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.

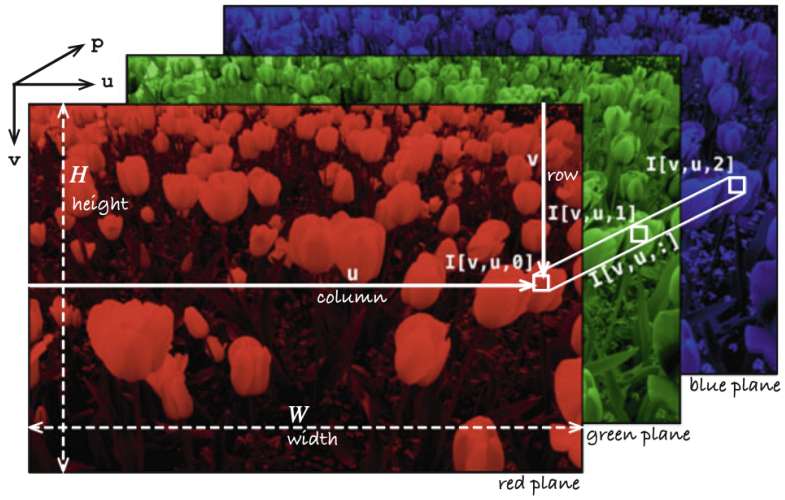


Fig. 11.3 Color image shown as a 3-dimensional structure with dimensions: row, column, and color plane

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
```

0.1 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)
```

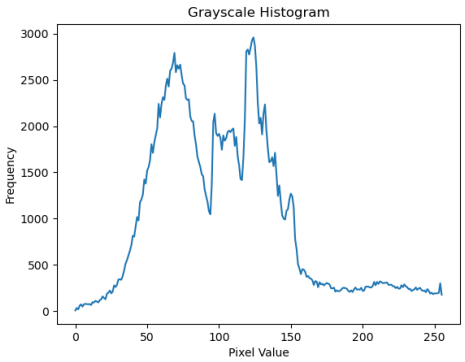
0.2 Pixel Value Distribution

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.

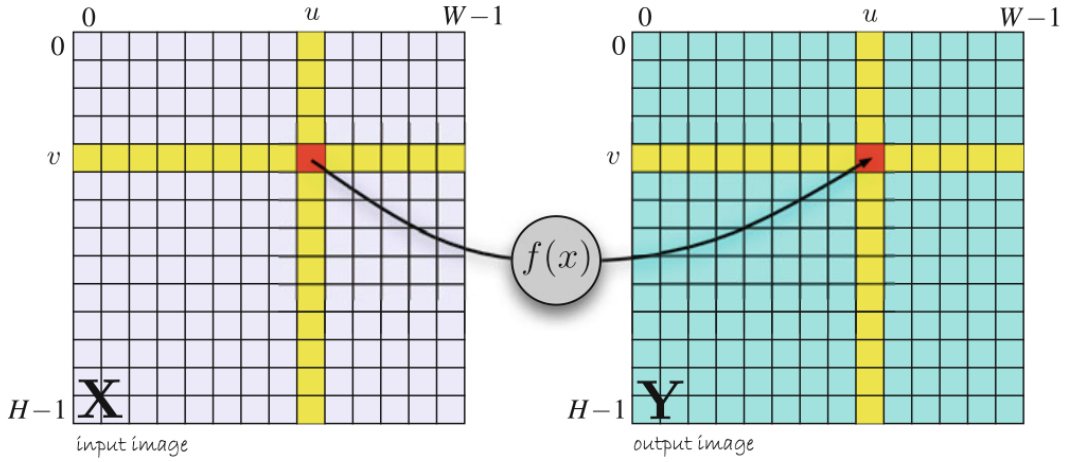
Looking at the histogram, one can observe several peaks in the histogram. The following code finds peaks in the histogram. There are a large number of peaks in a given histogram, which may not be useful. Hence, a set of filters can be applied to narrow the search and find peaks with the highest separation 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}")
```

0.3 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)
```

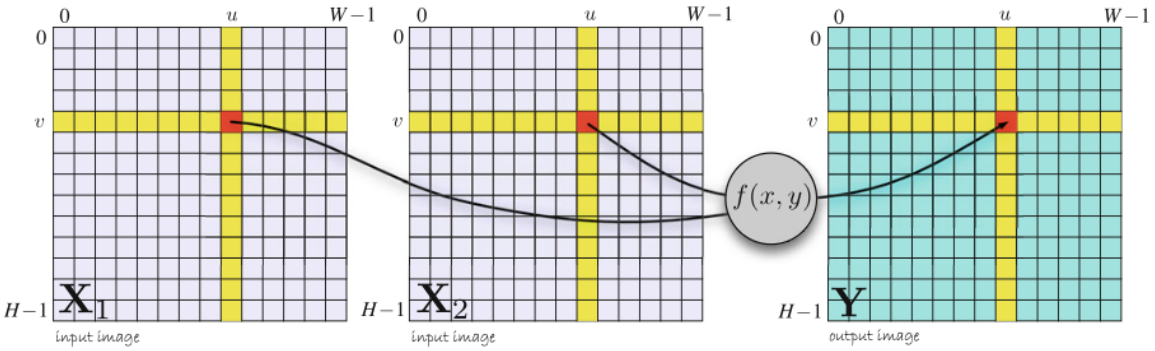
Examine the histogram of its new pixel-value distribution.

```
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()
```

0.4 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.

0.4.1 Chroma Keying

Chroma keying is a technique in which a specific color is removed from one image/video and imposed it on top of another image/video. This is commonly used in TV news broadcasting to have the reporter stand in front of a green screen and have the background from a different video.

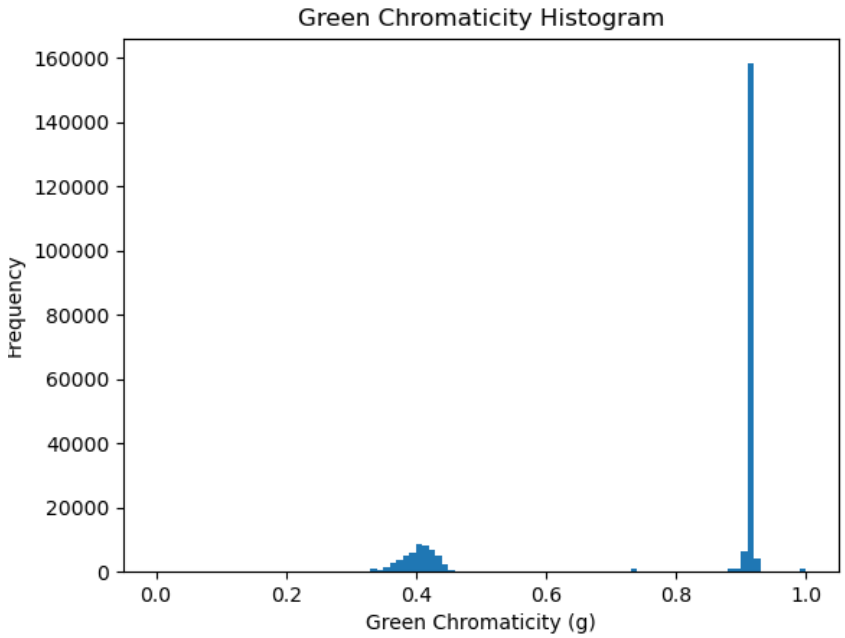
Assume that we have an image of a dog in front of a green screen. First step would be to load the image into the Python workspace. Next, task is identify the pixels of that image that are green in color. To do that, we calculate the chromacity of the green pixels in the image. For each pixel, the chromacity of the green channel g can be calculated as

$$g = \frac{G}{B + G + R}.$$

```
meme_dog = cv2.imread('/meme_dog.jpg')

# Compute green chromaticity
sum_rgb = np.sum(meme_dog, axis=2, keepdims=True)  # Calculate B+G+R pixel
    values
sum_rgb = np.where(sum_rgb == 0, 1, sum_rgb)  # Avoid division by zero
g_chroma = (meme_dog[:, :, 1] / sum_rgb[:, :, 0]) # g = G/(R+G+B)
```

Let us plot the green chromacity in a histogram to identify any particular thresholds to separate green pixels from the rest of the pixels in the green channel. Note the g values are in range between 0 and 1.

```
# Plot the chromacity histogram
plt.hist(g_chroma.flatten(), bins=100, range=(0, 1))
plt.title('Green Chromaticity Histogram')
plt.xlabel('Green Chromaticity (g)')
plt.ylabel('Frequency')
plt.savefig('meme_dog_green_chroma_hist.png')
plt.close()
```



Looking at the histogram, 0.6 would be a good threshold to separate the green screen background from the dig image. Next, apply the threshold and create mask.

```
# Based on the histogram, select a cut to create the mask
mask = (g_chroma < 0.6)
```

Finally, replace pixels of the original image, from the pixels of the dog image where the mask value equals TRUE.

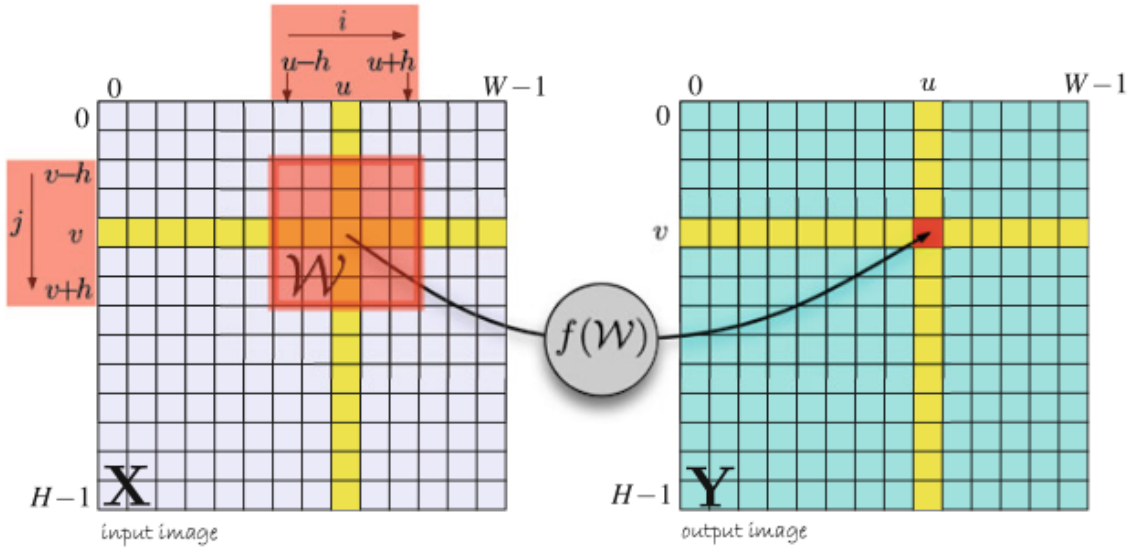
```
copy_of_img = img.copy() # Otherwise, we'd be changing the img itself.

# Apply the same mask on both images pixel by pixel
copy_of_img[mask == True] = meme_dog[mask == True]

# Save the results as an image
cv2.imwrite('meme_dog_in_denali.jpg', copy_of_img)
```

0.5 Spatial Operations

Each pixel in the output image is a function of all pixels in a region surrounding the corresponding pixel in the input image. It can be mathematically represented as the following

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

where $W_{u,v}$ is a region that surrounds a given pixel. This is known as a window.



There are two main categorizations of these spatial filters if they are linear. Assume we have a $n * n$ matrix of values where n is odd, and the center of the matrix is identified by a coordinate (0,0). This is known as a Kernel.

Correlation: The kernel K is applied to every pixel of the image, while pixel-wise multiplication and summing replace every pixel according to the following equation.

$$Y_{u,v} = \sum_{(i,j)\in W} x_{u+i,v+j} k_{i,j}, \forall (u,v) \in X.$$

This can be written as

$$Y = K \otimes X.$$

Convolution:

The kernel is rotated and flipped before being applied to every pixel of the image. This can be expressed as

$$Y_{u,v} = \sum_{(i,j)\in W} x_{u-i,v-j} k_{i,j}, \forall (u,v) \in X.$$

Convolution is often expressed in the operator form as

$$Y = K * X.$$

Convolution and correlation are computationally expensive – an $N \times N$ input image with a $w \times w$ kernel requires $w^2 N^2$ multiplications and additions.

0.5.1 Edge Detection

Detecting edges in images is a useful operation that applies a particular kernel over the image.

An edge in a grayscale image is a boundary where pixel intensity changes abruptly, indicating transitions between objects or regions. These edges can be detected by calculating the gradient of the pixel values.

Mathematically, this gradient around a pixel u on the horizontal axis of an image can be written as

$$\left.\frac{dp}{du}\right|_u = \frac{1}{2}(-p_{u-1} + p_{u+1}).$$

This is equivalent to a 1-dimensional correlation kernel K_{corr} where

$$K_{corr} = (-\frac{1}{2}, 0, \frac{1}{2})$$

The following code applies the above kernel to a grayscale image to detect horizontal edges.

```
gray_dog = cv2.cvtColor(meme_dog, cv2.COLOR_BGR2GRAY)

kernel = np.array([[-0.5, 0, 0.5]])
dog_edges = cv2.filter2D(gray_dog, -1, kernel)
cv2.imwrite('meme_dog_horizontal_edges.jpg', dog_edges)
```

A popular kernel for detecting horizontal edges is the Sobel kernel. The following code defines and applies the Sobel kernel.

```
# # Sobel kernels
sobel_x = np.array([[0.125, 0, -0.125], [-0.25, 0, 0.25], [-0.125, 0,
    0.125]])
sobel_y = np.array([[-0.125, -0.25, -0.125], [0, 0, 0], [0.125, 0.25,
    0.125]])

# Apply Sobel kernels
edges_x = cv2.filter2D(gray_dog, -1, sobel_x)
edges_y = cv2.filter2D(gray_dog, -1, sobel_y)

# Combine edges (\sqrt(x^2 +))
edges = np.sqrt(np.square(edges_x) + np.square(edges_y)).astype(np.uint8)

# Normalize to 0-255 range for better visibility
edges = cv2.normalize(edges, None, 0, 255, cv2.NORM_MINMAX)

cv2.imwrite('meme_dog_sobel.jpg', edges)
```

However, single derivative edge detectors like Sobel (or Laplacian) would often contain noise on the images where there would be a lot of insignificant edges in the resulting image. Therefore, it is common to combine a smoothing filter such as a Gaussian to suppress the noise. The following code combines a Gaussian with a Sobel filter.

```
## Gaussian with Sobel
# Define Sobel kernels
sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

# Define 5x5 Gaussian kernel
gaussian = cv2.getGaussianKernel(5, sigma=1.0)
gaussian = gaussian @ gaussian.T  # Create 5x5 kernel

# Convolve Sobel and Gaussian kernels
kernel_x = convolve2d(sobel_x, gaussian, mode='same')
kernel_y = convolve2d(sobel_y, gaussian, mode='same')

# Apply convolved kernels
edges_x = cv2.filter2D(gray_dog, cv2.CV_32F, kernel_x)
edges_y = cv2.filter2D(gray_dog, cv2.CV_32F, kernel_y)

# Compute edge magnitude
edges = np.sqrt(np.square(edges_x) + np.square(edges_y))

# Normalize to 0-255
```

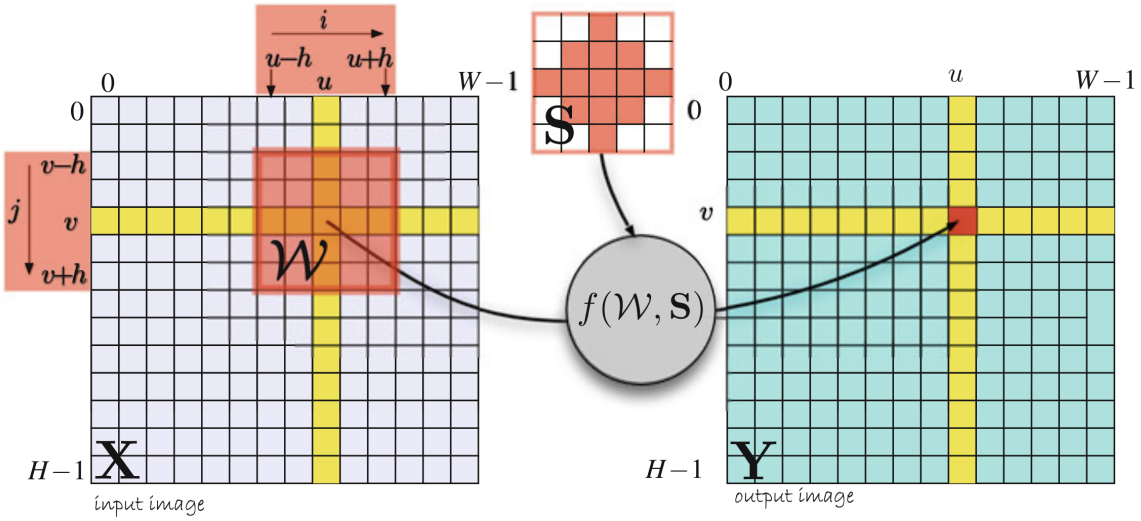
```
edges = cv2.normalize(edges, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)

# Save result
cv2.imwrite('sobel_gaussian_edges.jpg', edges)
```

0.6 Mathematical Morphology

In this class of non-linear operators, each pixel in the output image is a function of a subset of pixels in a region surrounding the corresponding pixel in the input image. The following is its mathematical and graphical representations.

$$Y_{u,v} = f(W_{u,v}, S), \forall (u,v) \in X$$

where W is the window and S is the structuring element.



We design these structuring elements so that it gets to pick which pixels the structure gets applied to in the window.

An application of the morphological operator is the detection of boundaries in a grayscale image. The following example illustrates the application of a circular type operator to an image where the image pixels are thresholded to black or white to detect the boundry region of black and white.

```
## Boundary detection
# Binarize image (threshold to create a binary mask)
_, binary = cv2.threshold(gray_dog, 127, 255, cv2.THRESH_BINARY)
cv2.imwrite('thresholded_img.jpg', binary)

# Define circular structuring element (approximated as a disk)
radius = 5
se = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (2*radius+1, 2*radius+1))

# Erode the binary image
eroded = cv2.erode(binary, se)

# Compute boundary (original - eroded)
boundary = binary - eroded

# Save result
cv2.imwrite('boundaries_morphological.jpg', boundary)
```