1 Image processing for robotics

First, let us read an image saved on the hard disk and analyze its attributes. For this operation, we use the *imread*() function provided by the *OpenCV* library. Check for the output of the following code.

```
import cv2

img = cv2.imread('/workspaces/base_ros2/src/image_processing/images/
    denali_grayscale.jpg')

print(img.shape) # prints image dimensions
print(img.dtype) # Print the data type of the values

blue, green, red = img[:, :, 0], img[:, :, 1], img[:, :, 2]  # Split
    channels
print(blue.shape)  # dimensions of a single channel
```

The output of the function *imrea*() is a 3-dimensional NumPy array. These dimensions correspond to (width, height, channels). For a colored image, these channels will be BGR, meaning Blue, Green, and Red. Each channel will have values ranging from 0 to 255 and hence, their data type would be *uint*8.

The following image breaks down how a color image is broken down to its 3 channels.

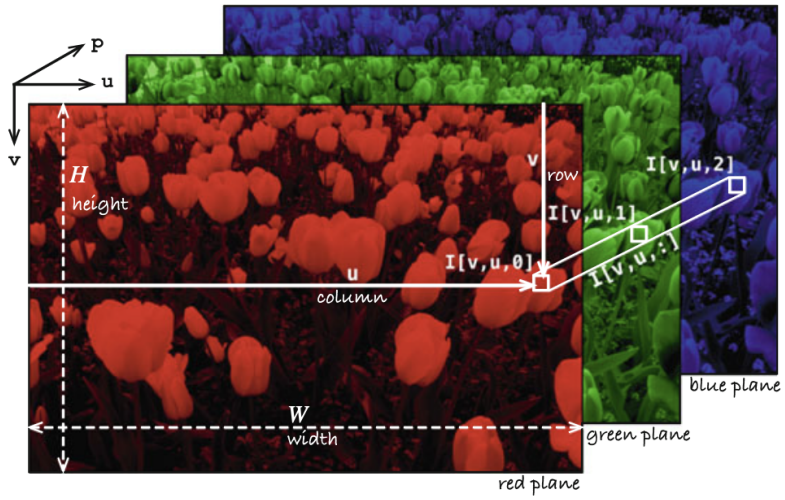


Fig. 11.3 Color image shown as a 3-dimensional structure with dimensions: row, column, and color plane

Self-learning assignment: save each dimension separately as an image.

For a grayscale image, there will be only one channel. OpenCV allows you to convert a BGR image to grayscale using the following code.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # Converts to grayscale
print(gray.shape) # Examin the dimensions
```

1.1 Image Slicing

Now that we know that we can access each element of the image, we can access parts of the image and save them separately as a new image. This is known as image slicing.

```
import cv2

img = cv2.imread('/workspaces/base_ros2/src/image_processing/images/denali.
    jpg')

sliced_img = img[0:200,0:200,:]

cv2.imwrite('sliced_image.jpg', sliced_img)
```

1.2 Pixel Value Distribution

Analyzing the distribution of the pixels can be useful in obtaining additional information about the quality of the image and the composition of a scene.

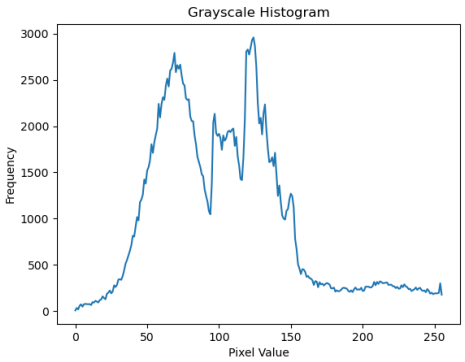
In a grayscale version of an image, let's analyze statistics of it and plot the pixel-value distribution as a histogram.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('/workspaces/base_ros2/src/image_processing/images/denali.
    jpg',cv2.IMREAD_GRAYSCALE)

hist = cv2.calcHist([img], [0], None, [256], [0, 256])
plt.plot(hist)
plt.title('Grayscale Histogram')
plt.xlabel('Pixel Value')
plt.ylabel('Frequency')
plt.savefig('histogram.png')
plt.close()
```

(a) Grayscale image



(b) Pixel-value histogram

One example of additional information that can be obtained from a pixel-value histogram is exposure. An underexposed image will have pixel values clustered closer to 0, while an overexposed image will have pixel values clustered closer to 255.

Looking at the histogram, one can observe several peaks in the histogram. The following code finds peaks in the histogram. There are a large number of peaks in a given histogram, which may not be useful. Hence, a set of filters can be applied to narrow the search and find peaks with the highest separation and height. The peaks in the histogram correspond to particular populations of pixels in the image. The lowest pixel value peak corresponds to the dark pixels and vise versa.

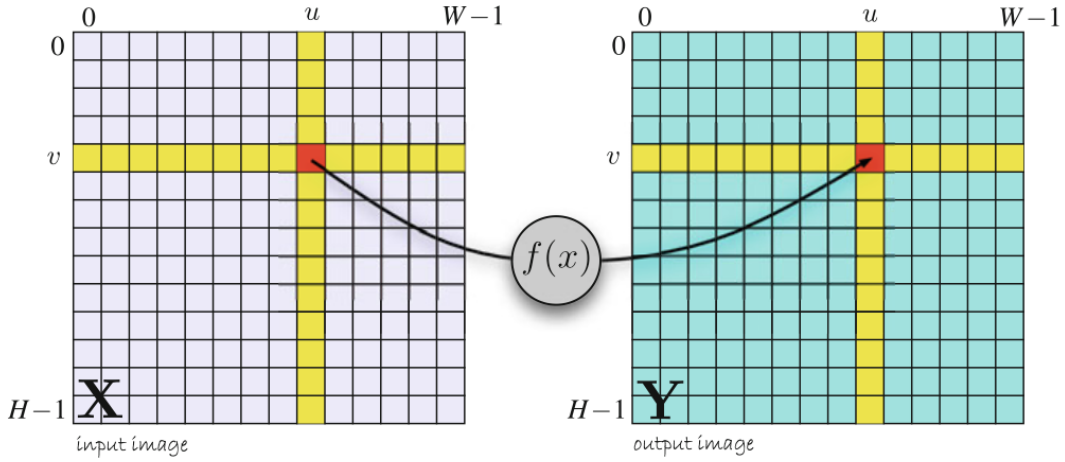
```
peaks, _ = find_peaks(hist.flatten()) # find all peaks
print(f"number of peaks: {peaks.size}")

peaks, _ = find_peaks(hist.flatten(), threshold=100, distance=50) # find
    peaks with filters
print(f"number of peaks after filters: {peaks.size}")
print(f"position of peaks: {peaks}")
```

1.3 Monadic Operations

Monadic operators are operators that are applied independently to each pixel in a single image to produce an output image,

$$Y_{u,v} = f(X_{u,v}), \forall (u,v) \in X.$$



One common operation is thresholding. This monadic operator separates the pixel values to 0 or 255 depending on the original pixel value is less than or greater than a user defined threshold.

```
# Set threshold
threshold = 200

# Apply binary thresholding
gray[gray > threshold] = 255
gray[gray <= threshold] = 0

# Save result
cv2.imwrite('monadic_threshold_binary.jpg', gray)
```

Another monadic operator will skew the pixel values to adjust for the over or under exposure. In the following example, we apply a non-linear operator to each pixel. Note that we normalize the image before applying the operator.

```
# Apply gamma correction to skew histogram towards 255
gamma = 0.5  # Gamma < 1 brightens image
img_normalized = gray / 255.0 # Normalize the image
skewed = np.power(img_normalized, gamma) * 255.0
skewed = skewed.astype(np.uint8)

# Save the skewed image
```

```
cv2.imwrite('skewed_gray.jpg', skewed)
```

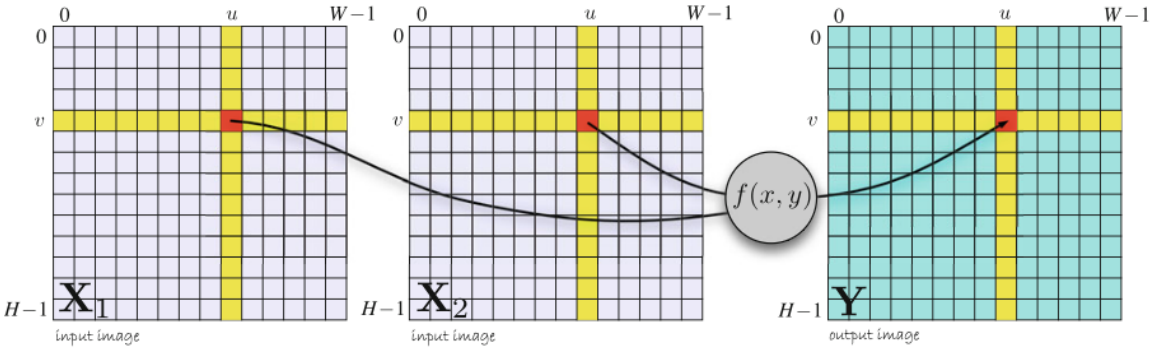
Examine the histogram of its new pixel-value distribution.

```
hist = cv2.calcHist([skewed], [0], None, [256], [0, 256])
plt.plot(hist)
plt.title('Skewed Histogram')
plt.xlabel('Pixel Value')
plt.ylabel('Frequency')
plt.savefig('histogram_skewed_gray.png')
plt.close()
```

1.4 Dyadic Operations

These operators take two images of similar size and creqte a thrid image. Every pixel in the third image corresponds to the operator-applied pixels in the two input images.

$$Yu, v = f(X_{1:u,v}, X_{2:u,v}), \forall (u, v) \in X_1, X_2.$$



Examples of these operators include binary arithmetic operators such as addition, subtraction, element-wise multiplication, or NumPy dyadic array functions such as *max*, *min*, and *atan*2.