Homework 3: Connected Component Analysis & Color Correction

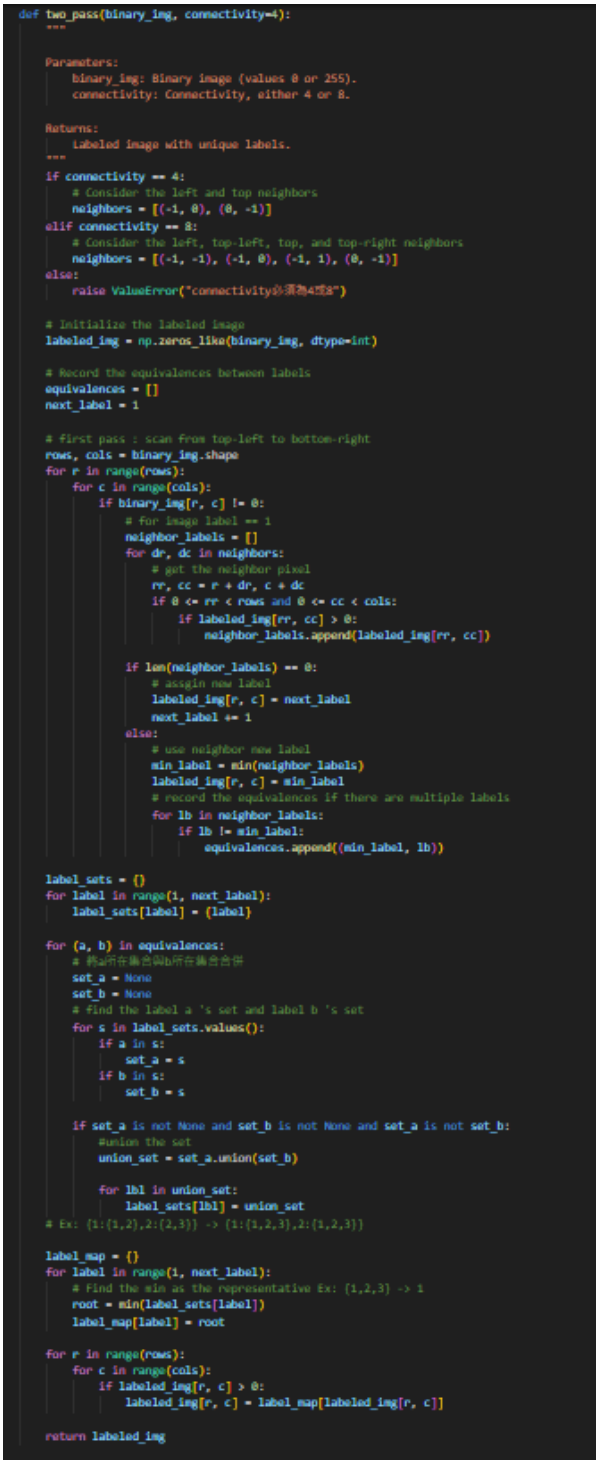
Part I. Implementation

Connected Component

```python
def to_binary(img):
    #Covert the image to gray style
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    #Apply the threshold to the image
    thersohold = 128
    binary = (gray<thersohold)*255
    binary = binary.astype(np.uint8)
    return binary
```

The function `to_binary(img)`:

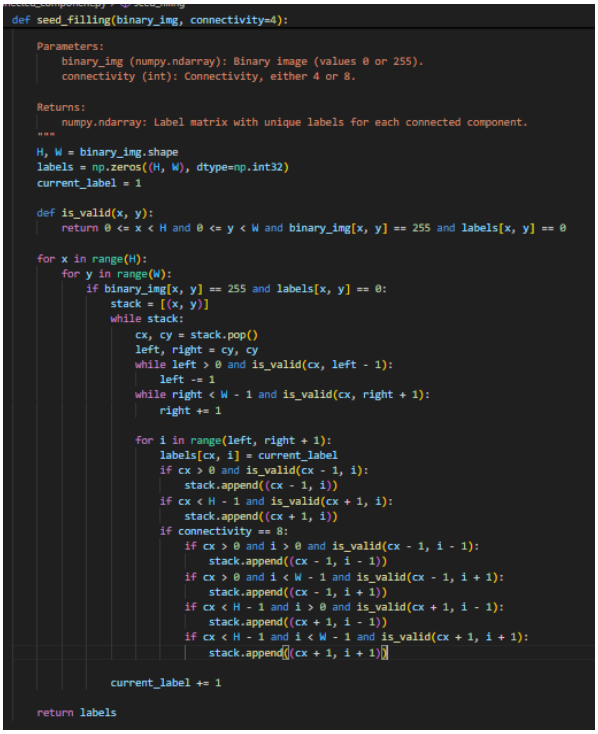
1. Converts a given image to grayscale.
2. Applies a threshold (128) to create a binary image where pixel values are either 0 or 255.



```python
def two_pass(binary_img, connectivity=4):
    """

    Parameters:
        binary_img: Binary image (values 0 or 255).
        connectivity: Connectivity, either 4 or 8.

    Returns:
        Labeled image with unique labels.
    """
    if connectivity == 4:
        # Consider the left and top neighbors
        neighbors = [(-1, 0), (0, -1)]
    elif connectivity == 8:
        # Consider the left, top-left, top, and top-right neighbors
        neighbors = [(-1, -1), (-1, 0), (-1, 1), (0, -1)]
    else:
        raise ValueError("connectivity必須為4或8")

    # Initialize the labeled image
    labeled_img = np.zeros_like(binary_img, dtype=int)

    # Record the equivalences between labels
    equivalences = []
    next_label = 1

    # first pass : scan from top-left to bottom-right
    rows, cols = binary_img.shape
    for r in range(rows):
        for c in range(cols):
            if binary_img[r, c] != 0:
                # for image label == 1
                neighbor_labels = []
                for dr, dc in neighbors:
                    # get the neighbor pixel
                    rr, cc = r + dr, c + dc
                    if 0 <= rr < rows and 0 <= cc < cols:
                        if labeled_img[rr, cc] > 0:
                            neighbor_labels.append(labeled_img[rr, cc])

                if len(neighbor_labels) == 0:
                    # assgin new label
                    labeled_img[r, c] = next_label
                    next_label += 1
                else:
                    # use neighbor new label
                    min_label = min(neighbor_labels)
                    labeled_img[r, c] = min_label
                    # record the equivalences if there are multiple labels
                    for lb in neighbor_labels:
                        if lb != min_label:
                            equivalences.append((min_label, lb))

    label_sets = {}
    for label in range(1, next_label):
        label_sets[label] = {label}

    for (a, b) in equivalences:
        # 將a所在集合與b所在集合合併
        set_a = None
        set_b = None
        # find the label a 's set and label b 's set
        for s in label_sets.values():
            if a in s:
                set_a = s
            if b in s:
                set_b = s

        if set_a is not None and set_b is not None and set_a is not set_b:
            #union the set
            union_set = set_a.union(set_b)

            for lbl in union_set:
                label_sets[lbl] = union_set
    # Ex: {1:{1,2},2:{2,3}} -> {1:{1,2,3},2:{1,2,3}}

    label_map = {}
    for label in range(1, next_label):
        # Find the min as the representative Ex: {1,2,3} -> 1
        root = min(label_sets[label])
        label_map[label] = root

    for r in range(rows):
        for c in range(cols):
            if labeled_img[r, c] > 0:
                labeled_img[r, c] = label_map[labeled_img[r, c]]

    return labeled_img
```

The `two_pass` function performs connected component labeling on a binary image with either 4- or 8-connectivity.

1. **First Pass**:
    - Scans the image from top-left to bottom-right.
    - Assigns a new label if no labeled neighbors exist or records equivalences between multiple labels.
2. **Equivalence Resolution**:
    - Merges equivalent labels into unified sets, ensuring consistent labeling.
3. **Second Pass**:
    - Updates the image using the minimum label from equivalence sets for consistent labeling across connected components.
4. **Output**:
    - Returns a labeled image where each connected component has a unique label.



```python
def seed_filling(binary_img, connectivity=4):
    """
    Parameters:
        binary_img (numpy.ndarray): Binary image (values 0 or 255).
        connectivity (int): Connectivity, either 4 or 8.

    Returns:
        numpy.ndarray: Label matrix with unique labels for each connected component.
    """
    H, W = binary_img.shape
    labels = np.zeros((H, W), dtype=np.int32)
    current_label = 1

    def is_valid(x, y):
        return 0 <= x < H and 0 <= y < W and binary_img[x, y] == 255 and labels[x, y] == 0

    for x in range(H):
        for y in range(W):
            if binary_img[x, y] == 255 and labels[x, y] == 0:
                stack = [(x, y)]
                while stack:
                    cx, cy = stack.pop()
                    left, right = cy, cy
                    while left > 0 and is_valid(cx, left - 1):
                        left -= 1
                    while right < W - 1 and is_valid(cx, right + 1):
                        right += 1

                    for i in range(left, right + 1):
                        labels[cx, i] = current_label
                        if cx > 0 and is_valid(cx - 1, i):
                            stack.append((cx - 1, i))
                        if cx < H - 1 and is_valid(cx + 1, i):
                            stack.append((cx + 1, i))
                        if connectivity == 8:
                            if cx > 0 and i > 0 and is_valid(cx - 1, i - 1):
                                stack.append((cx - 1, i - 1))
                            if cx > 0 and i < W - 1 and is_valid(cx - 1, i + 1):
                                stack.append((cx - 1, i + 1))
                            if cx < H - 1 and i > 0 and is_valid(cx + 1, i - 1):
                                stack.append((cx + 1, i - 1))
                            if cx < H - 1 and i < W - 1 and is_valid(cx + 1, i + 1):
                                stack.append((cx + 1, i + 1))

                current_label += 1

    return labels
```

The `seed_filling` function performs connected component labeling using a stack-based Seed Filling approach.

1. **Initialization**:
    - Creates a `labels` matrix initialized to zeros.
    - Starts with `current_label = 1`.
2. **Labeling Process**:
    - Iterates over each pixel in the binary image.
    - If a pixel belongs to a component (value 255) and is unlabeled, a stack is initialized with the current pixel.
    - Expands horizontally (left and right) for each row, marking all connected pixels with the same label.
    - Adds valid neighbors to the stack for vertical (4-connectivity) or diagonal (8-connectivity) expansion.
3. **Label Assignment**:
    - Each new connected component gets a unique label.
4. **Output**:
    - Returns the `labels` matrix, where each connected component has a unique identifier.

```python
def other_cca_algorithm(binary_img,connectivity=4):
    '''implement flood fill'''
    H,W = binary_img.shape
    labels=np.zeros((H,W),dtype=np.int32)
    current_label=1

    def fill(x, y):
        stack = [(x, y)]
        while stack:
            x, y = stack.pop()
            if labels[y, x] == 0 and binary_img[y, x] == 255:
                labels[y, x] = current_label

                if y > 0 and labels[y - 1, x] == 0: stack.append((x, y - 1))
                if y < H - 1 and labels[y + 1, x] == 0: stack.append((x, y + 1))
                if x > 0 and labels[y, x - 1] == 0: stack.append((x - 1, y))
                if x < W - 1 and labels[y, x + 1] == 0: stack.append((x + 1, y))

                if connectivity == 8:
                    if y > 0 and x > 0 and labels[y - 1, x - 1] == 0: stack.append((x - 1, y - 1))
                    if y > 0 and x < W - 1 and labels[y - 1, x + 1] == 0: stack.append((x + 1, y - 1))
                    if y < H - 1 and x > 0 and labels[y + 1, x - 1] == 0: stack.append((x - 1, y + 1))
                    if y < H - 1 and x < W - 1 and labels[y + 1, x + 1] == 0: stack.append((x + 1, y + 1))

    for y in range(H):
        for x in range(W):
            if binary_img[y, x] == 255 and labels[y, x] == 0:
                fill(x, y)
                current_label += 1
    print(labels)
    return labels
```

1. Initialization

- Create a `labels` matrix initialized to zeros.
- Set `current_label = 1` for unique component IDs.

2. Labeling Process

- Iterate through each pixel in the binary image:
    1. If the pixel is foreground (`255`) and unlabeled:
        - Use a stack to flood-fill connected pixels.
        - Assign the current label to all connected pixels.
    2. **4-connectivity**: Check neighbors (top, bottom, left, right).
    3. **8-connectivity**: Also check diagonal neighbors.

3. Label Assignment

- Increment `current_label` for each new connected component.

4. Output

- Return the `labels` matrix, where each unique label represents a connected component.

```python
def color_mapping(labels):
    # Create an empty RGB image
    H, W = labels.shape
    color_img = np.zeros((H, W, 3), dtype=np.uint8)


    # Assign a random color to each label
    unique_labels = np.unique(labels) # Get all unique labels
    label_to_color = {label: tuple(np.random.randint(0, 255, size=3)) for label in unique_labels if label > 0}
    # Ex : {1: (255, 0, 0), 2: (0, 255, 0), 3: (0, 0, 255)}

    for y in range(H):
        for x in range(W):
            if labels[y, x] > 0:
                color_img[y, x] = label_to_color[labels[y, x]]

    return color_img
```

1. Initialization

- Creates an empty RGB image (`color_img`) of the same height and width as `labels`.

2. Random Color Assignment

- Extracts all unique labels using `np.unique(labels)`.
- Generates random RGB colors for each label greater than `0` using a dictionary (`label_to_color`).
    - Example: `{1: (255, 0, 0), 2: (0, 255, 0)}`.

3. Pixel Coloring

- Iterates through each pixel in the `labels` matrix:
    - If the label is greater than `0`, assigns the corresponding color from `label_to_color` to the pixel in `color_img`.

4. Output

- Returns the resulting RGB image (`color_img`) where each connected component has a unique color.

Color Correction

```python
def white_patch_algorithm(img):
    # get the maximum value of each channel
    max_R = np.max(img[:, :, 2])
    max_G = np.max(img[:, :, 1])
    max_B = np.max(img[:, :, 0])

    # 按比例放大
    img = img.astype(np.float32)
    img[:, :, 2] = img[:, :, 2] * (255 / max_R)
    img[:, :, 1] = img[:, :, 1] * (255 / max_G)
    img[:, :, 0] = img[:, :, 0] * (255 / max_B)

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

1. Maximum Channel Value Calculation

- Calculates the maximum intensity value for each color channel (R, G, B):

2. Intensity Normalization

- Converts the image to a floating-point format (`np.float32`) for accurate scaling.
- Scales each pixel value in the R, G, and B channels proportionally:

3. Clipping and Type Conversion

4. Output

- Returns the adjusted image with corrected white balance.

```python
def gray_world_algorithm(img):
    # get the average value of each channel
    avg_R = np.mean(img[:, :, 2])
    avg_G = np.mean(img[:, :, 1])
    avg_B = np.mean(img[:, :, 0])

    # calculate the ratio of each channel
    mean_all = (avg_R+avg_G+avg_B)/3
    sacle_R = mean_all/avg_R
    sacle_G = mean_all/avg_G
    sacle_B = mean_all/avg_B

    # According to the ratio, adjust the value of each channel
    img = img.astype(np.float32)
    img[:, :, 2] = img[:, :, 2] * sacle_R
    img[:, :, 1] = img[:, :, 1] * sacle_G
    img[:, :, 0] = img[:, :, 0] * sacle_B

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

1. Average Channel Value Calculation

- Computes the average intensity value for each color channel:

2. Mean Ratio Calculation

- Calculates the overall mean of all three channel averages (`mean_all`).
- Derives scaling ratios for each channel to align their average intensity with the overall mean:

3. Channel Adjustment

- Converts the image to floating-point format (`np.float32`) for accurate scaling.

- Adjusts each channel based on the calculated scaling ratios.

4. Clipping and Type Conversion

5. Output

- Returns the white-balanced image.

```python
def other_white_balance_algorithm(img):
    """
    Gaussian-Based White Balance Algorithm.
    """
    # turn the image into float32
    img = img.astype(np.float32)


    for i in range(3):
        channel = img[:, :, i]

        # calculate the mean and standard deviation of the channel
        mean = np.mean(channel)
        std = np.std(channel)


        if std > 0:
            img[:, :, i] = (channel - mean) / std * 64 + 128  # Remapping
        else:
            img[:, :, i] = channel  # Noting change

    # clip and convert the image back to uint8
    img = np.clip(img, 0, 255).astype(np.uint8)
    return img
```

1.Input Preparation

- Converts the input image to a floating-point format (`np.float32`) to ensure precision in calculations.

2. Channel Normalization

- Iterates through each color channel (R, G, B):
  - Calculate the mean and std
  - Adjusts pixel intensities using the formula: new_pixel_value=(channel-mean)/std*64+128
  - If the standard deviation is 0, no adjustment is made (to avoid division by zero).

3. Clipping and Type Conversion

4. Output

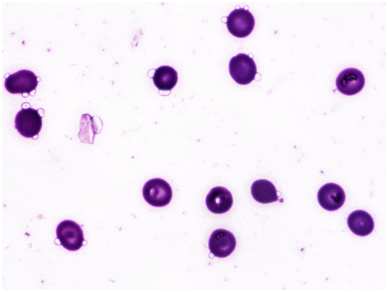
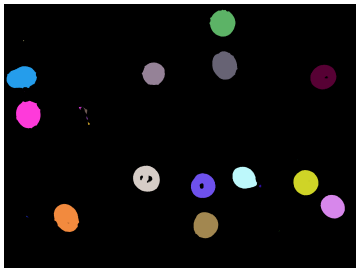
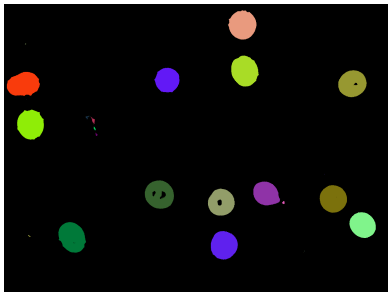
- Returns the white-balanced image where pixel values are normalized and re-mapped for balanced appearance.

Part II. Results & Analysis

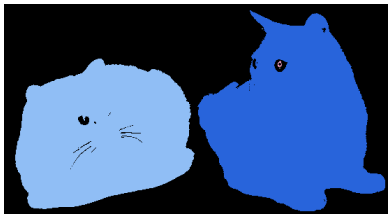
Task 1: Connected Component Analysis

Two-pass Algorithm

Original | 4-coonectivity | 4-coonectivity

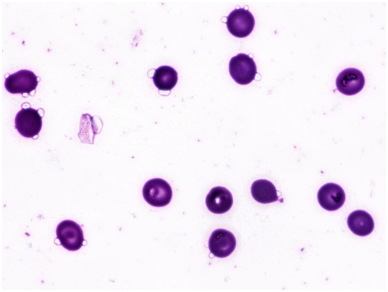
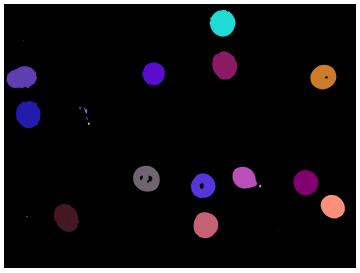
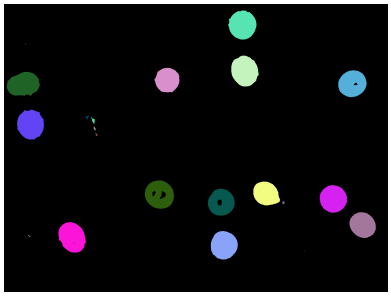








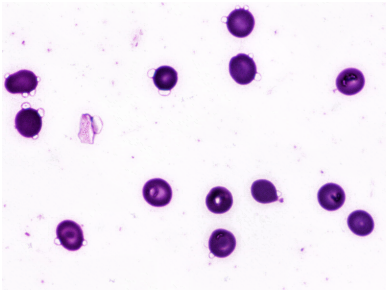
Seed-filling Algorithm

Original | 4-coonectivity | 4-coonectivity

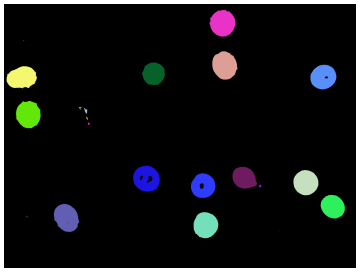




(Bonus) Algotithms Flood-filling Algo

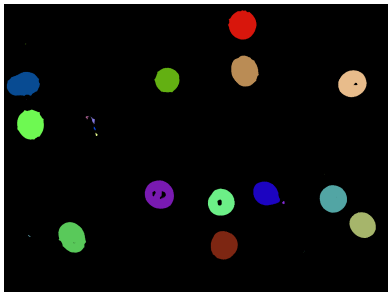
Original



4-coonectivity



4-coonectivity



Compare and discuss the above result

Aspect	Two-pass	Seed Filling	Flood Filling
Method Description	Uses a label scanning technique to traverse connected components	Expands from a seed point recursively or iteratively to identify	Similar to seed filling but generally more efficient, suitable for larger regions.

	twice, marking each distinct region.	connected components.	
Component Count	Consistent across all connected components.	Consistent across all connected components.	Consistent across all connected components.
Centroid Positions	Centroids are almost identical across methods.	Centroids are almost identical across methods.	Centroids are almost identical across methods.
Minor Differences	Rare, minor differences may occur near edges or noise but are negligible overall.	Rare, minor differences may occur near edges or noise but are negligible overall.	Rare, minor differences may occur near edges or noise but are negligible overall.
Performance	Suitable for medium-scale components, accurate results but requires two full scans.	Performs well with small components but may require more resources in dense regions.	Highly efficient and ideal for processing large-scale or extensive connected regions.
Strengths	Accurate and reliable for moderate datasets.	Good for smaller and simpler connected components.	Efficient for larger and more complex datasets.
Weaknesses	Requires two passes, which can be slower for very large datasets.	Recursive nature may cause higher memory usage in dense or large areas.	May require additional optimization for smaller datasets with sparse components.
Best Use Case	Scenarios where accuracy is paramount and dataset size is moderate.	Ideal for smaller, less complex images or where memory constraints are manageable.	Large-scale image processing or scenarios requiring faster execution.
Overall Consistency	Produces consistent results, no significant deviations from other methods.	Produces consistent results, no significant deviations from other methods.	Produces consistent results, no significant deviations from other methods.

Task 2: Color Correction

White Patch Algorithm

Principle

- The **White Patch Algorithm** operates on the assumption that the brightest pixel in an image corresponds to a perfect white under the illumination source.
- It adjusts the color channels (R, G, B) so that the brightest pixel in each channel becomes white (or neutral gray), thereby neutralizing the effect of the scene's light source.

Advantages

1. **Simplicity**:
    - The algorithm is computationally light and easy to implement.
2. **Effective in Controlled Lighting**:
    - Performs well in scenes where the brightest area is truly white or neutral.

Limitations

1. **Relies on White Regions**:
    - If the image does not contain any white pixels, the algorithm may fail to produce accurate results.
2. **Sensitive to Highlights**:
    - Bright reflections or highlights can be mistaken for white, leading to incorrect adjustments.

Applications

- Suitable for scenes where there is a clear white object or patch.
- Often used in controlled environments like product photography or studio setups.

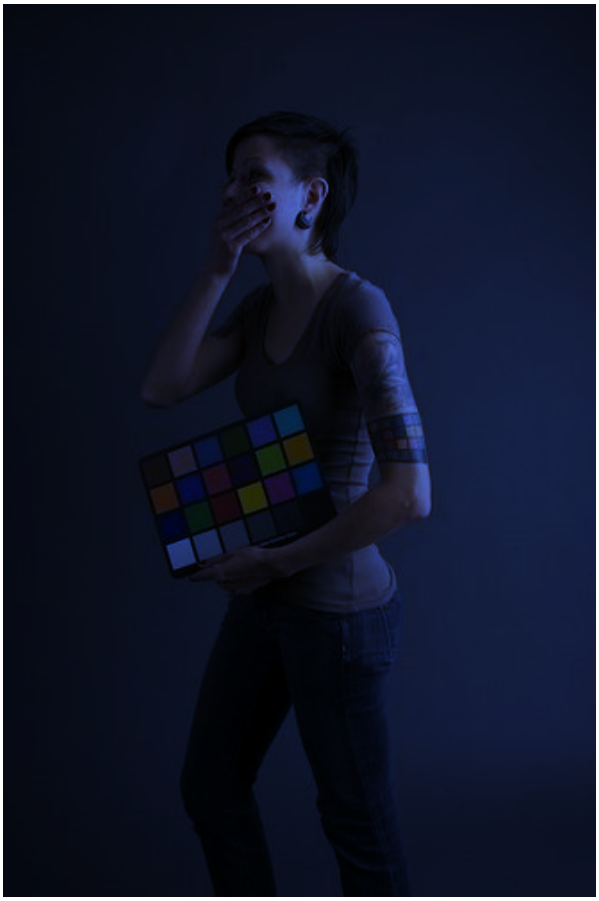
Original Image

Result Image

- The White Patch Algorithm effectively removes the color cast and enhances the color accuracy of the image. However, the result depends on having a bright white reference point in the original image, which this scene satisfies.

Original Image

Result Image



- The White Patch Algorithm effectively eliminates the excessive blue cast, restoring natural colors and improving overall visual clarity. This correction is particularly effective when the image has a pronounced color imbalance due to lighting conditions or incorrect camera settings.

Gray-world Algorithm

Principle

- The **Gray World Algorithm** operates on the assumption that, under normal lighting conditions, the average color of an image should be a neutral gray.
- It adjusts the color channels (R, G, B) so that their averages are equal, thereby neutralizing any color bias caused by the illumination source.

Advantages

1. **Broad Applicability**:

- Does not require a specific white or neutral object in the scene, making it more versatile than the White Patch Algorithm.

2. **Robust for General Scenes**:
    - Works well in diverse environments where the scene contains a variety of colors evenly distributed.

Limitations

1. **Assumes Balanced Colors**:
    - The algorithm assumes the average of all colors in the scene is gray, which may not hold true for scenes dominated by a single color (e.g., a field of grass).
2. **Struggles with Strong Color Casts**:
    - In scenes with extreme color biases, the algorithm may fail to produce accurate results.

Applications

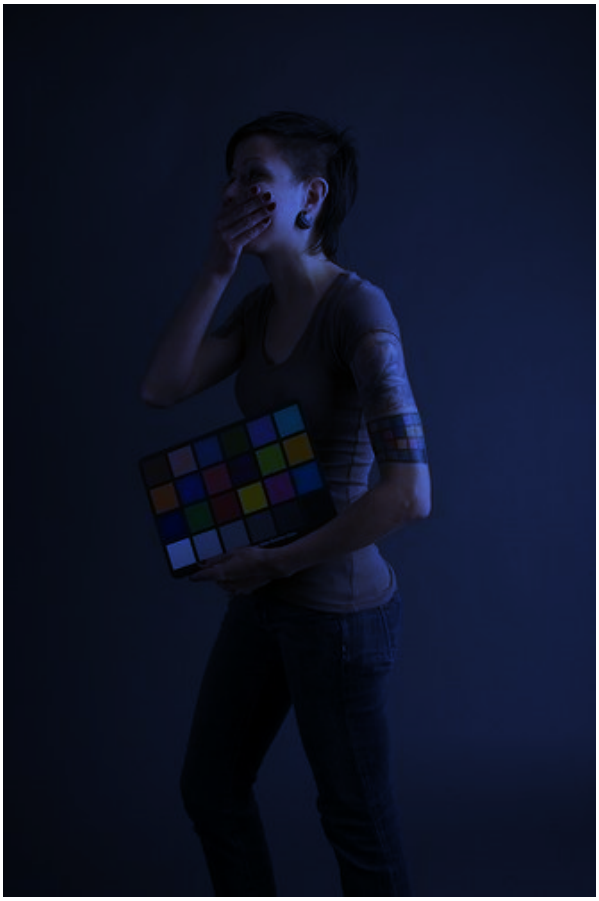
- Suitable for general-purpose photography where the scene contains a balanced distribution of colors.
- Often used in applications like photo editing software or camera firmware for basic white balance correction.

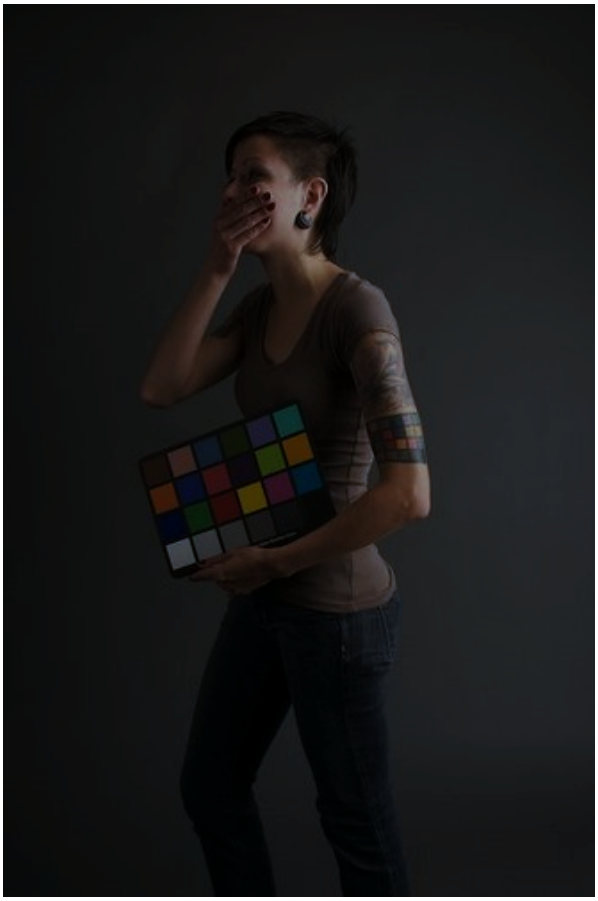
Original Image Result Image

- The **Gray-world Algorithm** works well to neutralize strong color casts (e.g., the yellow/orange in the original image), but it may overcorrect in certain cases, introducing a slight blue or cold tint. This behavior is typical for Gray-world corrections, as it averages the color distribution without considering the scene's context or expected color temperature.

Original Image

Result Image





- The **Gray-world Algorithm** effectively neutralizes the heavy blue tint present in the original image, improving the balance and accuracy of colors. However, this method can sometimes make images appear slightly darker, as it equalizes the overall color intensity. For a more vibrant result, combining Gray-world with brightness or contrast adjustments might be beneficial.

Other Algotithms (Gaussian-Based White Balance Algorithm.)

Principle

- The **Gaussian-Based White Balance Algorithm** assumes that the distribution of colors in an image under a specific illuminant follows a Gaussian distribution in the RGB color space.

- It estimates the scene illumination by analyzing the statistical properties (mean and variance) of the red, green, and blue channels, then adjusts the channels to neutralize the color cast.

Advantages

1. **Statistical Robustness**:
    - By leveraging the Gaussian model, this algorithm effectively handles color casts caused by a dominant illuminant.
2. **Versatile in Mixed Lighting**:
    - Works well in scenes with complex lighting conditions by considering the overall statistical properties of the image.
3. **Balances Subtle Color Variations**:
    - Provides a more nuanced correction compared to simpler algorithms like White Patch or Gray World.

Limitations

1. **Assumption of Gaussian Distribution**:
    - The algorithm assumes the color distribution is Gaussian, which may not hold true in all scenes, especially those with highly non-uniform lighting.
2. **Computational Complexity**:
    - More computationally intensive compared to simpler methods, as it involves fitting and adjusting based on Gaussian statistics.
3. **Dependent on Global Analysis**:
    - Struggles in scenarios where the illuminant varies significantly across the image.

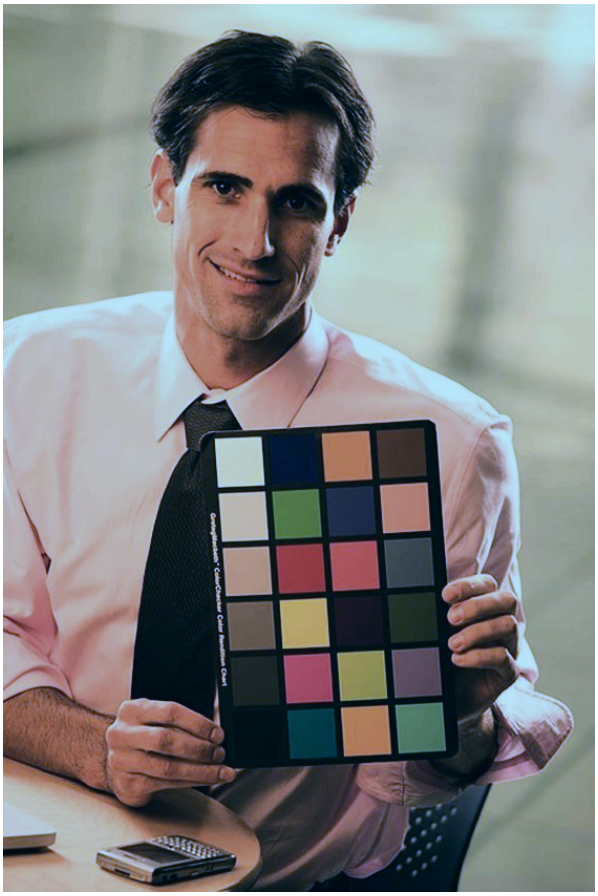
Applications

- Ideal for photography or video processing in environments with challenging lighting (e.g., sunsets, artificial lighting).

- Used in advanced computational photography workflows to improve color balance and natural appearance.
- Suitable for professional-grade imaging systems that require more precise and adaptive white balance adjustments.

Original Image

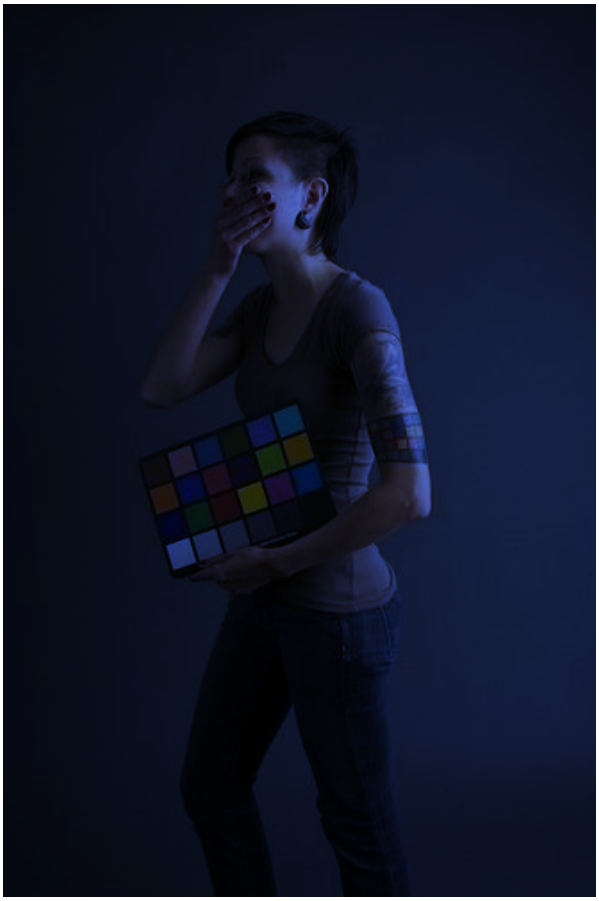
Result Image



- The **Gaussian-Based White Balance Algorithm** successfully neutralized the strong color cast in the original image, resulting in more natural and visually appealing colors.

Original Image

Result Image



- The result image demonstrates the effectiveness of the applied white balance algorithm in correcting color bias. It improves both the realism and color accuracy, making the final output more suitable for professional use or visual clarity.

Compareand discuss the above result.

Aspect	White Patch Algorithm	Gray World Algorithm	Gaussian-Based White Balance Algorithm
Principle	Assumes the brightest pixel represents pure white and adjusts channels accordingly.	Assumes the average color of the image should be neutral gray and adjusts channels to match.	Assumes color distribution follows a Gaussian model and adjusts based on statistical properties.
Advantages	Simple and computationally light.	Does not require explicit white regions.	Robust for complex lighting conditions.

	Effective for scenes with clear white regions.	Works well in balanced color scenes.	Corrects both subtle and strong color casts.
Limitations	Requires a white region in the scene. Sensitive to bright reflections or highlights.	Assumes naturally balanced colors, which may fail in scenes dominated by a single color.	Assumes Gaussian distribution, which may not always hold. More computationally intensive.
Applications	Suitable for scenes with clear white objects or patches. Often used in studio or controlled environments.	Useful for general-purpose photography with balanced color distributions. Basic correction in cameras.	Ideal for challenging lighting conditions, such as sunsets or artificial lighting. Used in advanced imaging workflows.
Performance in Simple Scenes	Highly effective when a white region is present.	Performs well in balanced color environments.	Performs adequately but may not be necessary for simple lighting conditions.
Performance in Complex Scenes	Struggles in the absence of white regions or under mixed lighting.	May fail when the assumption of average gray is violated.	Performs better in mixed and complex lighting conditions due to statistical modeling.

Part III. Answer the questions

- Please describe a problem you encountered and how you solved it?

> Ans:
> One problem I encountered was optimizing connected component analysis in large-scale images. Initially, the implementation using a recursive approach for seed-filling resulted in a stack overflow error for images with a high density of components.

> To solve this problem, I switched to an iterative implementation using a stack data structure, which avoided excessive memory usage by limiting recursion depth. Additionally, I used a flood-filling algorithm, which proved more efficient for handling large and complex images, as it reduced unnecessary computation on sparse areas.

- What are the advantages and limitations of two-pass and seed-filling algorithms for object segmentation in images, and in which scenarios are they most appropriate?

> Ans:
>
> - **Two-pass Algorithm**
>   - **Advantages**: Accurate and consistent; processes the entire image systematically, ensuring all connected components are labeled correctly.
>   - **Limitations**: Requires two full scans of the image, which can be time-consuming for large datasets.
>   - **Most Appropriate Scenarios**: Ideal for moderate-sized images where accuracy is a priority, especially when processing time is not critical.
> - **Seed-Filling Algorithm**
>   - **Advantages**: Efficient for small and sparse regions; can adapt to dynamic shapes by expanding from a seed point.
>   - **Limitations**: Recursive implementations may lead to stack overflow for large or complex regions; iterative

versions may require additional data structures.

- **Most Appropriate Scenarios**: Best suited for small images with isolated objects or dynamic environments where components vary in size and shape.

- What are the advantages and limitations of the white patch and gray-world algorithms for image white balance, and in which scenarios are they most appropriate?

Ans:

- **White Patch Algorithm**
  - **Advantages**: Simple and effective in well-controlled lighting conditions; assumes the brightest pixel in the image represents white.
  - **Limitations**: Sensitive to specular highlights and may fail in images without a clear white region.
  - **Most Appropriate Scenarios**: Best for studio photography or images with distinct white objects under uniform lighting.
- **Gray-World Algorithm**
  - **Advantages**: Assumes the average color of an image should be gray, making it robust in a variety of scenes.
  - **Limitations**: May fail in images with dominant colors (e.g., a red sunset or underwater images), leading to incorrect white balance.

- **Most Appropriate Scenarios**: Suitable for general-purpose photography where there is a mix of colors and no single dominant color.