Feature in image processing

In image processing and computer vision, a feature refers to a distinct and informative piece of data or characteristic that can be extracted from an image. Features are essential because they capture the key patterns, structures, or details within an image that are relevant for tasks such as classification, recognition, matching, and segmentation. Essentially, features serve as a condensed representation of the image's content, allowing algorithms to process and analyze the image efficiently.

What types of features an image can contain

Features can represent various properties of the image, such as intensities or colors, which are the raw pixel values in grayscale or color images. For example, the intensity of pixels in a grayscale image or the RGB (Red, Green, Blue) values in a color image can serve as basic features. Edges, another key feature, represent boundaries where there is a significant change in pixel intensity, often corresponding to object contours and structural transitions. Corners and key points, where multiple edges meet or there is a sharp change in direction, are distinctive features commonly used in image matching and object recognition. Textures refer to repetitive patterns or structures in an image, capturing the smoothness, roughness, or pattern of regions. Shape-based features describe the geometric properties of objects, such as size and orientation, and are commonly applied in object recognition and segmentation tasks.

In addition to these, other important image features include blobs, which are uniform regions in an image with consistent properties, and moments, which are statistical descriptors of an image's shape, area, and other characteristics. Histograms of colors or intensities summarize the distribution of pixel values, making them useful for image comparison or detecting changes in brightness or color. Gradient-based features describe the rate of intensity change between neighboring pixels and help identify edges or boundaries within an image. Finally, scale-invariant features are robust to changes in scale or rotation and are particularly useful in tasks like object recognition where objects may appear at different sizes or orientations. Together, these features provide powerful tools for understanding and analyzing images.

importing two image data

```
[104]
image_fruits = 'fruits.jpg'
image_puppy = 'puppy.png'
```

importing necessary libraries

```
[102]
import numpy as np
from skimage.io import imread, imshow
import cv2
import matplotlib.pyplot as plt
from scipy.stats import skew
from skimage.feature import graycomatrix, graycoprops,local_binary_pattern
```

Image Loading and Conversion to RGB

```
[112]
image_rgb = cv2.imread(image_fruits)
image_rgb = cv2.cvtColor(image_rgb, cv2.COLOR_BGR2RGB)
```

Displaying the two image data

```
image1 = cv2.imread(image_fruits)
image2 = cv2.imread(image_puppy)

# Convert from BGR to RGB (OpenCV loads in BGR format)
image1_rgb = cv2.cvtColor(image1, cv2.COLOR_BGR2RGB)
image2_rgb = cv2.cvtColor(image2, cv2.COLOR_BGR2RGB)

# Set up the plot with 1 row and 2 columns
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

axes[0].imshow(image1_rgb)
axes[0].axis('off')
axes[0].set_title("Image 1")

axes[1].imshow(image2_rgb)
axes[1].axis('off')
axes[1].set_title("Image 2")
plt.tight_layout()
plt.show()

```

Image 1

Image 2

Feature Extraction Techniques of Image Data

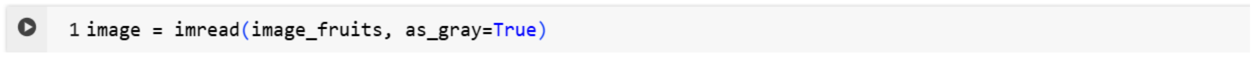
1. Grayscale Pixel Value Extraction

Grayscale Pixel Value Extraction is a technique in image processing where the pixel values of an image, converted to grayscale, are used as features for analysis, machine learning, or computer vision tasks. Grayscale images are simpler representations of color images, as they only contain intensity values ranging from black (0) to white (255), unlike color images that have separate intensity channels for Red, Green, and Blue (RGB). In grayscale images, each pixel is represented by a single intensity value, which encodes the brightness level of that pixel.

The main technique used in Grayscale Pixel Value Extraction involves converting the image into a grayscale format, where each pixel is represented by a single intensity value. These pixel values are then flattened into a 1D vector, transforming the 2D image into a single array of intensity values. This 1D vector can then be used as features for further analysis or fed into machine learning models for tasks such as classification, recognition, or segmentation, allowing efficient processing and pattern detection based on pixel intensity variations.

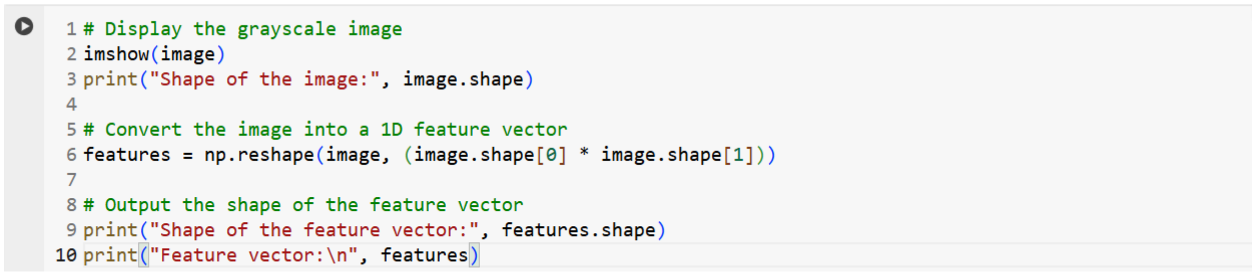
Grayscale Pixel Value Extraction simplifies image data by focusing on intensity values, reducing the complexity of color channels. This makes processing faster and more efficient, especially for tasks where color is unnecessary, like in medical imaging or industrial inspections. By extracting pixel intensity features, this technique aids in analyzing textures and structures, crucial for tasks such as object detection and pattern recognition, while ensuring computational efficiency.

Load the image in grayscale



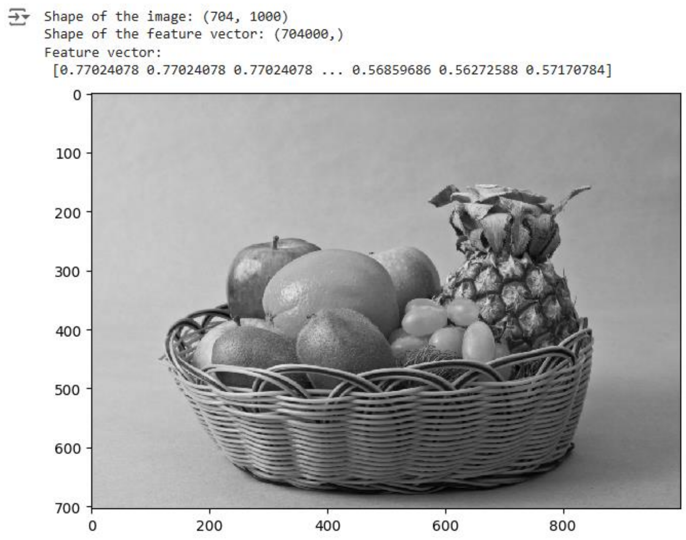
```python
image = imread(image_fruits, as_gray=True)
```

Display the grascalye image, Convert the image to 1D vector, display the Vector shape and number of elements(features)



```python
# Display the grayscale image
imshow(image)
print("Shape of the image:", image.shape)

# Convert the image into a 1D feature vector
features = np.reshape(image, (image.shape[0] * image.shape[1]))

# Output the shape of the feature vector
print("Shape of the feature vector:", features.shape)
print("Feature vector:\n", features)
```

Output



```
Shape of the image: (704, 1000)
Shape of the feature vector: (704000,)
Feature vector:
 [0.77024078 0.77024078 0.77024078 ... 0.56859686 0.56272588 0.57170784]
```

2. **Color and Intensity Based Features**

Color and Intensity Based Features refer to the properties extracted from an image that describe the distribution of pixel values in terms of both color (such as red, green, and blue channels) and intensity (brightness). These features are essential for understanding the composition of an image and play a key role in tasks like image classification, object recognition, and segmentation. Color features capture the dominant colors and their variations across the image, while intensity features focus on the brightness levels, which are important for detecting shapes, textures, and object boundaries.

Color Histogram

In this technique, a **color histogram** is computed for each color channel (Red, Green, and Blue) of an image, and the pixel intensity values for each channel are plotted to show the distribution of colors within the image. The process involves calculating the frequency of each intensity value (from 0 to 255) for each channel and normalizing it to create a normalized distribution. The histograms for each color channel are then visualized to give insights into the image's overall color composition. Additionally, a combined histogram representing the total intensity distribution across all channels is also created, allowing for an integrated view of the image's color and intensity information.

This technique is particularly useful for comparing images based on their color composition, identifying objects based on their color, and performing color-based segmentation tasks. By analyzing the histograms, we can understand the dominant colors in an image and how they are distributed, which is valuable for many computer vision tasks like image retrieval, object recognition, and tracking.

Calculate histograms for each channel

```python
hist_r = cv2.calcHist([image_rgb], [0], None, [256], [0, 256]).flatten()
hist_g = cv2.calcHist([image_rgb], [1], None, [256], [0, 256]).flatten()
hist_b = cv2.calcHist([image_rgb], [2], None, [256], [0, 256]).flatten()
```

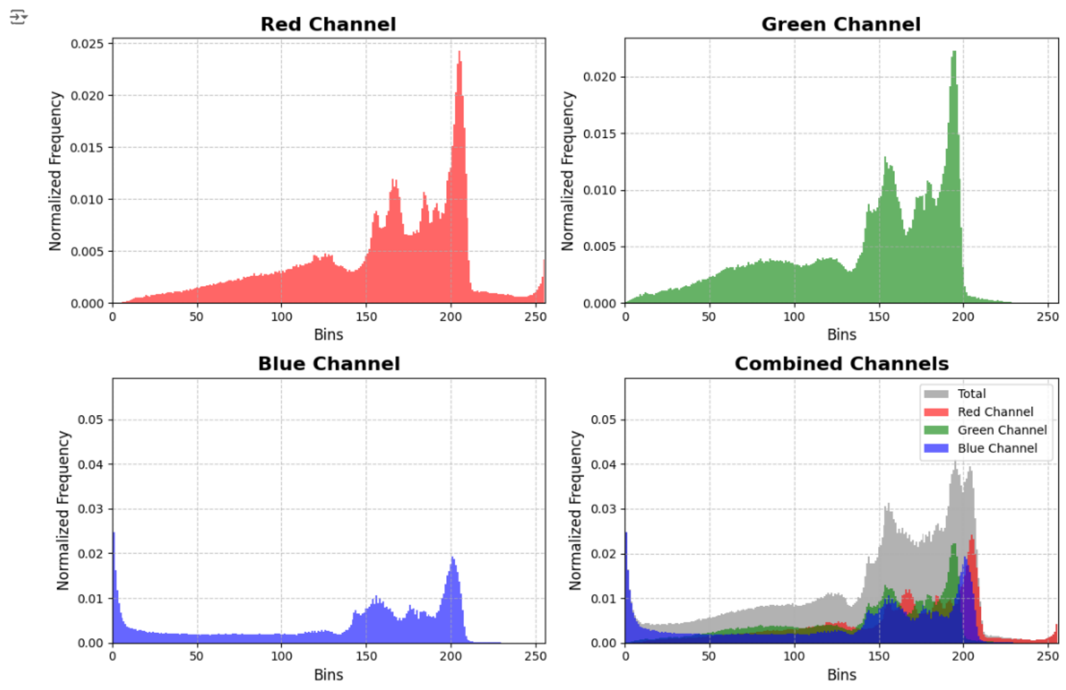
Normalize the histograms

```python
hist_r /= hist_r.sum()
hist_g /= hist_g.sum()
hist_b /= hist_b.sum()
```

Visualizing the histograms

```python
# Create a figure with two rows and two columns
fig, axs = plt.subplots(2, 2, figsize=(12, 8), facecolor="white")
fig.subplots_adjust(hspace=0.4, wspace=0.4)

# Plot individual Red channel in the top-left subplot
axs[0, 0].bar(range(256), hist_r, color="r", width=1.0, alpha=0.6)
axs[0, 0].set_xlim([0, 256])
axs[0, 0].set_title("Red Channel", fontsize=16, fontweight="bold")
axs[0, 0].set_xlabel("Bins", fontsize=12)
axs[0, 0].set_ylabel("Normalized Frequency", fontsize=12)
axs[0, 0].grid(True, linestyle="--", alpha=0.7)

# Plot individual Green channel in the top-right subplot
axs[0, 1].bar(range(256), hist_g, color="g", width=1.0, alpha=0.6)
axs[0, 1].set_xlim([0, 256])
axs[0, 1].set_title("Green Channel", fontsize=16, fontweight="bold")
axs[0, 1].set_xlabel("Bins", fontsize=12)
axs[0, 1].set_ylabel("Normalized Frequency", fontsize=12)
axs[0, 1].grid(True, linestyle="--", alpha=0.7)

# Plot individual Blue channel in the bottom-left subplot
axs[1, 0].bar(range(256), hist_b, color="b", width=1.0, alpha=0.6)
axs[1, 0].set_xlim([0, 256])
axs[1, 0].set_title("Blue Channel", fontsize=16, fontweight="bold")
axs[1, 0].set_xlabel("Bins", fontsize=12)
axs[1, 0].set_ylabel("Normalized Frequency", fontsize=12)
axs[1, 0].grid(True, linestyle="--", alpha=0.7)

# Plot combined histogram for all three channels in the bottom-right subplot
hist_total = hist_r + hist_g + hist_b
axs[1, 1].bar(range(256), hist_total, color="gray", width=1.0, alpha=0.6, label="Total")
axs[1, 1].bar(range(256), hist_r, color="r", width=1.0, alpha=0.6, label="Red Channel")
axs[1, 1].bar(range(256), hist_g, color="g", width=1.0, alpha=0.6, label="Green Channel")
axs[1, 1].bar(range(256), hist_b, color="b", width=1.0, alpha=0.6, label="Blue Channel")
axs[1, 1].set_xlim([0, 256])
axs[1, 1].set_title("Combined Channels", fontsize=16, fontweight="bold")
axs[1, 1].set_xlabel("Bins", fontsize=12)
axs[1, 1].set_ylabel("Normalized Frequency", fontsize=12)
axs[1, 1].legend(loc="upper right", fontsize=10)
axs[1, 1].grid(True, linestyle="--", alpha=0.7)

# Tight layout for better spacing
plt.tight_layout()

# Display the histograms
plt.show()

```

Output



Color Moments

Color Moments are statistical measures used to describe the distribution of colors in an image, providing a compact representation of its color properties. The most commonly used color moments are the mean, standard deviation, and skewness, each capturing unique characteristics of the color distribution. The **mean** represents the average color intensity in a channel, giving an overall sense of the image's brightness or color dominance. The **standard deviation** quantifies the spread or variation in color intensity, indicating how diverse or uniform the colors are. The **skewness** measures the asymmetry of the color distribution, reflecting whether the pixel intensity values are biased toward darker or lighter tones. These moments are computed for each color channel (Red, Green, and Blue), providing a detailed yet compact summary of the image's color profile.

Mean: Mean can be understood as the average color value in the image.

$$E_i = \frac{1}{N}\sum_{j=1}^{N} p_{ij}$$

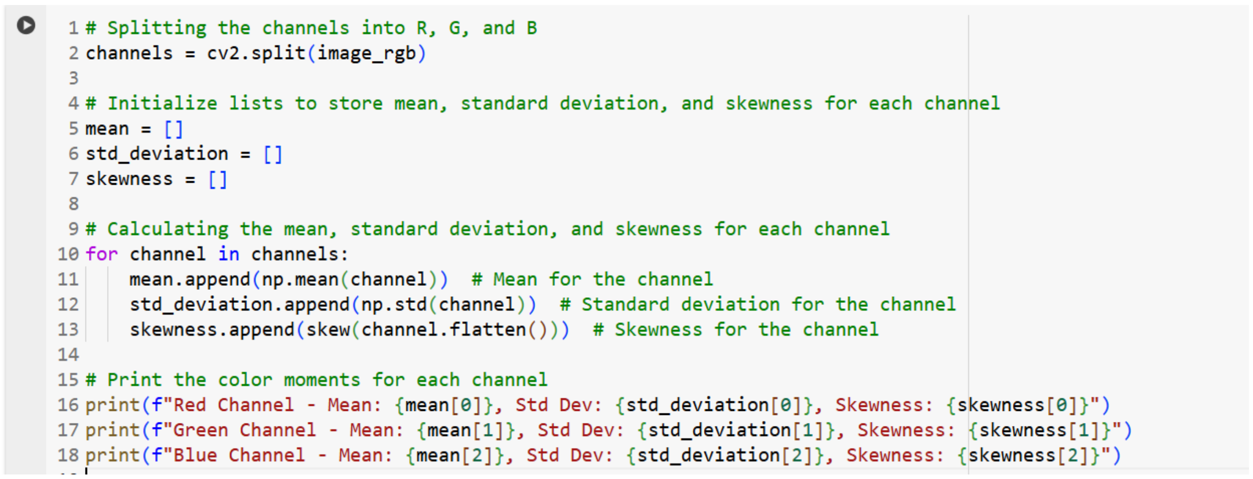
Standard Deviation: The standard deviation is the square root of the variance of the distribution.

$$\sigma_i = \sqrt{\frac{1}{N}\sum_{j=1}^{N}\left(p_{ij} - E_i\right)^2}$$

Skewness: Skewness can be understood as a measure of the degree of asymmetry in the distribution.

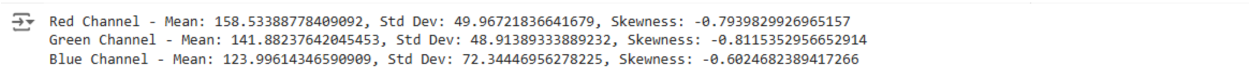
$$s_i = \sqrt[3]{\frac{1}{N}\sum_{j=1}^{N}\left(p_{ij} - E_i\right)^3}$$

Color moments are important because they are simple yet effective features for tasks such as image retrieval, classification, and comparison. Unlike high-dimensional pixel-based features, color moments offer a concise representation that reduces computational complexity while retaining the essence of the image's color distribution. They are particularly useful in comparing images for similarity, as they enable algorithms to differentiate images based on their overall color characteristics rather than pixel-by-pixel analysis. For example, in content-based image retrieval systems, color moments are often used to match and retrieve images with similar color compositions, making them a powerful tool for tasks that rely on color-based analysis.



```python
# Splitting the channels into R, G, and B
channels = cv2.split(image_rgb)

# Initialize lists to store mean, standard deviation, and skewness for each channel
mean = []
std_deviation = []
skewness = []

# Calculating the mean, standard deviation, and skewness for each channel
for channel in channels:
    mean.append(np.mean(channel))  # Mean for the channel
    std_deviation.append(np.std(channel))  # Standard deviation for the channel
    skewness.append(skew(channel.flatten()))  # Skewness for the channel

# Print the color moments for each channel
print(f"Red Channel - Mean: {mean[0]}, Std Dev: {std_deviation[0]}, Skewness: {skewness[0]}")
print(f"Green Channel - Mean: {mean[1]}, Std Dev: {std_deviation[1]}, Skewness: {skewness[1]}")
print(f"Blue Channel - Mean: {mean[2]}, Std Dev: {std_deviation[2]}, Skewness: {skewness[2]}")
```

Output



```
Red Channel - Mean: 158.53388778409092, Std Dev: 49.96721836641679, Skewness: -0.7939829926965157
Green Channel - Mean: 141.88237642045453, Std Dev: 48.91389333889232, Skewness: -0.8115352956652914
Blue Channel - Mean: 123.99614346590909, Std Dev: 72.34446956278225, Skewness: -0.6024682389417266
```

3. Edge Detection

Edge detection is a fundamental feature extraction technique used to identify boundaries and significant transitions in intensity within an image. It highlights regions where pixel intensity changes abruptly, typically corresponding to object edges or structural details. This is essential for understanding the shape, structure, and composition of objects in an image. Edge detection methods are often used as a preprocessing step in computer vision tasks like object recognition, image segmentation, and feature extraction. Popular techniques for edge detection include Sobel, Prewitt, and Canny algorithms, each offering unique advantages based on accuracy and efficiency.

Canny Edge Detection

Canny Edge Detection is one of the most widely used and robust edge detection techniques. Developed by John F. Canny, it is designed to detect edges in an image by optimizing the signal-to-noise ratio and ensuring accurate localization of edges. The algorithm consists of multiple steps:

1. Gaussian Smoothing

• Smooth the image to reduce noise using a Gaussian filter:

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

• Where σ is the standard deviation, and \$ x\$ and \$ y\$ are pixel coordinates

2. Gradient Calculation

• Compute the gradients \$ G_x\$ and \$ G_y\$ using Sobel operators:

$$G_x = \frac{\partial I}{\partial x}, \quad G_y = \frac{\partial I}{\partial y}$$

• Calculate the gradient magnitude \$ G\$ and direction θ:

$$G = \sqrt{G_x^2 + G_y^2}, \quad \theta = \arctan\left(\frac{G_y}{G_x}\right)$$

3. Non-Maximum Suppression

• Thin the edges by suppressing all non-maximum points in the gradient direction: – Compare each pixel's gradient magnitude to its neighbors along the gradient direction. Keep the pixel if it is the maximum.

4. Double Thresholding

• Apply high (TH) and low (TL) thresholds:

$$\text{Strong edges: } G > T_H, \quad \text{Weak edges: } T_L < G \le T_H, \quad \text{Suppress: } G \le T_L$$

5. Edge Tracking by Hysteresis

• Connect weak edges to strong edges if they are connected:

– Strong edges are retained.

– Weak edges connected to strong edges are also retained, otherwise suppressed.

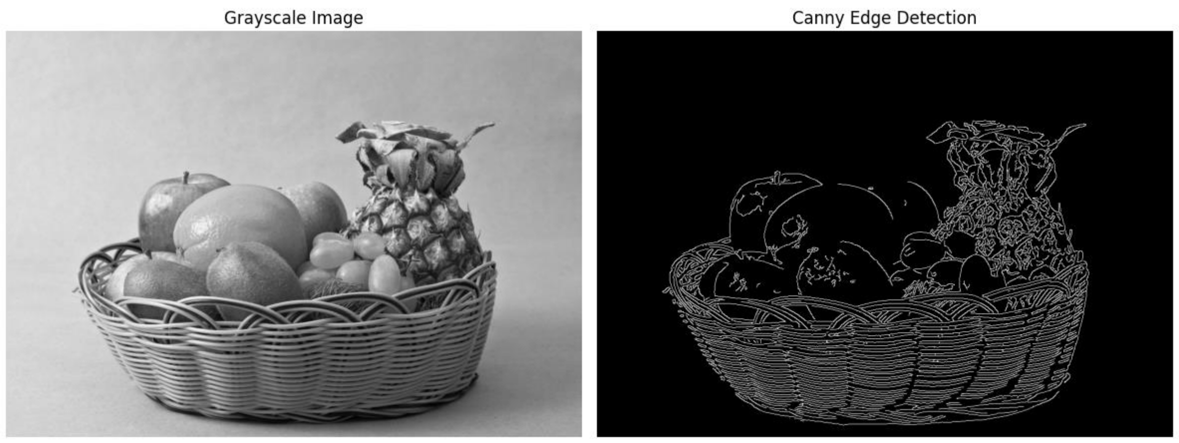
Read the image in grayscale and apply Gaussian blur

```python
image_gray = cv2.imread(image_fruits, cv2.IMREAD_GRAYSCALE)
# Apply Gaussian blur to reduce noise
image_blurred = cv2.GaussianBlur(image_gray, (5, 5), 0)
```

Apply Canny edge detection and display the image

```python
# Apply Canny edge detection
edges = cv2.Canny(image_blurred, 50, 100)

# Display original grayscale image and the detected edges
fig, axs = plt.subplots(1, 2, figsize=(12, 6))

# Original grayscale image
axs[0].imshow(image_gray, cmap='gray')
axs[0].set_title("Grayscale Image")
axs[0].axis('off')

# Canny edge detection result
axs[1].imshow(edges, cmap='gray')
axs[1].set_title("Canny Edge Detection")
axs[1].axis('off')

plt.tight_layout()
plt.show()
```

Output



Corner Detection Feature Extraction

Corner detection is a feature extraction technique used to identify points in an image where two or more edges intersect, forming a corner. Corners are stable and distinctive features, making them ideal for tasks such as image matching, tracking, and object recognition. Corners have high variation in intensity in all directions, making them easy to distinguish from edges and flat regions. They are considered robust features as they remain invariant to transformations like rotation and scaling. Popular corner detection methods include Harris Corner Detection, Shi-Tomasi, and FAST, each offering different balances between accuracy and computational efficiency.

Harris Corner Detection Algorithm

The Harris Corner Detection algorithm is used to identify corner points in an image. Corners are defined as points where the image gradients change significantly in multiple directions. The Harris Corner Detection algorithm identifies corners by calculating the gradient products, applying Gaussian smoothing, computing a corner response function, and then performing thresholding and dilation to highlight the corners on the original image. The following steps outline the algorithm:

1. Convert Image to Grayscale

Convert the input image to grayscale using the following formula:

$$I(x,y) = 0.299 \cdot R(x,y) + 0.587 \cdot G(x,y) + 0.114 \cdot B(x,y)$$

Where $R(x,y)$, $G(x,y)$ and $B(x,y)$ are the red, green, and blue color channels, respectively.

2. Compute Image Gradients

The gradients in the x and y directions are calculated as:

$$I_x = \frac{\partial I}{\partial x}, \quad I_y = \frac{\partial I}{\partial y}$$

3. Compute Gradient Products

The products of the gradients are computed as:

$$I_x^2 = I_x \cdot I_x, \quad I_y^2 = I_y \cdot I_y, \quad I_xI_y = I_x \cdot I_y$$

4. Apply Gaussian Filter

Smooth the gradient products using a Gaussian filter:

$$M_{11} = \sum_{x,y} I_x^2 \cdot G(x,y), \quad M_{22} = \sum_{x,y} I_y^2 \cdot G(x,y), \quad M_{12} = \sum_{x,y} I_xI_y \cdot G(x,y)$$

Where $G(x,y)$ is the Gaussian kernel with standard deviation σ

5. Compute Corner Response

The Harris corner response RRR is calculated as:

$$R = \det(M) - k \cdot \left(\mathrm{trace}(M)\right)^2$$

Determinant:

$$\det(M) = M_{11} \cdot M_{22} - (M_{12})^2$$

Trace:

$$\mathrm{trace}(M) = M_{11} + M_{22}$$

6. Thresholding and Non-Maximum Suppression

Apply a threshold to filter strong corner responses:

$$\text{Corners: } R > \text{threshold} \times R_{\max}$$

Where $R_{\max}$ is the maximum value of RRR.

7. Dilate Corners

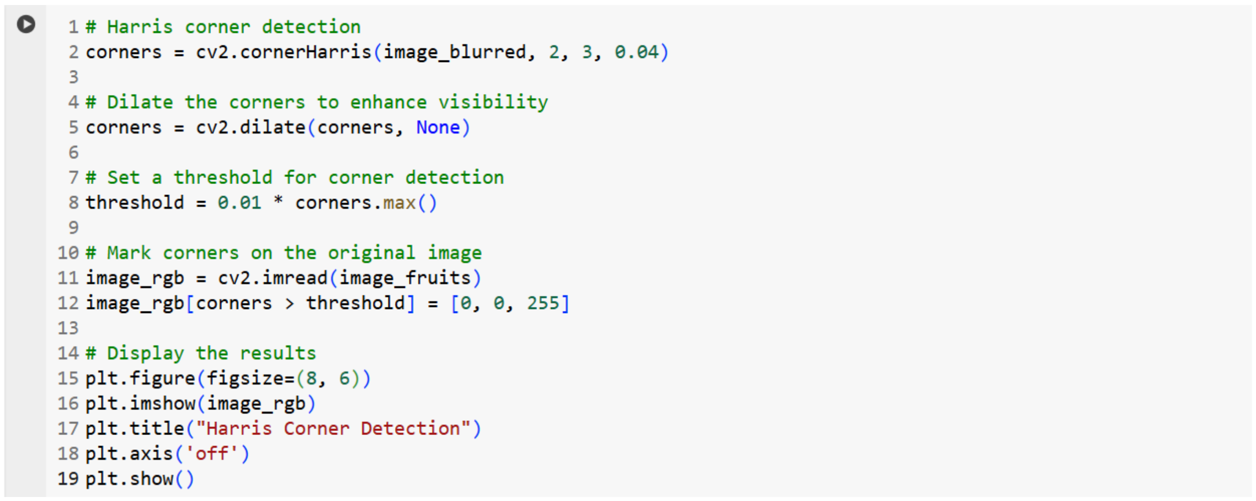
Dilate the corner response to enhance visibility:

$$\text{Dilated} = \text{dilate}(R, \text{None})$$

8. Overlay Corners on Original Image

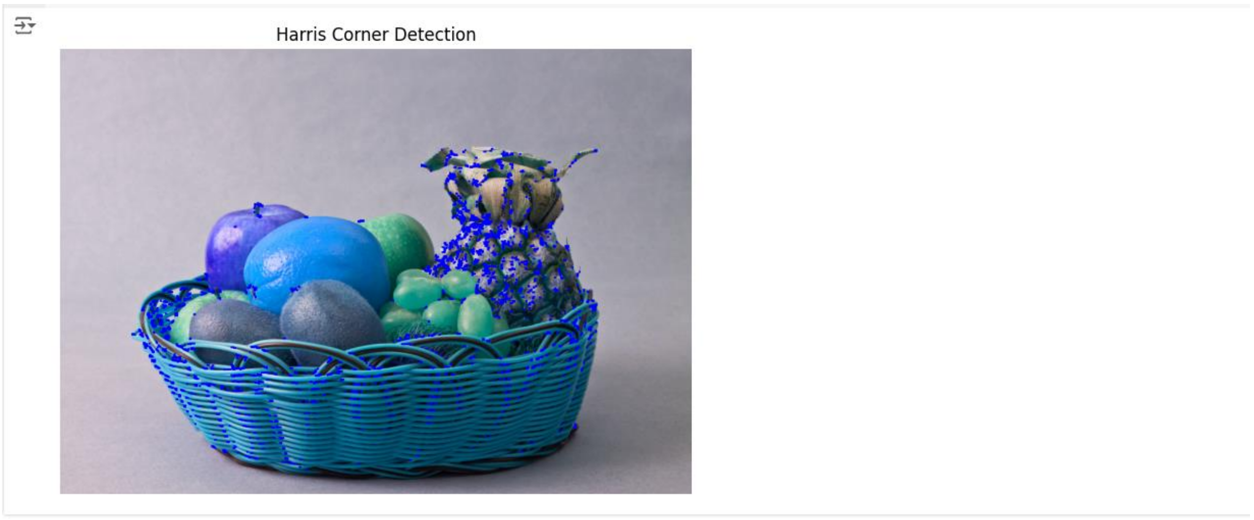
Overlay the detected corners on the original image by marking them :

$$\text{image}[R > 0.01 \times R_{\max}] = [0,0,255]$$



```python
# Harris corner detection
corners = cv2.cornerHarris(image_blurred, 2, 3, 0.04)

# Dilate the corners to enhance visibility
corners = cv2.dilate(corners, None)

# Set a threshold for corner detection
threshold = 0.01 * corners.max()

# Mark corners on the original image
image_rgb = cv2.imread(image_fruits)
image_rgb[corners > threshold] = [0, 0, 255]

# Display the results
plt.figure(figsize=(8, 6))
plt.imshow(image_rgb)
plt.title("Harris Corner Detection")
plt.axis('off')
plt.show()
```

Output



4. Blob Detection

Blob detection is a feature extraction technique used in image processing to identify and analyze regions in an image that differ in properties such as intensity, color, or texture compared to the surrounding areas. These regions, called "blobs," are typically circular or elliptical and represent features of interest like keypoints, objects, or irregularities in the image. Blob detection is widely used in tasks like object recognition, feature matching, and medical imaging, where identifying distinct regions or patterns is critical.

Blob detection works by analyzing variations in pixel intensity and identifying regions that exhibit consistency or significant change. Popular methods for blob detection include **Laplacian of Gaussian (LoG)**, **Difference of Gaussian (DoG)**, and **SimpleBlobDetector** in OpenCV. These methods detect blobs at multiple scales, making them robust to changes in size, shape, and orientation. The detected blobs can then be used as input features for further analysis or machine learning tasks. Blob detection is particularly valuable for identifying keypoints in images with complex structures, such as biological cells, astronomical data, or industrial parts.

Here's a step-by-step overview of the blob detection process:

1. Convert Image to Grayscale

Convert the input image to grayscale to simplify processing. The formula for grayscale conversion is:

$$I(x,y) = 0.299 \cdot R(x,y) + 0.587 \cdot G(x,y) + 0.114 \cdot B(x,y)$$

Where:

- $R(x,y)$ Red channel intensity

- $G(x, y)$ Green channel intensity
- $B(x, y)$ Blue channel intensity

2. Create Blob Detector

Use OpenCV's SimpleBlobDetector to detect blobs in the image. This internally uses the **Difference of Gaussians (DoG)** method. The detector is initialized as:

detector = cv2.SimpleBlobDetector_create()

3. Detect Blobs

Apply the blob detector to the grayscale image to identify keypoints representing blobs:

keypoints = detector.detect(image_gray)

4. Draw Keypoints

Draw the detected blobs on the original image to highlight them visually:

blob_image = cv2.drawKeypoints(image, keypoints, np.array([]), (0, 0, 255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

Explanation of Parameters:

- image: The original image on which blobs will be highlighted.
- keypoints: The detected blobs.
- np.array([]): An empty array for the color.
- (0, 0, 255): Color for the blobs (red in BGR format).
- cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS: Flag to draw blobs with size information.

3.1 Difference of Gaussians (DoG) Method

The DoG method identifies blobs by subtracting two Gaussian-blurred versions of the image.

1. Apply Gaussian Filters

Smooth the image using two Gaussian filters with different standard deviations (σ1\sigma_1σ1 and σ2\sigma_2σ2):

$$G_1(x, y) = \frac{1}{2\pi\sigma_1^2} e^{-\frac{x^2+y^2}{2\sigma_1^2}}$$

$$G_2(x,y) = \frac{1}{2\pi\sigma_2^2} e^{-\frac{x^2+y^2}{2\sigma_2^2}}$$

2. Compute Difference

Subtract the two Gaussian-blurred images to highlight regions with significant intensity changes:

$$\mathrm{DoG}(x,y) = (I * G_1)(x,y) - (I * G_2)(x,y)$$

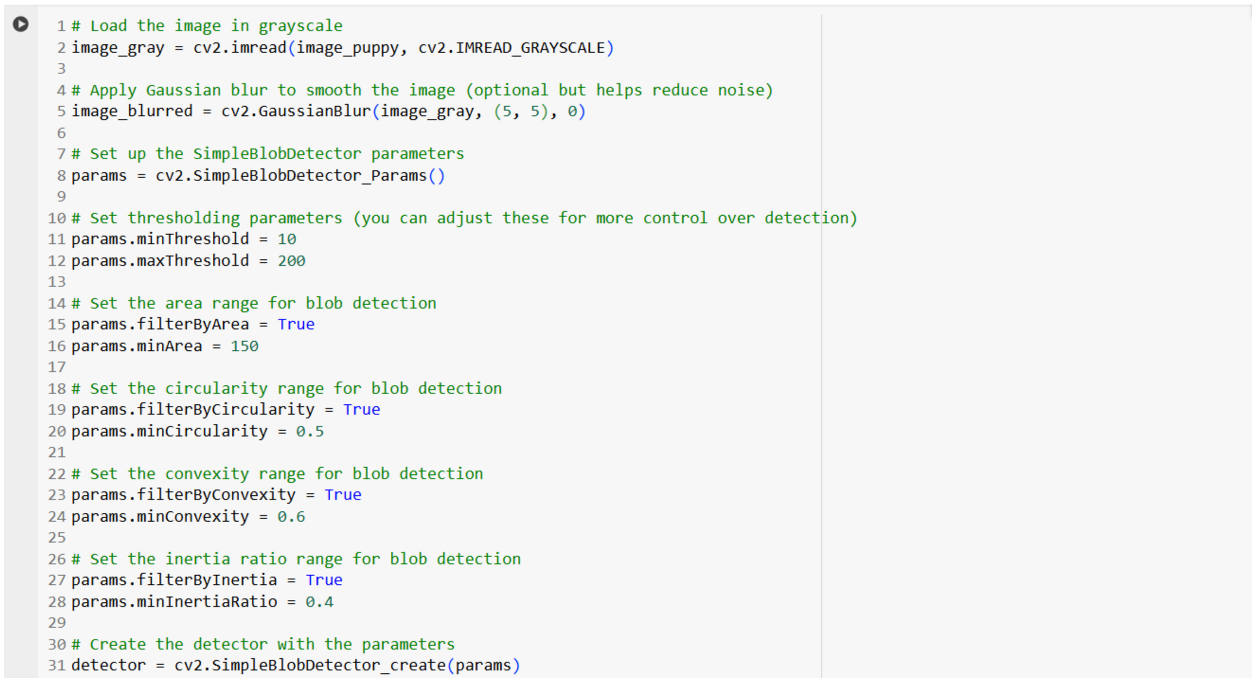
Where $(I * G)$ represents the convolution of the image III with the Gaussian kernel G.

3. Detect Local Extrema

Find local extrema in the DoG image to identify blobs. These extrema represent regions where the intensity difference between the two Gaussian-blurred images is maximized.

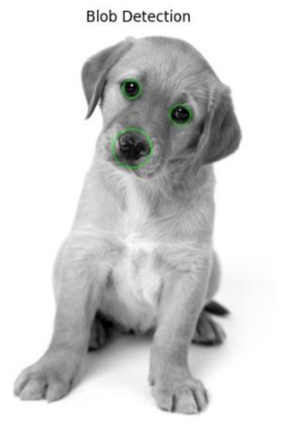
This step involves:

- Identifying local maxima (bright blobs).
- Identifying local minima (dark blobs).



```
# Load the image in grayscale
image_gray = cv2.imread(image_puppy, cv2.IMREAD_GRAYSCALE)

# Apply Gaussian blur to smooth the image (optional but helps reduce noise)
image_blurred = cv2.GaussianBlur(image_gray, (5, 5), 0)

# Set up the SimpleBlobDetector parameters
params = cv2.SimpleBlobDetector_Params()

# Set thresholding parameters (you can adjust these for more control over detection)
params.minThreshold = 10
params.maxThreshold = 200

# Set the area range for blob detection
params.filterByArea = True
params.minArea = 150

# Set the circularity range for blob detection
params.filterByCircularity = True
params.minCircularity = 0.5

# Set the convexity range for blob detection
params.filterByConvexity = True
params.minConvexity = 0.6

# Set the inertia ratio range for blob detection
params.filterByInertia = True
params.minInertiaRatio = 0.4

# Create the detector with the parameters
detector = cv2.SimpleBlobDetector_create(params)
```

```python

# Detect blobs in the image
keypoints = detector.detect(image_blurred)

# Draw detected blobs on the image (green circles)
image_with_blobs = cv2.drawKeypoints(image_gray, keypoints, np.array([]), (0, 255, 0),
                                     flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

# Display the original image with the detected blobs
plt.figure(figsize=(8, 6))
plt.imshow(image_with_blobs)
plt.title("Blob Detection")
plt.axis('off')
plt.show()

```

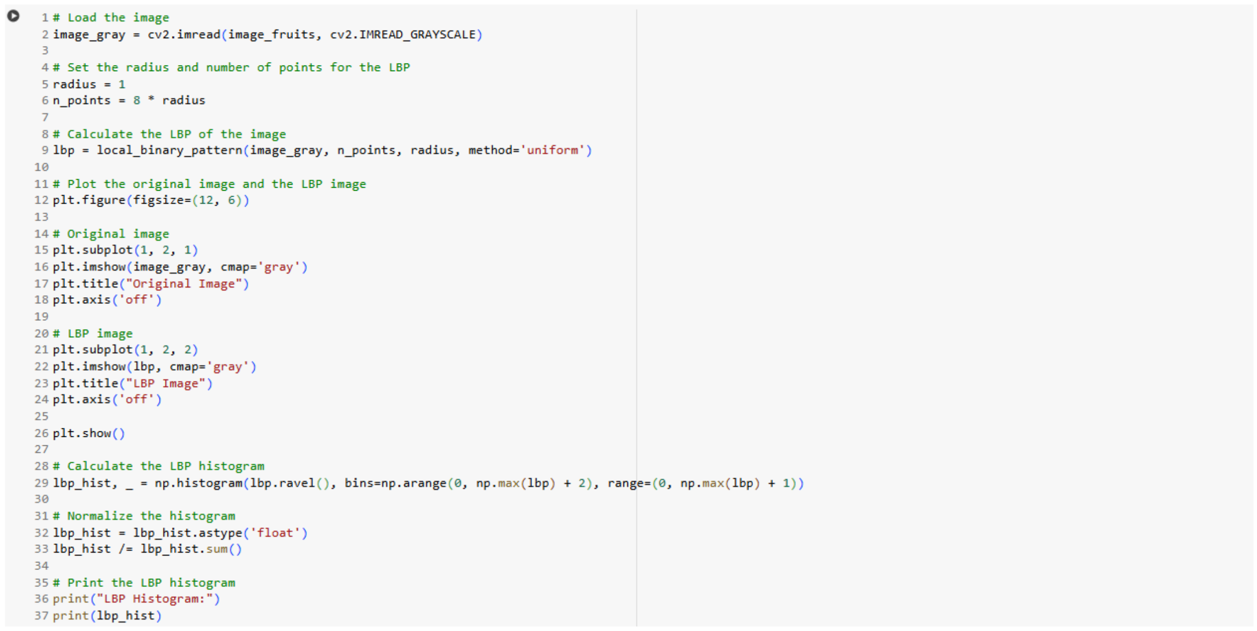


Local Binary Patterns (LBP)

Local Binary Patterns (LBP) is a powerful and efficient method for texture classification and feature extraction in image processing. It works by comparing the intensity of each pixel in an image to its neighboring pixels and assigning a binary code based on whether the neighboring pixel's intensity is greater or lesser than the central pixel. This technique effectively captures the spatial patterns or textures within an image, making it especially useful for analyzing textures and patterns in various applications such as face recognition, object detection, and image retrieval.

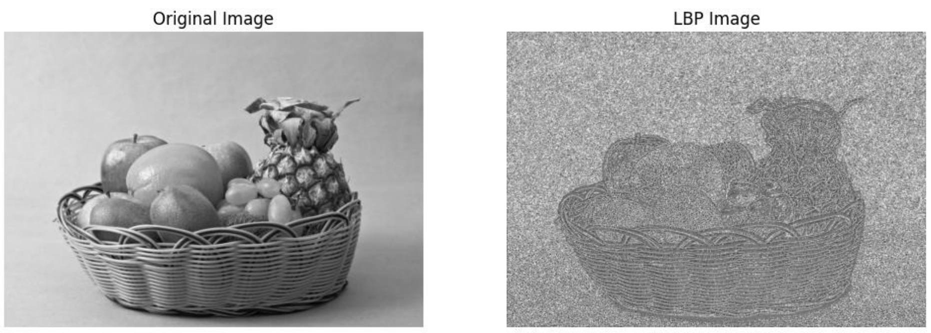
How LBP Works:

1. **Grayscale Conversion**: The first step in LBP is to convert the image into grayscale. This simplifies the analysis by focusing only on the intensity values, ignoring the color information. Each pixel in the image is represented by a single intensity value, typically ranging from 0 to 255.
2. **Neighborhood Comparison**: For each pixel, a small neighborhood of surrounding pixels is selected (usually 3x3 or 5x5). The intensity of each of these neighboring pixels is compared with the intensity of the central pixel. If the intensity of the neighbor is greater than or equal to the central pixel, it is assigned a value of 1; otherwise, it is assigned a value of 0. This results in a binary string for each pixel based on its local neighborhood.
3. **Binary Code Formation**: The binary values from the neighboring pixels are concatenated to form a unique binary code for the central pixel. The length of this binary code depends on the number of neighboring pixels considered, typically 8 neighbors (in a 3x3 neighborhood), resulting in an 8-bit binary number.
4. **Histogram of LBP Codes**: Once the binary codes for all pixels are computed, a histogram of the LBP codes is generated. The histogram represents the frequency of each unique LBP code in the image and serves as a compact feature representation of the image's texture.



```python
# Load the image
image_gray = cv2.imread(image_fruits, cv2.IMREAD_GRAYSCALE)

# Set the radius and number of points for the LBP
radius = 1
n_points = 8 * radius

# Calculate the LBP of the image
lbp = local_binary_pattern(image_gray, n_points, radius, method='uniform')

# Plot the original image and the LBP image
plt.figure(figsize=(12, 6))

# Original image
plt.subplot(1, 2, 1)
plt.imshow(image_gray, cmap='gray')
plt.title("Original Image")
plt.axis('off')

# LBP image
plt.subplot(1, 2, 2)
plt.imshow(lbp, cmap='gray')
plt.title("LBP Image")
plt.axis('off')

plt.show()

# Calculate the LBP histogram
lbp_hist, _ = np.histogram(lbp.ravel(), bins=np.arange(0, np.max(lbp) + 2), range=(0, np.max(lbp) + 1))

# Normalize the histogram
lbp_hist = lbp_hist.astype('float')
lbp_hist /= lbp_hist.sum()

# Print the LBP histogram
print("LBP Histogram:")
print(lbp_hist)
```

Output



```
LBP Histogram:
[0.03480682 0.05814347 0.02562784 0.09755114 0.16983949 0.16758949
 0.06562074 0.09928835 0.17561648 0.10591619]
```