- **Assignment title** : Point Operations and Spatial Filtering
- **Name** : K.M.S.B. Chandrasena
- **Index Number** : D/ENG/24/0034/ET
- **Module name** : Image Processing and Machine Vision
- **GitHub profile link** : https://github.com/SHELINI-34/ET3112_Assignment_01

Q1: Gamma Correction and Contrast Stretching

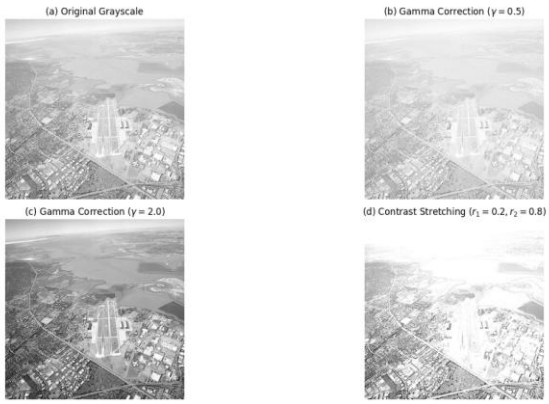


Fig. 1: Gamma correction and Contrast stretching results

```
def apply_gamma(image, gamma):
    return np.power(image, gamma)
gamma_05 = apply_gamma(img_norm, 0.5)
gamma_20 = apply_gamma(img_norm, 2.0)
```

Gamma correction changes how we see brightness. When γ < 1, darker regions are enhanced, revealing shadow details. When γ > 1, higher intensities are suppressed, resulting in a darker image.

```
r1 = 0.2
r2 = 0.8
def contrast_stretch(image, r1, r2):
    stretched = np.zeros_like(image)
    stretched[image < r1] = 0
    mask = (image >= r1) & (image <= r2)
    stretched[mask] = (image[mask] - r1) / (r2 - r1)
    stretched[image > r2] = 1
    return stretched
contrast_img = contrast_stretch(img_norm, r1, r2)
```

This method is more targeted. Instead of changing the whole image smoothly, it takes a specific range of "gray" pixels (0.2 to 0.8) and stretches them to cover the full range from pure black to pure white. This makes the runway pop out much more clearly from the background because it ignores the extreme light/dark outliers and focuses on the most important details.

Q2. Highlights & Shadows

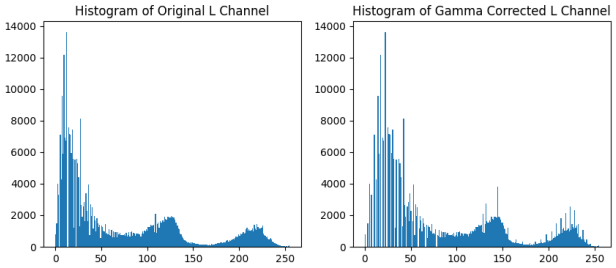
(a)

Fig. 2: Highlight and shadow enhancement using LAB color space.

```
lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
L, a, b = cv2.split(lab)
L_norm = L / 255.0
gamma = 0.8
L_gamma = np.power(L_norm, gamma)
L_gamma_uint8 = (L_gamma * 255).astype(np.uint8)
lab_gamma = cv2.merge((L_gamma_uint8, a, b))
img_gamma = cv2.cvtColor(lab_gamma, cv2.COLOR_LAB2BGR)
```

The image was converted to the L*a*b* color space so we could fix the lighting without changing the actual colors. By applying **gamma = 0.8** only to the L (brightness) channel, we brightened the dark rocks to show their texture while keeping the white clothes and skin tones looking natural.

(b)



- **Original:** Pixels are clustered on the left, indicating a dark/underexposed image.
- **Corrected:** The distribution shifts right and spreads out, proving increased brightness and better tonal range.

Q3. Histogram Equalization

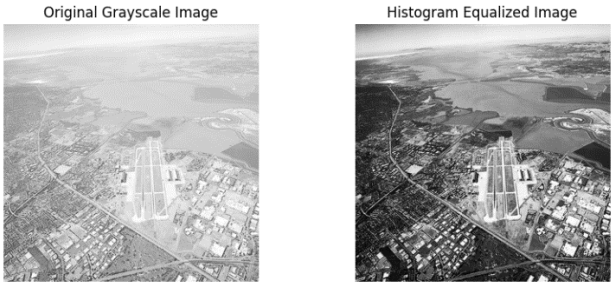
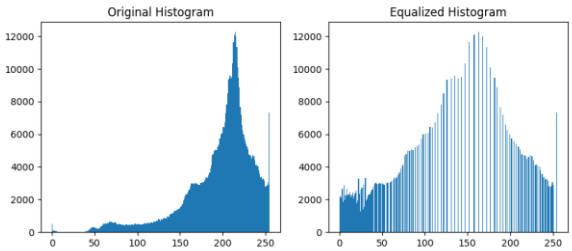


Fig. 3: Histogram equalization results showing original and equalized images with corresponding histograms.




```
def histogram_equalization(image):
    hist, bins = np.histogram(image.flatten(), 256, [0, 256])
    cdf = hist.cumsum()
    cdf_normalized = cdf / cdf[-1]
    equalized = np.floor(255 * cdf_normalized[image]).astype(np.uint8)
    return equalized
img_eq = histogram_equalization(img)
```

This function improves contrast by spreading pixel intensities evenly across the 0–255 range using the cumulative distribution function (cdf). It transforms the original clustered histogram into a uniform distribution, making hidden details much clearer throughout the entire runway image.

Q4. Image Thresholding

Fig. 4: Binary image obtained using Otsu's thresholding & Enhanced foreground

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# --- Part (a): Otsu thresholding ---
threshold_value, binary_mask = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
print(f"Reported Threshold Value: {threshold_value}")
```

- Using Otsu's method, an optimal **threshold value** = **101.0**

Otsu's method was used to automatically calculate the optimal intensity threshold to separate the image into two classes. By applying cv2.THRESH_BINARY_INV, a mask was created where the dark foreground (the woman and the room) is represented by white pixels (255) and the bright background is black (0).

```
# --- Part (b): Histogram equalization on foreground only ---
equalized_img = gray.copy()
foreground_pixels = gray[binary_mask == 255]
equalized_foreground = cv2.equalizeHist(foreground_pixels)
equalized_img[binary_mask == 255] = equalized_foreground.flatten()
```

Hidden Features Revealed:

- **Texture and Clothing:** Details of the woman's sweater and hair become visible.
- **Room Interior:** Architectural details of the doorway, the texture of the walls and the railing of the staircase in the bottom right corner emerge from the darkness.
- **Depth:** The spatial relationship between the woman and the interior room objects is now clear.

Q5. Gaussian Filtering

(a) A normalized 5×5 Gaussian kernel was computed using σ = 2. The kernel coefficients sum to unity.

```
sigma = 2
kernel_size = 5
k = kernel_size // 2
x, y = np.mgrid[-k:k+1, -k:k+1]
gaussian_kernel = np.exp(-(x**2 + y**2) / (2 * sigma**2))
gaussian_kernel = gaussian_kernel / gaussian_kernel.sum()
print("5x5 Normalized Gaussian Kernel:\n", gaussian_kernel)
```

(b)

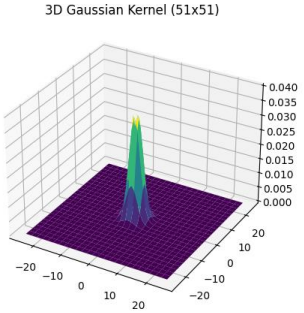


Fig. 5: 3D surface plot of a 51×51 Gaussian kernel.

```
x3, y3 = np.mgrid[-k3:k3+1, -k3:k3+1]
gaussian_3d = np.exp(-(x3**2 + y3**2) / (2 * sigma**2))
gaussian_3d = gaussian_3d / gaussian_3d.sum()

fig = plt.figure(figsize=(6, 5))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x3, y3, gaussian_3d, cmap='viridis')
ax.set_title('3D Gaussian Kernel (51x51)')
plt.show()
```

Fig. 6: Gaussian smoothing using manual implementation and OpenCV.

Gaussian filtering reduces noise by weighted averaging of neighboring pixels. The manually implemented filter produces results comparable to OpenCV's optimized GaussianBlur function, validating the correctness of the computed kernel.

Q6. Derivative of Gaussian

(a) First-order derivatives of the Gaussian

$$G(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \quad \text{——①}$$

Differentiate w.r.t. x.

$$\frac{\partial G}{\partial x} = \frac{1}{2\pi\sigma^2} \cdot \frac{\partial}{\partial x}\left[\exp\left(-\frac{x^2+y^2}{2\sigma}\right)\right]$$

By chain rule.

$$\frac{\partial}{\partial x} \exp f(x) = \exp(f(x)) \cdot f'(x)$$

Here, $f(x) = -\frac{x^2+y^2}{2\sigma} \Rightarrow f'(x) = -\frac{x}{\sigma^2}$

By substituting

$$\frac{\partial G}{\partial x} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma}\right) \cdot \left(-\frac{x}{\sigma^2}\right) \quad \text{——②}$$

By ① & ②,

$$\frac{\partial G}{\partial x} = -\frac{x}{\sigma^2} G(x,y)$$

Similarly,

$$\frac{\partial G}{\partial y} = -\frac{y}{\sigma^2} G(x,y)$$

(b)

```
x, y = np.mgrid[-k:k+1, -k:k+1]
G = (1 / (2 * np.pi * sigma**2)) * np.exp(-(x**2 + y**2) / (2 * sigma**2))
Gx = -(x / sigma**2) * G
Gy = -(y / sigma**2) * G
Gx = Gx / np.sum(np.abs(Gx))
Gy = Gy / np.sum(np.abs(Gy))
```

(c)

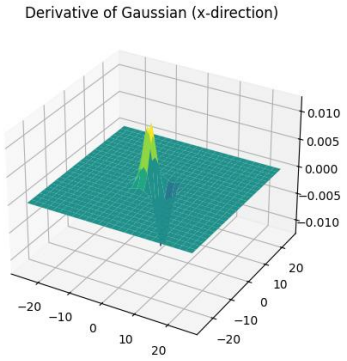


Fig. 7: 3D surface visualization of the derivative of Gaussian kernel.

(d)

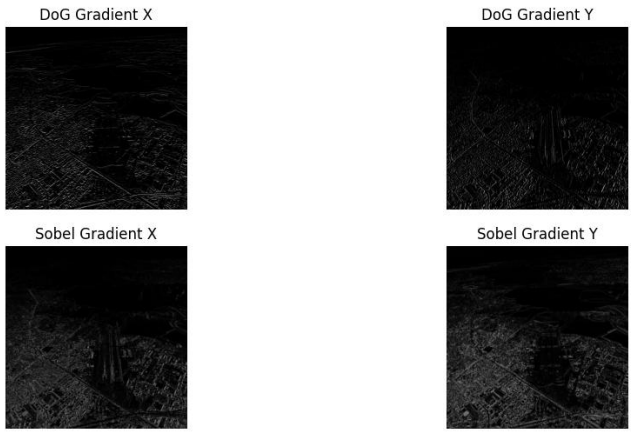


Fig. 8: Image gradients obtained using derivative of Gaussian and Sobel operators.

```
# PART (d): IMAGE GRADIENTS
grad_x_dog = cv2.filter2D(img, -1, Gx)
grad_y_dog = cv2.filter2D(img, -1, Gy)
# PART (e): SOBEL GRADIENTS
sobel_x = cv2.Sobel(img, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(img, cv2.CV_64F, 0, 1, ksize=3)
sobel_x = np.uint8(np.absolute(sobel_x) / np.max(np.absolute(sobel_x)) * 255)
sobel_y = np.uint8(np.absolute(sobel_y) / np.max(np.absolute(sobel_y)) * 255)
```

(e)

Compared to Sobel filtering, the derivative-of-Gaussian approach produces smoother gradient images with reduced noise sensitivity. Sobel filters detect edges more sharply but amplify noise due to the absence of prior smoothing.

Q7. Image Zooming and SSD

Fig. 9: Images zooming using nearest-neighbor and bilinear interpolation.

(a)
```
def zoom_nearest(img, scale):

    h, w = img.shape[:2]
    new_h, new_w = int(round(h * scale)), int(round(w * scale))
    zoomed = np.zeros((new_h, new_w, 3), dtype=img.dtype)

    for i in range(new_h):
        for j in range(new_w):
            x = min(int(i / scale), h - 1)
            y = min(int(j / scale), w - 1)
            zoomed[i, j] = img[x, y]

    return zoomed
```

(b)
```
def zoom_bilinear(img, scale):

    h, w = img.shape[:2]
    new_h, new_w = int(round(h * scale)), int(round(w * scale))
    zoomed = np.zeros((new_h, new_w, 3), dtype=np.float32)
```

Image	SSD(Nearest)	SSD(Bilinear)
taylor_small.jpg to taylor.jpg	0.003683	0.003469
taylor_very_small.jpg to taylor.jpg	0.007394	0.006883
im01small.png to im01.png	0.002110	0.001779
im02small.png to im02.png	0.000410	0.000283
Im03 im03small.png to im03.png	0.001016	0.000755

The SSD values show that **Bilinear interpolation** is more accurate than **Nearest-Neighbor** because it averages the four closest pixels to create smooth transitions, while Nearest-Neighbor simply copies the single closest pixel, leading to blocky edges and higher error. This trend also shows that the smaller the starting image, the higher the SSD becomes, as there is less original information available to accurately reconstruct the details.

Q8. Gaussian vs Median Filtering

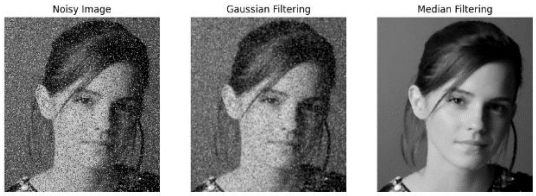


Fig. 10: Noise removal using Gaussian and median filtering.

```
gaussian = cv2.GaussianBlur(img, (5, 5), sigmaX=1)
median = cv2.medianBlur(img, 5)
```

Median filtering is more effective in removing impulse noise while preserving edges, whereas Gaussian filtering introduces edge blurring due to averaging.

Q9. Image Sharpening

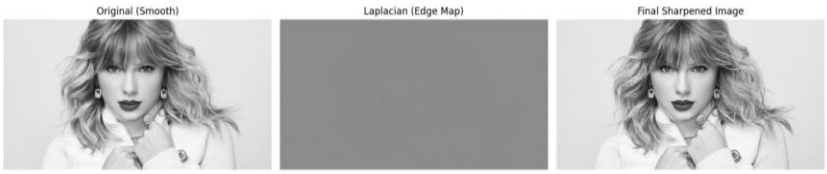


Fig. 11: Image sharpening using Laplacian filtering.

```
laplacian = cv.Laplacian(img, cv.CV_64F)
# Sharpen by subtracting the Laplacian from the original
sharpened_lap = img - laplacian
# Post-processing
sharpened_lap = np.clip(sharpened_lap, 0, 255).astype(np.uint8)
```

Sharpening was done by using a Laplacian filter to find edges, then combining them with the original image to boost contrast. This enhances fine details and textures, making the output look much crisper. The values were clipped to 0–255 to keep the image stable and clear.

Q10. Bilateral Filtering

Fig. 12: Comparison of Gaussian, OpenCV bilateral, and manually implemented bilateral filtering.

```
def bilateral_filter_manual(img, diameter, sigma_s, sigma_r):

    h, w = img.shape
    radius = diameter // 2
    output = np.zeros_like(img, dtype=np.float32)
    for i in range(def bilateral_filter_manual(img, diameter, sigma_s, sigma_r):

    h, w = img.shape
    radius = diameter // 2
    output = np.zeros_like(img, dtype=np.float32)


    for i in range(h):
        for j in range(w):
            wp = 0
            filtered_value = 0
            i_pad, j_pad = i + radius, j + radius
```

```
            p_intensity = padded_img[i_pad, j_pad]
            for x in range(-radius, radius + 1):
                for y in range(-radius, radius + 1):
                    q_intensity = padded_img[i_pad + x, j_pad + y]
                    # Spatial Weight
                    spatial_dist_sq = x**2 + y**2
                    spatial_weight = math.exp(-spatial_dist_sq / (2 * sigma_s**2))
                    # Range Weight
                    intensity_diff_sq = (float(q_intensity) - float(p_intensity))**2
                    range_weight = math.exp(-intensity_diff_sq / (2 * sigma_r**2))
                    weight = spatial_weight * range_weight
                    wp += weight
                    filtered_value += weight * q_intensity
            output[i, j] = filtered_value / wp
    return np.uint8(np.clip(output, 0, 255))
```

(b)

```
# (b) Gaussian Smoothing
gaussian = cv2.GaussianBlur(img, (7, 7), sigmaX=3)
```

(c)

```
# (c) Bilateral Filtering (OpenCV)

bilateral_cv = cv2.bilateralFilter(img, d=7, sigmaColor=50, sigmaSpace=50)
```

(d)

```
# (d) Bilateral Filtering (Manual)
bilateral_manual = bilateral_filter_manual(img, diameter=7, sigma_s=3, sigma_r=50)
```

Bilateral filtering improves upon Gaussian smoothing because it cleans up noise without blurring the important edges. While a Gaussian filter only considers the spatial distance between pixels, the bilateral filter adds a second weight based on intensity difference. This means the filter only averages pixels that are both nearby and similar in brightness, preventing it from blurring across sharp boundaries. Consequently it effectively smooths out flat, noisy areas while keeping structural outlines and textures perfectly crisp.