# Visual Computing

## TP-5

-

## “ Image segmentation“

Participants:

RIDAOUI Hatim

&

NASSIR SHAFI Ayda

## Overview

This assignment explores K-means clustering for image segmentation, grouping pixels by similarity in RGB values and, optionally, spatial location. We examine the impact of initial center values and the number of regions K on segmentation results. A convergence-based stop condition is implemented.

**Additionally, we analyze how including pixel location in clustering affects results and adjust the balance between color and location to achieve optimal segmentation. Finally, we apply the method to different images, tuning K, initialization, and the color-location balance for the best outcome.**

## Introduction

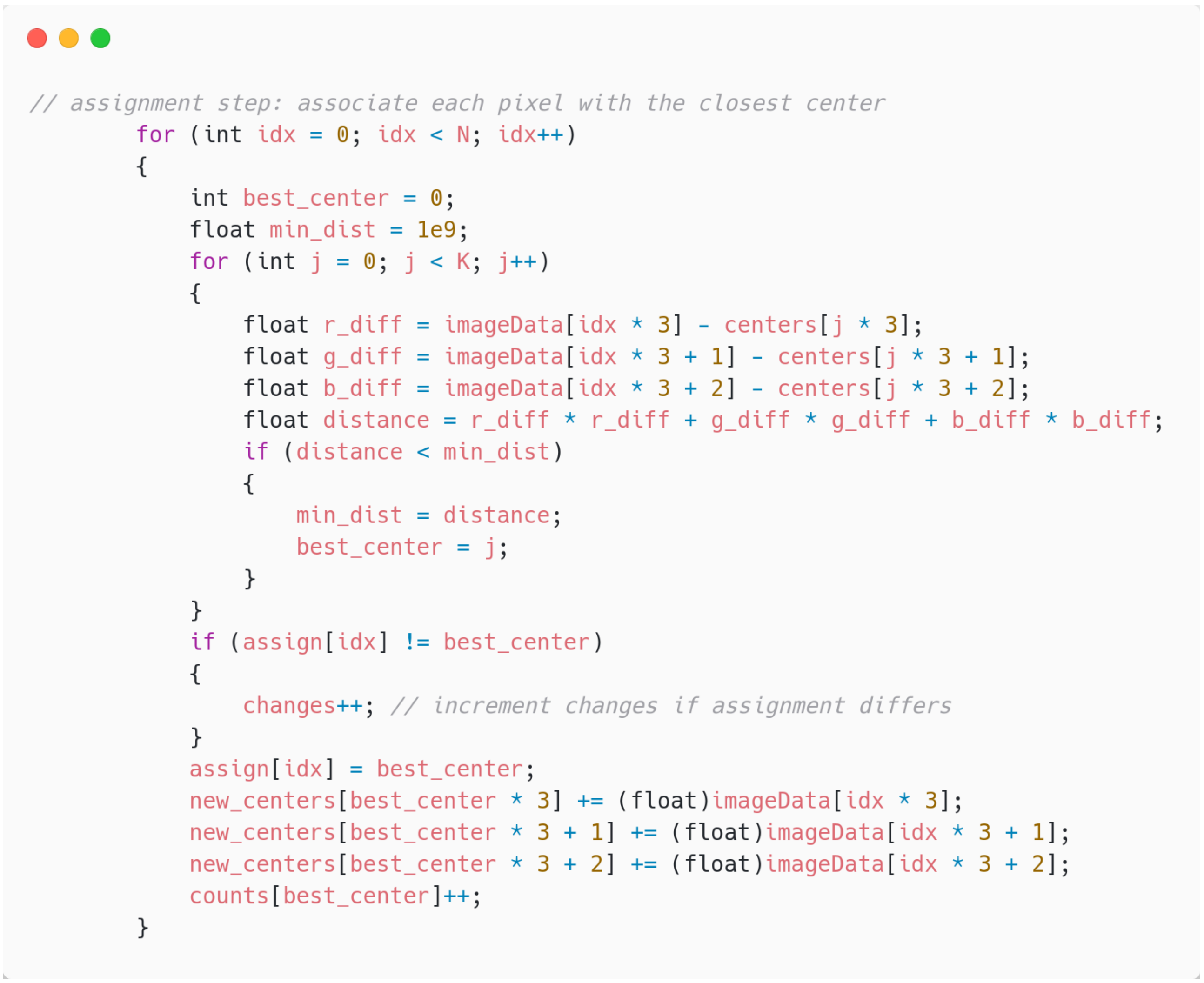
Image segmentation is a fundamental task in computer vision and image processing that involves partitioning an image into meaningful regions. These regions typically correspond to different objects, surfaces, or textures within the image. In this practical work, we explore image segmentation using the K-means clustering algorithm, focusing on its implementation and how various parameters affect its performance.

K-means clustering is a widely-used unsupervised learning algorithm that partitions a dataset into K clusters.

1. Assignment: Each pixel is assigned to the cluster whose center is closest to its feature vector.
2. Update: The cluster centers are updated as the mean of all feature vectors assigned to them.

By experimenting with different parameters and testing the algorithm on various images (frog and bell pepper images), we aim to analyze and understand the strengths and limitations of K-means clustering for image segmentation.

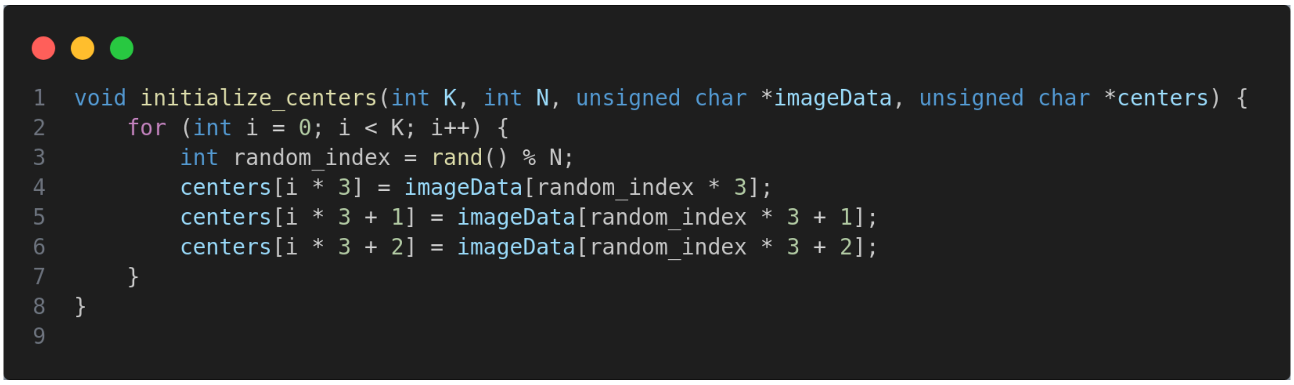
## Implementation of K means



```
// assignment step: associate each pixel with the closest center
        for (int idx = 0; idx < N; idx++)
        {
            int best_center = 0;
            float min_dist = 1e9;
            for (int j = 0; j < K; j++)
            {
                float r_diff = imageData[idx * 3] - centers[j * 3];
                float g_diff = imageData[idx * 3 + 1] - centers[j * 3 + 1];
                float b_diff = imageData[idx * 3 + 2] - centers[j * 3 + 2];
                float distance = r_diff * r_diff + g_diff * g_diff + b_diff * b_diff;
                if (distance < min_dist)
                {
                    min_dist = distance;
                    best_center = j;
                }
            }
            if (assign[idx] != best_center)
            {
                changes++; // increment changes if assignment differs
            }
            assign[idx] = best_center;
            new_centers[best_center * 3] += (float)imageData[idx * 3];
            new_centers[best_center * 3 + 1] += (float)imageData[idx * 3 + 1];
            new_centers[best_center * 3 + 2] += (float)imageData[idx * 3 + 2];
            counts[best_center]++;
        }
```

**Fig 1.1** Code snippet of the K-means clustering function for image segmentation

This code uses the K-means algorithm to divide an image into different regions based on color. First, it randomly selects initial cluster centers (representing colors) and assigns each pixel to the nearest cluster based on its RGB values. Then, it recalculates the centers by averaging the colors of all pixels assigned to each cluster. This process repeats up to in given times or until the cluster centers stop changing significantly, showing that the algorithm has converged. Once the centers are stable, the image is updated to reflect the new regions, with each pixel colored according to its cluster, and the program finishes by freeing any memory used.



```c
void initialize_centers(int K, int N, unsigned char *imageData, unsigned char *centers) {
    for (int i = 0; i < K; i++) {
        int random_index = rand() % N;
        centers[i * 3] = imageData[random_index * 3];
        centers[i * 3 + 1] = imageData[random_index * 3 + 1];
        centers[i * 3 + 2] = imageData[random_index * 3 + 2];
    }
}
```

**Fig 1.2** initialize_centers for image segmentation

This function randomly selects K pixels from the image to serve as the initial centers for the K-means clustering algorithm. It does this by choosing a random pixel from the image for each of the K clusters, and then assigns the RGB values of that pixel to the corresponding cluster center. This ensures that each cluster begins with a color that is based on actual pixels from the image.

## Output images

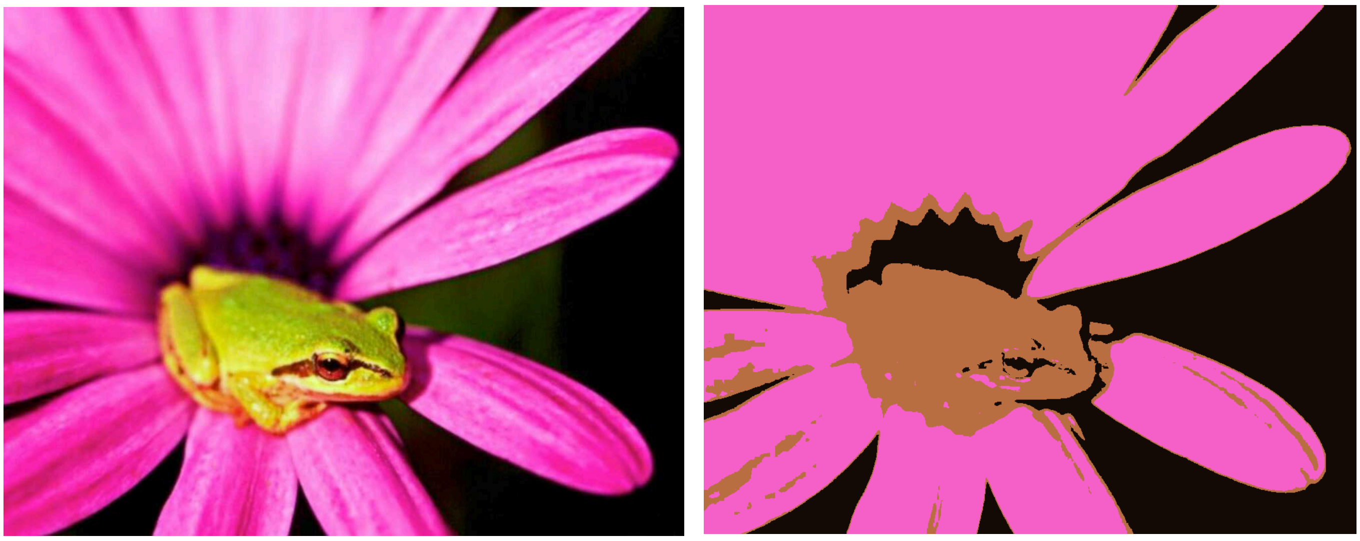
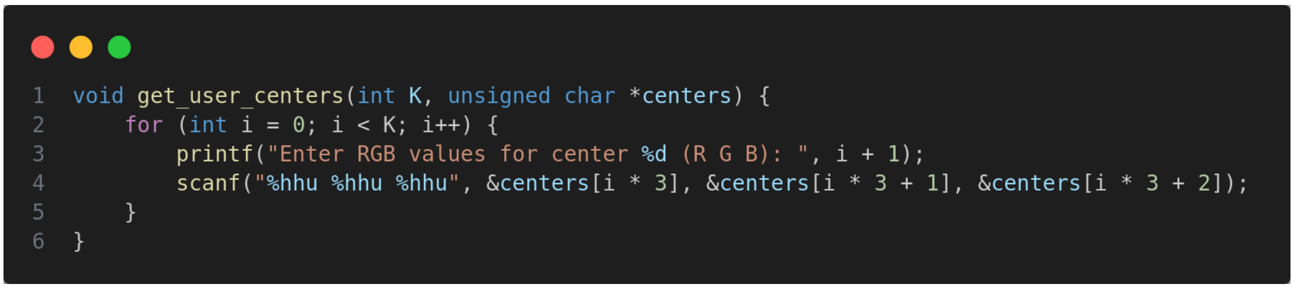


**Fig 1.3** Original vs segmented image with cluster of 3
(Please note that later in this assignment we will going to see the effect of K and experiment with the result)

## Influence of Region Initial Values for Region Centers

The initial placement of the region centers in K-means can greatly affect the final result. If the centers start in poor locations, the algorithm may not group the pixels correctly, leading to poor segmentation. Conversely, if the centers are chosen wisely, the algorithm will converge more quickly and produce more accurate results by better capturing the natural divisions in the image.

For this experiment, we added a function that allows users to input values for the constant cluster K.



```c
void get_user_centers(int K, unsigned char *centers) {
    for (int i = 0; i < K; i++) {
        printf("Enter RGB values for center %d (R G B): ", i + 1);
        scanf("%hhu %hhu %hhu", &centers[i * 3], &centers[i * 3 + 1], &centers[i * 3 + 2]);
    }
}
```

**Fig 1.4** Function to accept user rgb input value

The main function of this method is to display messages for the user to input RGB values which are used as initial values for the centers for cluster K.

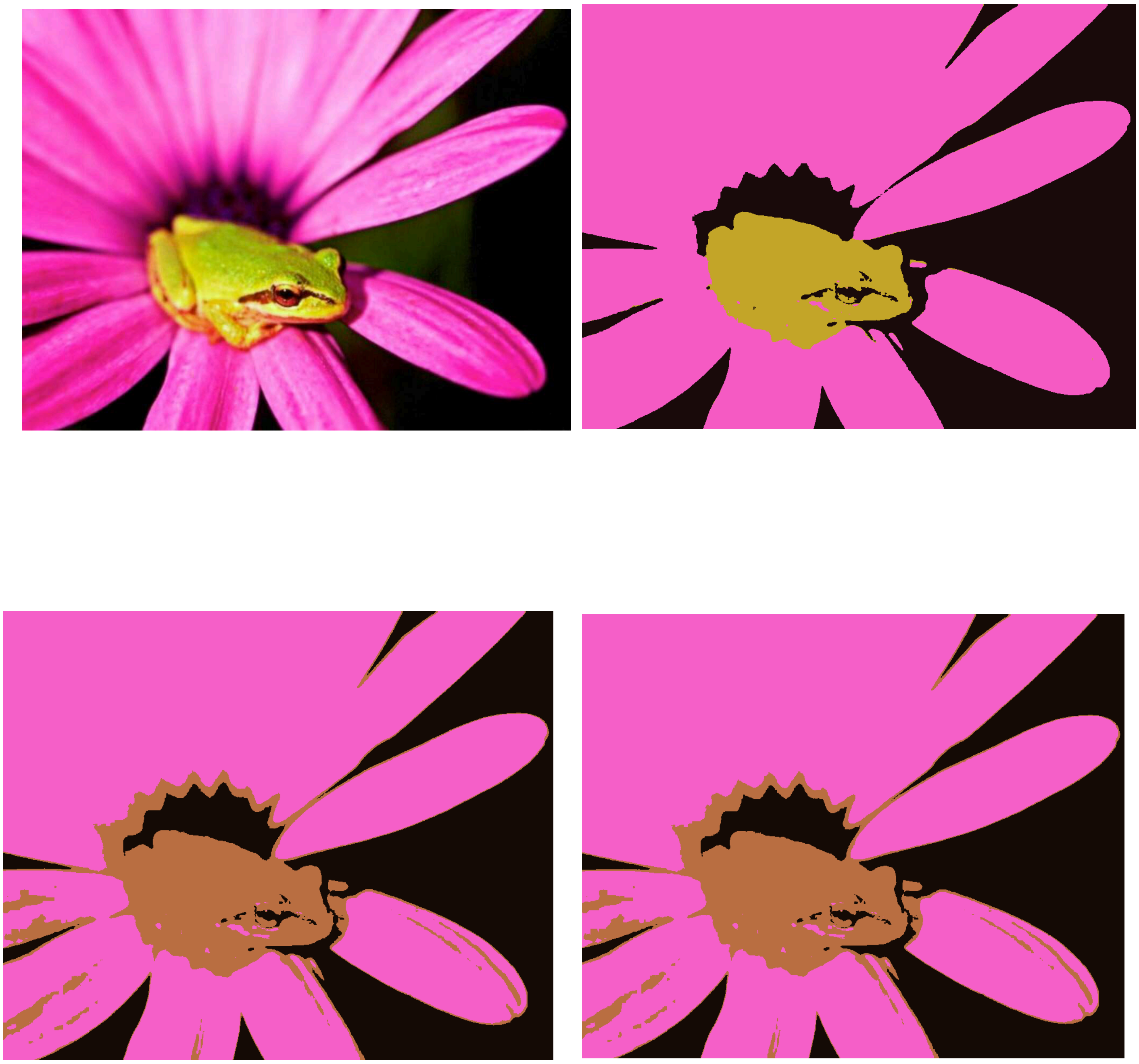


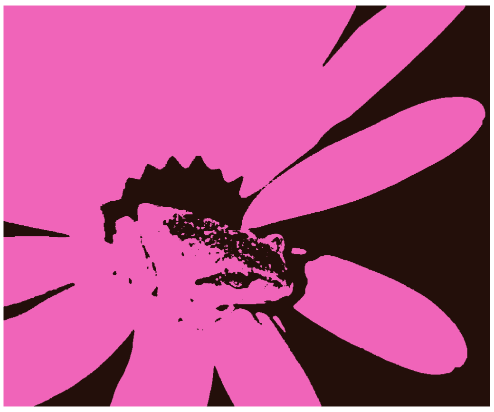
**Fig 1.5** comparison for different initial points for K=3

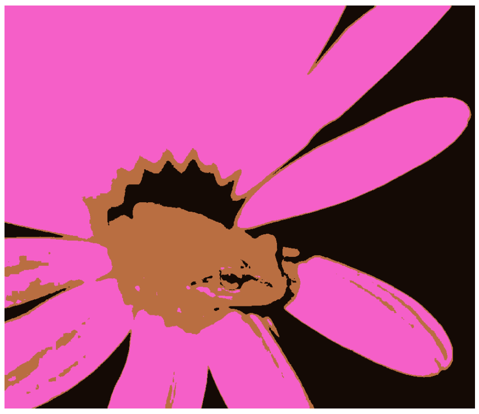
- **top right** with original frog.ppm
- **top left** image with RGB values for center 1(R G B): 255 0 0,RGB values for center 2 (R G B): 0 255 0, RGB values for center 3 (R G B): 0 0 255
- **bottom left** with RGB values for center 1(R G B):128 128 128 ,RGB values for center 2(R G B): = 130 130 130 ,RGB values for center 3(R G B): = 127 127 127 ,
- **bottom right** iRGB values for center 1(R G B): =0 0 0, RGB values for center 2(R G B): = 255 255 255 cRGB values for center 3 (R G B): =128 0 128.

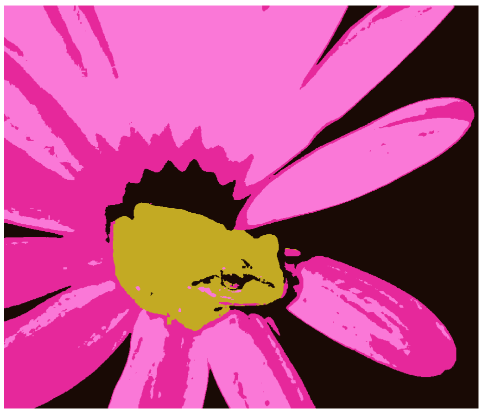
The choice of the value for rgb mainly relies on the closeness, the separation and arbitarity and the quality of segmented image which is discussed below for cluster K choice.

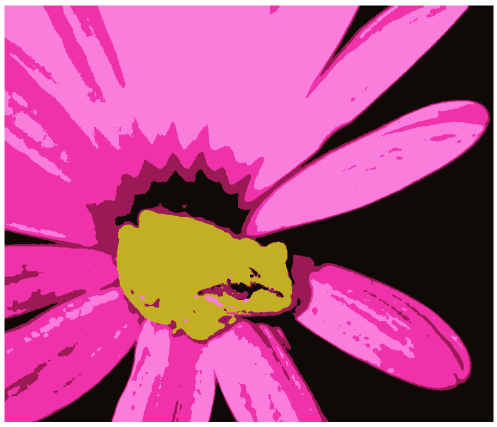
## Influence of the number of regions K

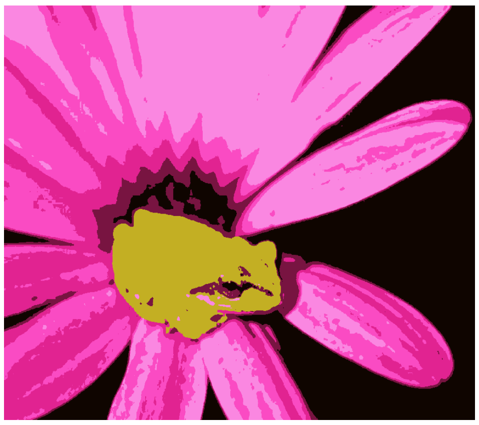
Up until now, we have used a constant K = 3 value for the clusters, even when K was provided by the user. In this section, we will experiment with how the value of K affects the segmentation. The selection of K takes into account how segmentation is impacted for a given image, as it might be under-segmented, well-balanced, or over-segmented.

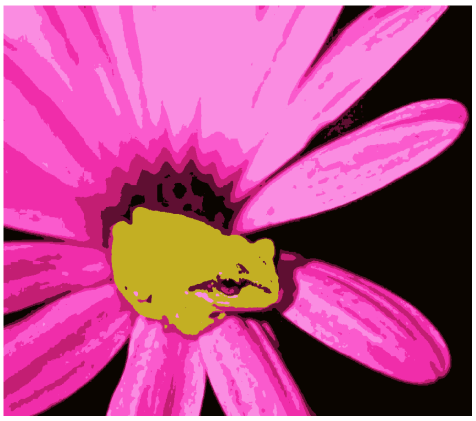

K = 2


K = 3


K = 4


K = 5


K = 6


K =7

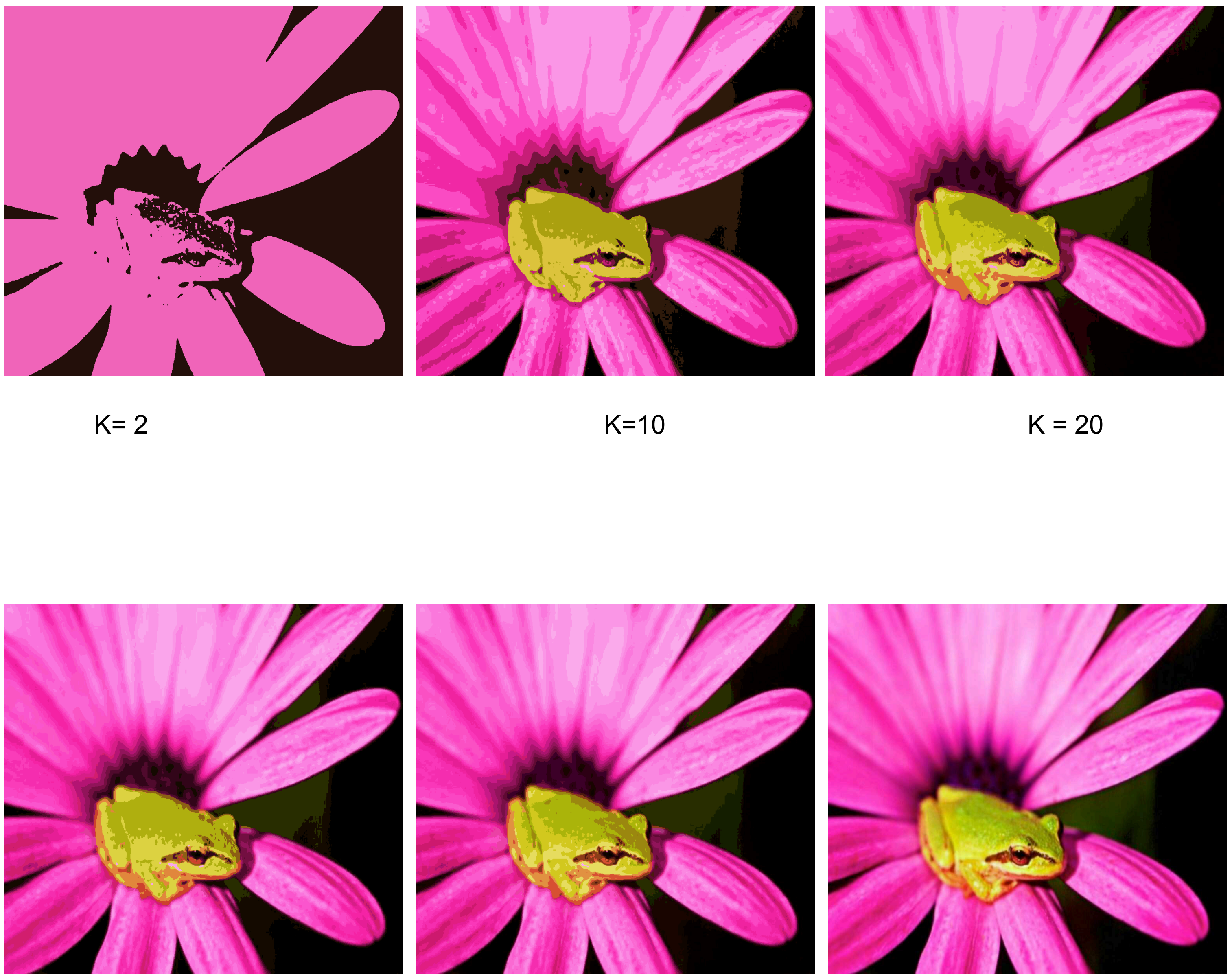
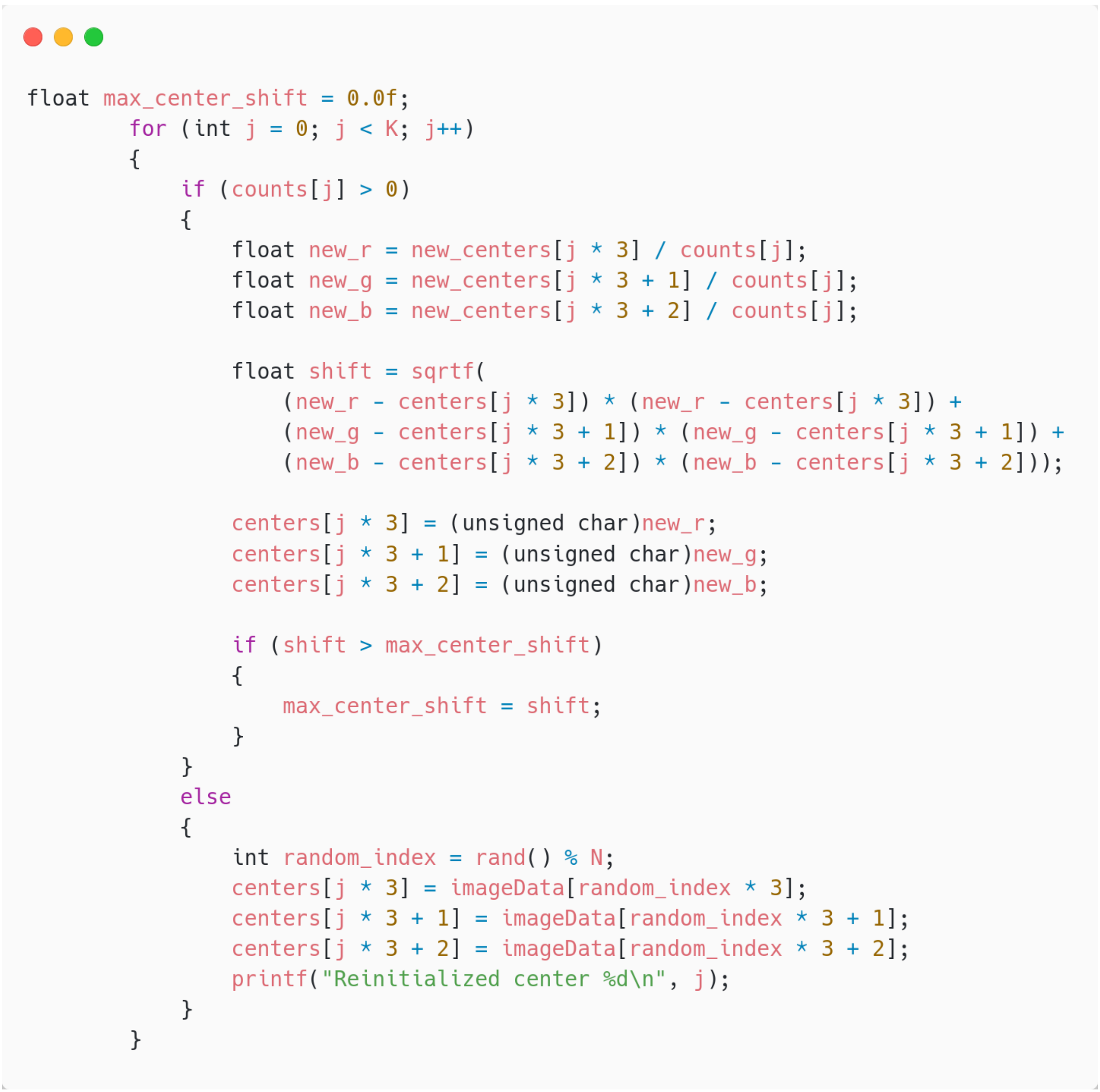


**Fig 1.7** Output of intervaled K values

During our experiment from the output images , we choose K = 10 as the best segmentation K value. Our choice relies on how sharp the boundaries are , consistency within the region, preservation of the original image, being balanced (not under or over) segmented for some.

## Stop condition used in the code



```c
float max_center_shift = 0.0f;
        for (int j = 0; j < K; j++)
        {
            if (counts[j] > 0)
            {
                float new_r = new_centers[j * 3] / counts[j];
                float new_g = new_centers[j * 3 + 1] / counts[j];
                float new_b = new_centers[j * 3 + 2] / counts[j];

                float shift = sqrtf(
                    (new_r - centers[j * 3]) * (new_r - centers[j * 3]) +
                    (new_g - centers[j * 3 + 1]) * (new_g - centers[j * 3 + 1]) +
                    (new_b - centers[j * 3 + 2]) * (new_b - centers[j * 3 + 2]));

                centers[j * 3] = (unsigned char)new_r;
                centers[j * 3 + 1] = (unsigned char)new_g;
                centers[j * 3 + 2] = (unsigned char)new_b;

                if (shift > max_center_shift)
                {
                    max_center_shift = shift;
                }
            }
            else
            {
                int random_index = rand() % N;
                centers[j * 3] = imageData[random_index * 3];
                centers[j * 3 + 1] = imageData[random_index * 3 + 1];
                centers[j * 3 + 2] = imageData[random_index * 3 + 2];
                printf("Reinitialized center %d\n", j);
            }
        }
```

**Fig 1.8** Convergence threshold inside the k-means algorithm

The stop condition in the K-means algorithm uses a convergence threshold to know when to stop. After each iteration, it checks if the cluster centers have shifted by a small enough amount. If the largest shift is less than a set threshold (0.001), it means the algorithm has stabilized and found stable clusters, so further iterations are unnecessary. Additionally, the algorithm has a maximum of 50 iterations to avoid running indefinitely if convergence is slow. This ensures that the algorithm stops once a good solution is found, saving both time and resources. The threshold ensures only significant changes in the cluster centers are considered.

**Implementation of K-means with a Second Stopping Condition**

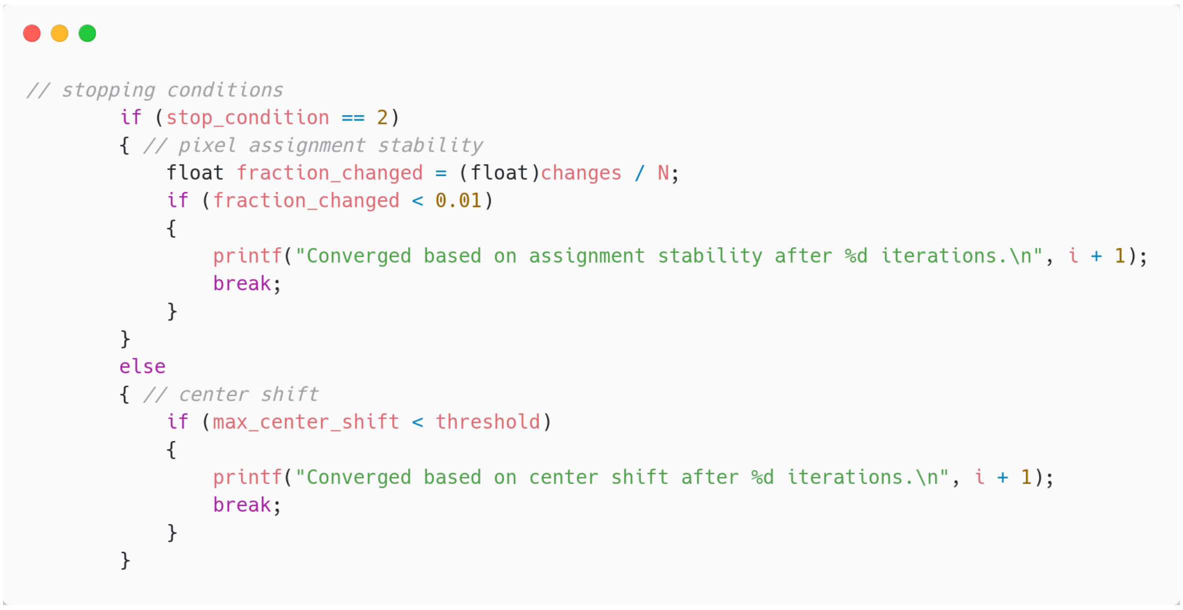
In this section, we extend the implementation of the K-means algorithm to incorporate a second stopping condition. While the original stopping condition terminates the algorithm when the shift in cluster centers falls below a threshold, the newly added condition focuses on the stability of pixel assignments. This enhancement provides a different perspective on convergence, especially when using both RGB and spatial coordinates (i,ji, ji,j) in the feature vector.

**First Stopping Condition: Center Shift Threshold**

- **Description**: The algorithm stops when the maximum movement of any cluster center across iterations is less than a predefined threshold (0.1)
- **Advantages**:
    - Converges faster since it only evaluates cluster center stability.
    - Suitable for cases where clusters stabilize quickly in terms of their centroids.
- **Limitations**:
    - May stop prematurely if pixel assignments are still fluctuating but center movements are small.

**Second Stopping Condition: Pixel Assignment Stability**

- **Description**: The algorithm stops when fewer than 1% of the pixels change their assigned clusters between two consecutive iterations.
- **Advantages**:
    - Ensures that segmentation results are stable at the pixel level.
    - Particularly useful when using RGB + location features, as spatial proximity can introduce additional fluctuations.
- **Limitations**:
    - Requires more iterations, especially for high values of K or larger images



```
// stopping conditions
        if (stop_condition == 2)
        { // pixel assignment stability
            float fraction_changed = (float)changes / N;
            if (fraction_changed < 0.01)
            {
                printf("Converged based on assignment stability after %d iterations.\n", i + 1);
                break;
            }
        }
        else
        { // center shift
            if (max_center_shift < threshold)
            {
                printf("Converged based on center shift after %d iterations.\n", i + 1);
                break;
            }
        }
```

**Fig 1.9** Code snippet of the two stopping conditions in the K-means algorithm

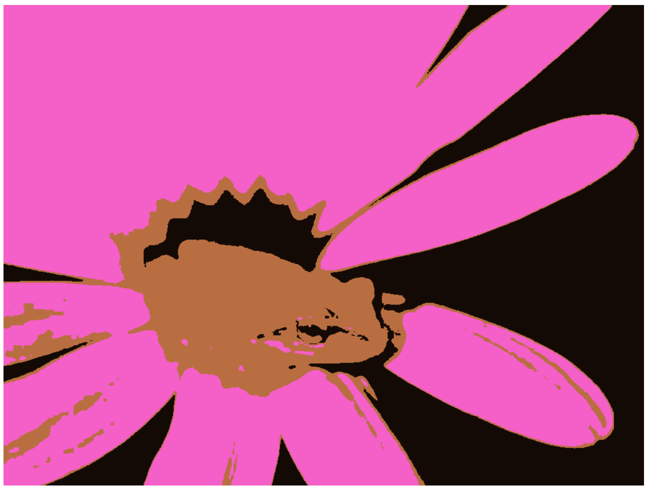
## Comparison of Results with K=3 and K=6

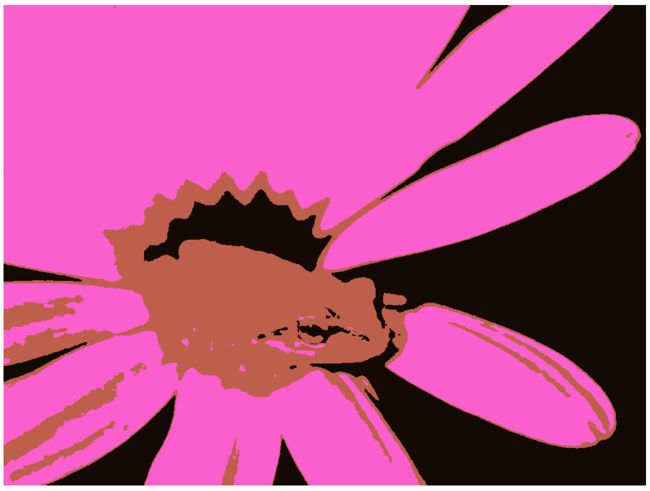
Below, we present segmentation results for both stopping conditions using two different cluster counts (K=3 and K=6). The images demonstrate the practical differences between the two approaches:

**Images**:

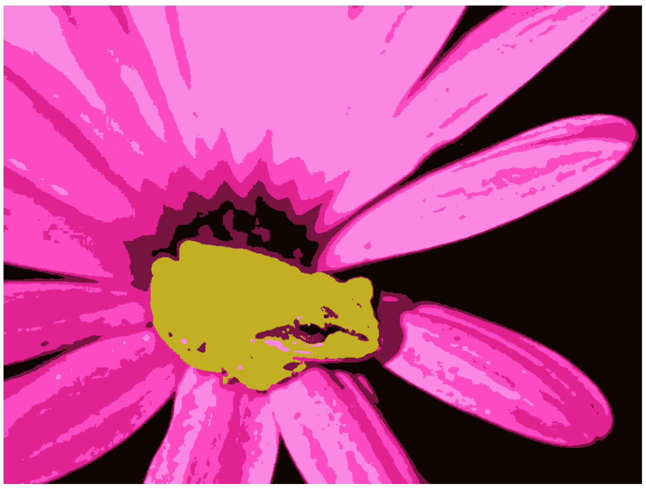
- **Left**: Results with the first stopping condition (center shift threshold).
- **Right**: Results with the second stopping condition (pixel assignment stability).

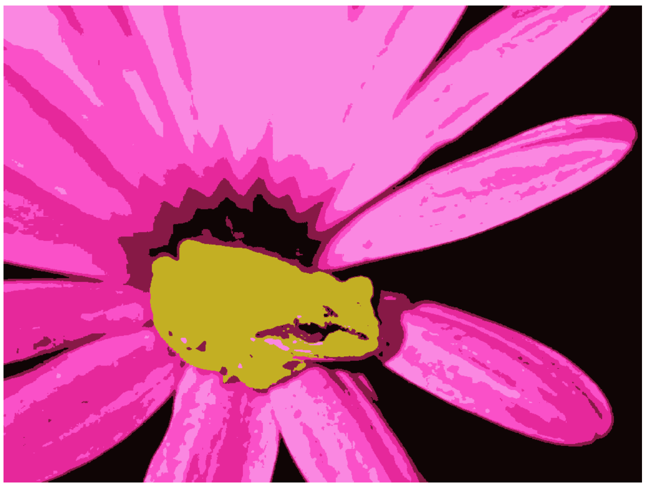
**K = 3**





**K = 6**





**Observations**

1. **With K=3**
   - The first stopping condition produces faster but slightly noisier results, with some disjoint pixel assignments near cluster boundaries.
   - The second stopping condition ensures more coherent clusters, particularly in areas with gradual color changes.
2. **With K=6**
   - The first stopping condition captures fine details but shows minor instability in boundary regions.
   - The second stopping condition provides smoother transitions and spatially consistent clusters, benefiting from the additional iterations.

The second stopping condition, while more computationally intensive, often produces more refined and stable results. This is particularly evident in scenarios where RGB + location features are used, as it better handles the added complexity of spatial information.

By comparing these two stopping conditions, we gain insights into the trade-offs between computational efficiency and segmentation quality. Both methods have their advantages, and the choice depends on the specific requirements of the application.

## Incorporating RGB and Location for Improved Segmentation

### How Does Including RGB + Location (i, j) Change the Results?

Adding spatial information (i, j) to the feature vector, alongside RGB values, significantly enhances segmentation quality. While RGB-only clustering relies solely on color similarity, the inclusion of spatial information ensures that clusters are not only color-coherent but also spatially continuous.

Key observations from the results:

- **RGB-only Segmentation** :
  - Clusters are formed purely based on color similarity.
  - This can result in spatially disjoint regions, especially for objects with varying lighting conditions or subtle color gradients (parts of the frog's body or the petals).
- **RGB + Location Segmentation** :
  - The additional spatial component (i, j) introduces a bias toward grouping pixels that are close to each other in space.
  - The frog's body and the flower petals are segmented more cohesively, producing more visually meaningful clusters.
  - Spatial coherence reduces "noise" where pixels with similar colors but distant locations might otherwise be incorrectly grouped.

### How Can We Balance the Influence of Colors and Locations in the Image?

The balance between RGB values and spatial information is controlled by the weighting factor lambda in the distance calculation:

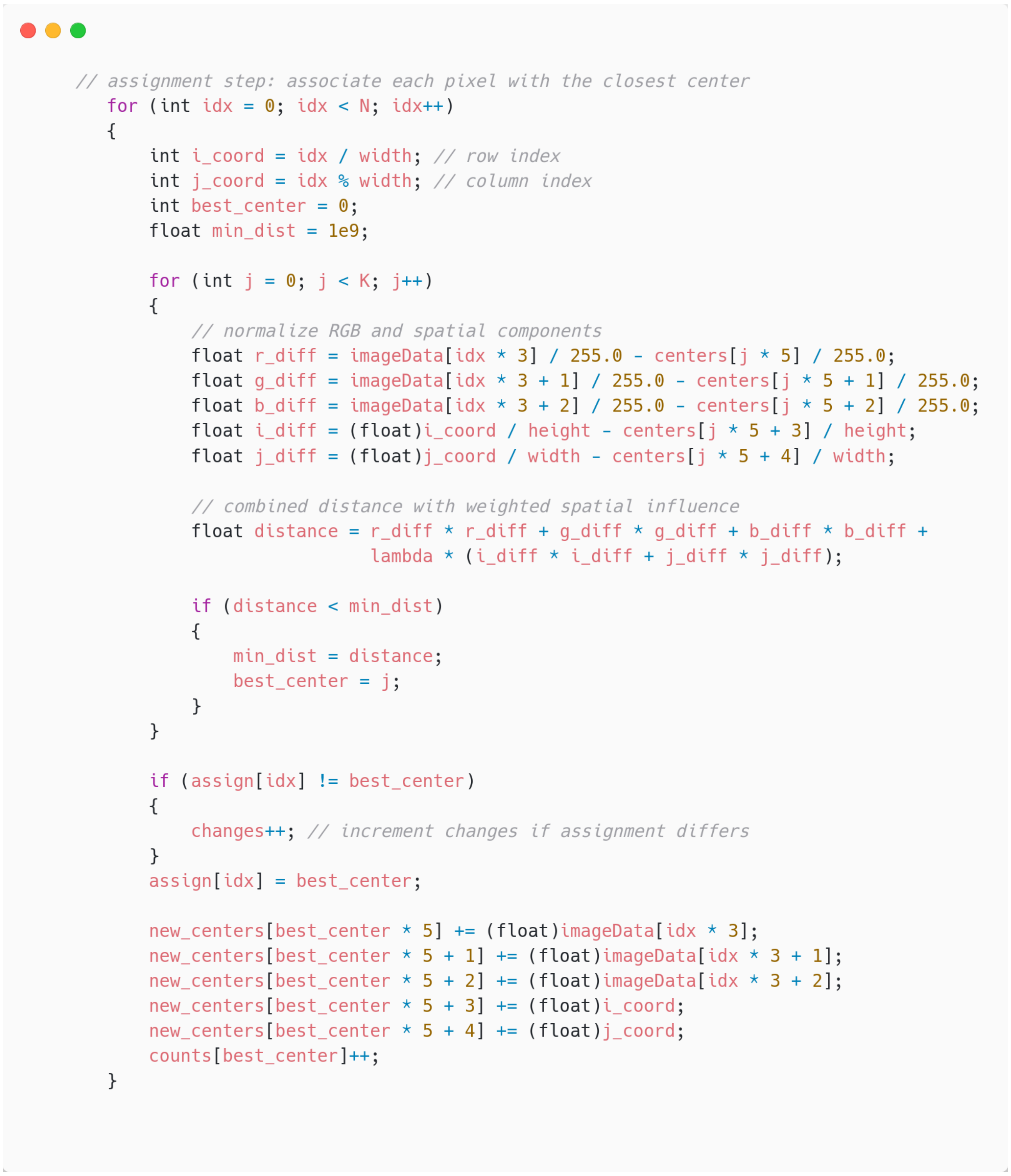
Distance = (Delta R)^2 + (Delta G)^2 + (Delta B)^2 + lambda * ((Delta i)^2 + (Delta j)^2),

where:

- (Delta R, Delta G, Delta B) represent differences in color values.
- (Delta i, Delta j) represent spatial differences in pixel coordinates.
- lambda determines the relative importance of spatial proximity versus color similarity.

**Impact of lambda:**

- **Low lambda values** ( lambda = 0.2):
  - Segmentation prioritizes color similarity.
  - Best suited for images with distinct color-based segmentation regions.
- **High lambda values** ( lambda = 0.5 or higher):
  - Segmentation prioritizes spatial proximity.
  - Ensures spatial coherence but can blur color-based distinctions, especially for objects with subtle color variations.



```
// assignment step: associate each pixel with the closest center
   for (int idx = 0; idx < N; idx++)
   {
       int i_coord = idx / width; // row index
       int j_coord = idx % width; // column index
       int best_center = 0;
       float min_dist = 1e9;

       for (int j = 0; j < K; j++)
       {
           // normalize RGB and spatial components
           float r_diff = imageData[idx * 3] / 255.0 - centers[j * 5] / 255.0;
           float g_diff = imageData[idx * 3 + 1] / 255.0 - centers[j * 5 + 1] / 255.0;
           float b_diff = imageData[idx * 3 + 2] / 255.0 - centers[j * 5 + 2] / 255.0;
           float i_diff = (float)i_coord / height - centers[j * 5 + 3] / height;
           float j_diff = (float)j_coord / width - centers[j * 5 + 4] / width;

           // combined distance with weighted spatial influence
           float distance = r_diff * r_diff + g_diff * g_diff + b_diff * b_diff +
                            lambda * (i_diff * i_diff + j_diff * j_diff);

           if (distance < min_dist)
           {
               min_dist = distance;
               best_center = j;
           }
       }

       if (assign[idx] != best_center)
       {
           changes++; // increment changes if assignment differs
       }
       assign[idx] = best_center;

       new_centers[best_center * 5] += (float)imageData[idx * 3];
       new_centers[best_center * 5 + 1] += (float)imageData[idx * 3 + 1];
       new_centers[best_center * 5 + 2] += (float)imageData[idx * 3 + 2];
       new_centers[best_center * 5 + 3] += (float)i_coord;
       new_centers[best_center * 5 + 4] += (float)j_coord;
       counts[best_center]++;
   }
```

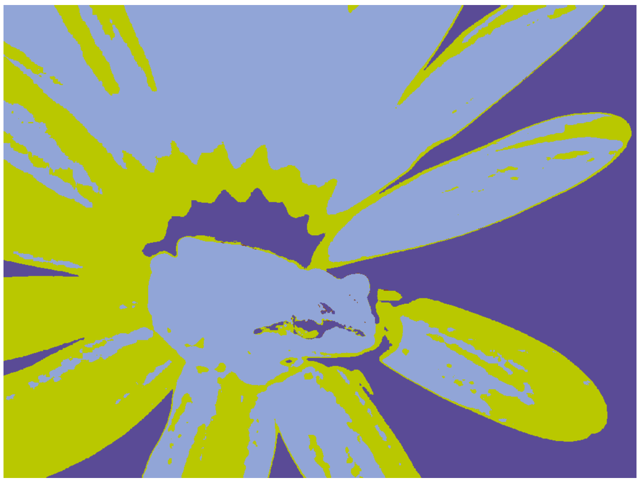
**Fig 1.10** Code snippet of the K-mean algorithm that considers both RGB and location (i,j)

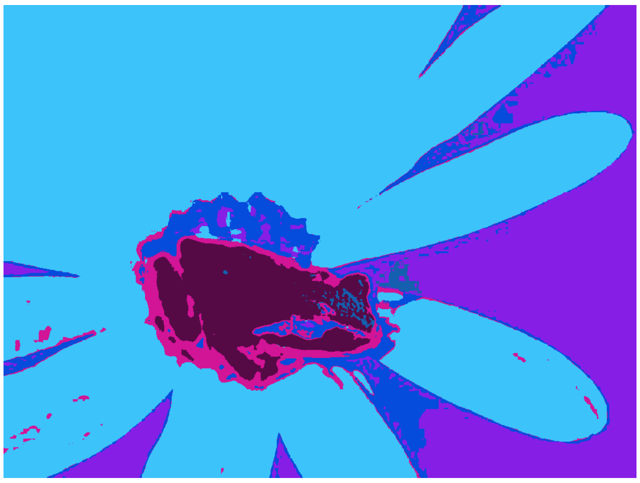
**Examples of Results**

To illustrate, the frog and pepper images were segmented using various lambda values:

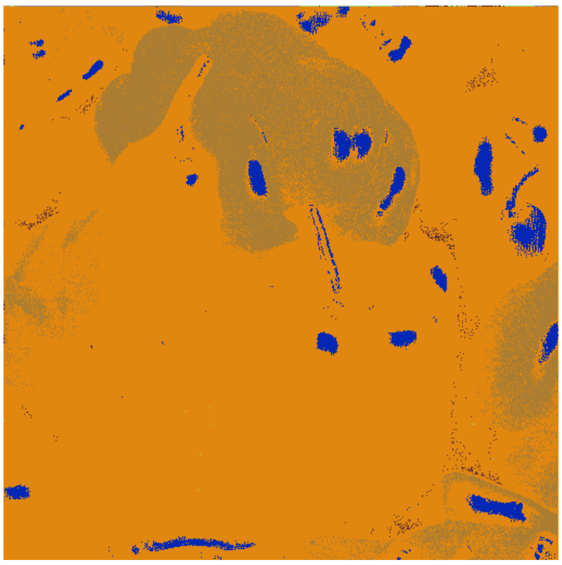
1. **Frog Image**:
   - With lambda = 0.2 (left), the segmentation focuses more on color similarity, capturing the frog's distinct colors but with minor spatial inconsistencies.

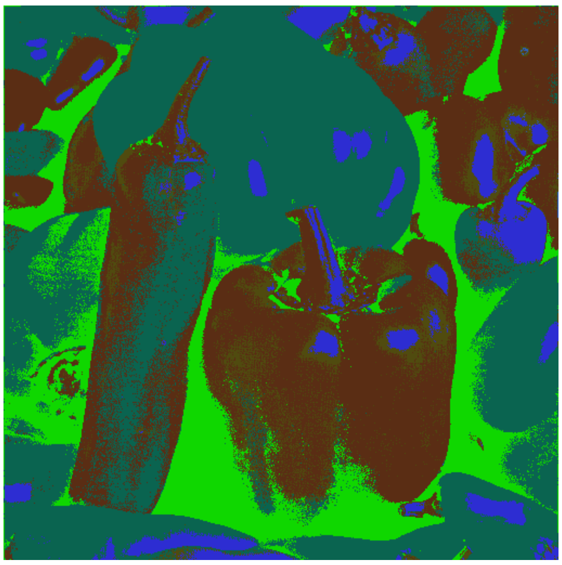
- With lambda = 0.5 (right), the segmentation becomes more spatially coherent, grouping nearby pixels effectively, especially along the frog's body and the flower petals.





2. **Pepper Image**:
   - At lambda = 0.2 (left), segmentation captures the distinct colors of the peppers but struggles to group spatially coherent regions.
   - At lambda = 0.5 (right), the segmentation better distinguishes spatial regions like individual peppers while maintaining reasonable color accuracy.





The inclusion of RGB + location (i, j) enhances segmentation by enforcing spatial coherence. The weighting factor lambda plays a critical role in balancing color similarity and spatial proximity, making it a key parameter for achieving meaningful segmentation results.

## Conclusion

In this practical work, we explored the K-means clustering algorithm for image segmentation, focusing on its implementation and the impact of various parameters. Through experimentation, we observed how the number of clusters (K), the initial values of cluster centers, and different stopping conditions influence the segmentation results. Additionally, incorporating spatial information (i,j) alongside RGB values significantly improved spatial coherence, especially when balanced with an appropriate weighting factor (λ).

The experiments demonstrated that while RGB-only segmentation is effective for color-based clustering, the addition of spatial features ensures better region consistency, making it particularly useful for images with gradual color transitions or complex patterns. Choosing the right balance between color and spatial information, as well as the appropriate stopping condition, is crucial for achieving meaningful segmentation results.

This study highlights the flexibility and limitations of K-means in image processing and provides insights into parameter tuning for effective segmentation.