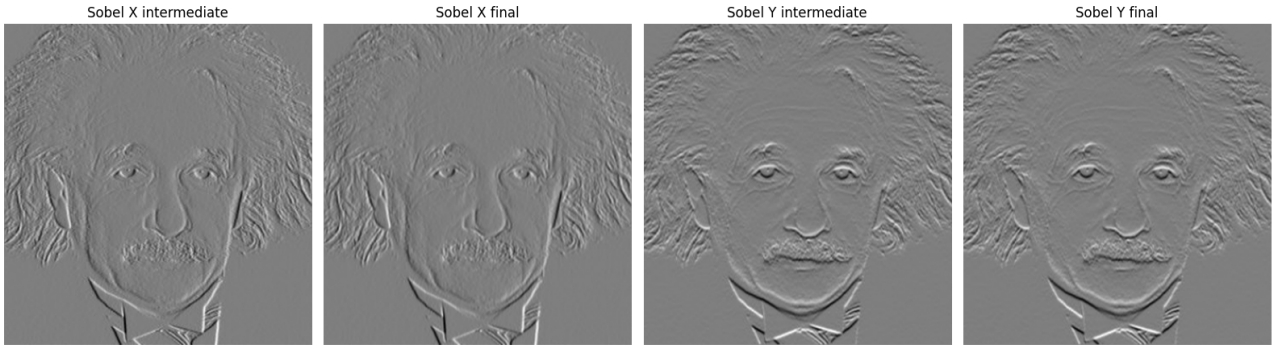


Figure 13: Q7(c): Separable Sobel—intermediate $(d/dx, d/dy)$ and final (G_x, G_y) results.

Discussion. All three implementations produce equivalent gradients (up to numeric rounding). The separable form reduces computation (two 1D filters per axis) while yielding the same Sobel response as full 3×3 convolution.

Question 8: Image zoom (NN vs. bilinear) and normalized SSD

Method. Two zoomers were implemented: nearest–neighbour and bilinear. For each small/large pair, the small image was upscaled to the *exact* size of the large image and compared using the normalized SSD:

$$\mathrm{nSSD}(A, B) = \frac{\sum (A - B)^2}{\sum B^2}.$$

(Only $\approx 4\times$ pairs were included as per the question.)

```
def zoom_nn_to(img, H, W):
    h, w = img.shape[:2]; sy, sx = H/h, W/w
    y2 = np.arange(H); x2 = np.arange(W)
    y  = np.clip(np.round((y2+0.5)/sy - 0.5).astype(int), 0, h-1)
    x  = np.clip(np.round((x2+0.5)/sx - 0.5).astype(int), 0, w-1)
    return img[y[:,None], x[None,:]] if img.ndim==2 else img[y[:,None], x[None,:], :]

def zoom_bilinear_to(img, H, W):
    h, w = img.shape[:2]; sy, sx = H/h, W/w
```

```
    y2 = np.arange(H); x2 = np.arange(W)
    y  = (y2+0.5)/sy - 0.5; x = (x2+0.5)/sx - 0.5
    y0 = np.floor(y).astype(int); x0 = np.floor(x).astype(int)
    y1 = np.clip(y0+1, 0, h-1);   x1 = np.clip(x0+1, 0, w-1)
    wy = (y-y0).astype(np.float32); wx = (x-x0).astype(np.float32)
    w00 = (1-wy)[:,None]*(1-wx)[None,:]; w01 = (1-wy)[:,None]*wx[None,:]
    w10 = wy[:,None]*(1-wx)[None,:];     w11 = wy[:,None]*wx[None,:]
    # ...standard bilinear blend for gray/RGB...
```

Listing 8: Q8: core zoomers (exact target size)

Image pair	nSSD (NN)	nSSD (Bilinear)
`im01small` → `im01`	0.012058	**0.010187**
`im02small` → `im02`	0.004190	**0.002915**
`im03small` → `im03`	0.007530	**0.005634**

Table 1: Q8: normalized SSD after 4× upscaling (lower is better).

Discussion. Bilinear interpolation consistently yields a lower nSSD (closer to the originals) than nearest–neighbour by smoothing aliasing that NN preserves.

Question 9: GrabCut segmentation and background blur

(a) Segmentation: The image was segmented with GrabCut. A trimap was initialized as probable background everywhere, with sure background at the image margins, a loose ellipse of probable foreground covering only the large flower, and a sure-background rectangle suppressing the small bud. GrabCut was run with `GC_INIT_WITH_MASK`; final labels $\{\text{FG}, \text{PR\_FG}\}$ were converted to a binary mask. Foreground and background were obtained with `cv2.bitwise_and`.

```
mask = np.full(img.shape[:2], cv.GC_PR_BGD, np.uint8)
mask[:m,:] = mask[-m:,:] = mask[:,:m] = mask[:,-m:] = cv.GC_BGD    # sure BG at borders
cv.ellipse(mask, center, axes, 0, 0, 360, cv.GC_PR_FGD, -1)         # probable FG (big flower)
cv.rectangle(mask, (bx0,by0), (bx1,by1), cv.GC_BGD, -1)             # sure BG (small bud)

bgModel = np.zeros((1,65), np.float64); fgModel = np.zeros((1,65), np.float64)
cv.grabCut(img, mask, None, bgModel, fgModel, 5, cv.GC_INIT_WITH_MASK)

mask_fg = np.where((mask==cv.GC_FGD)|(mask==cv.GC_PR_FGD), 255, 0).astype(np.uint8)
fg = cv.bitwise_and(img, img, mask=mask_fg)
bg = cv.bitwise_and(img, img, mask=cv.bitwise_not(mask_fg))
```

Listing 9: Q9: Essential GrabCut + extraction

(b) Enhanced image: The background was strongly blurred and recomposited with a feathered alpha derived from the mask to avoid hard edges:

$$I_{\text{out}} = \alpha\, I_{\text{orig}} + (1-\alpha)\,(\text{GaussianBlur}(I_{\text{orig}}, \sigma = 15)), \quad \alpha = \tfrac{1}{255}\,\text{GaussianBlur}(\text{mask}, \sigma \approx 3).$$

(a) Final mask. (b) Foreground. (c) Background. (d) Enhanced.

Figure 14: Q9(a–b): GrabCut segmentation and background-blurred enhancement.

(c) Why does the background just beyond the flower edge look dark? When a hard (binary) mask is used, pixels at the immediate background side of the boundary are blurred together with *zero/black* values from the masked foreground. Gaussian blur averages across the boundary, so these zeros pull the local average down, creating a dark halo. Using a *feathered* alpha (soft mask) to blend the original and blurred images reduces this artifact.