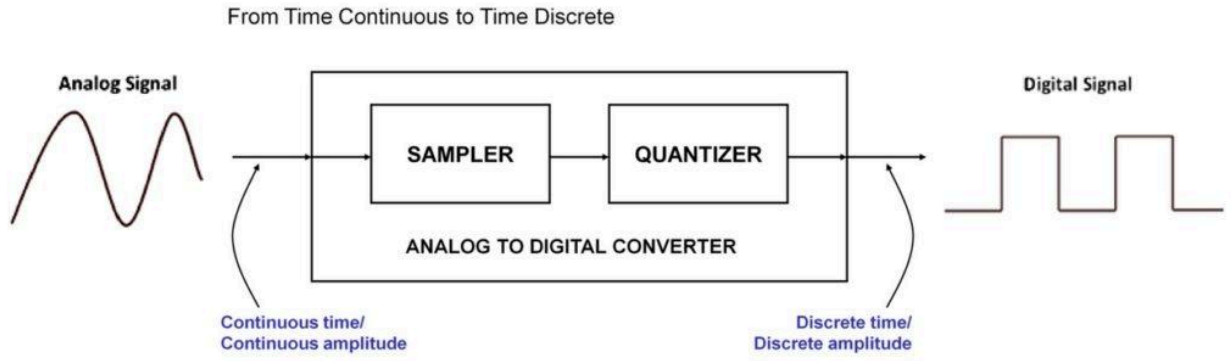
CHAPTER 2: DIGITAL IMAGE PROCESSING WITH OPENCV

2.1 Image Digitization

Mostly the output of image sensors is in the form of analog signal. Now the problem is that we cannot apply digital image processing and its techniques on analog signals.

This is due to the fact that we cannot store the output of image sensors which are in the form of analog signals because it requires infinite memory to store a signal that can have infinite values. So we have to convert this analog signal into digital signal.

To create a digital image, we need to convert the continuous data into digital form. This conversion from analog to digital involves two processes: sampling and quantization.

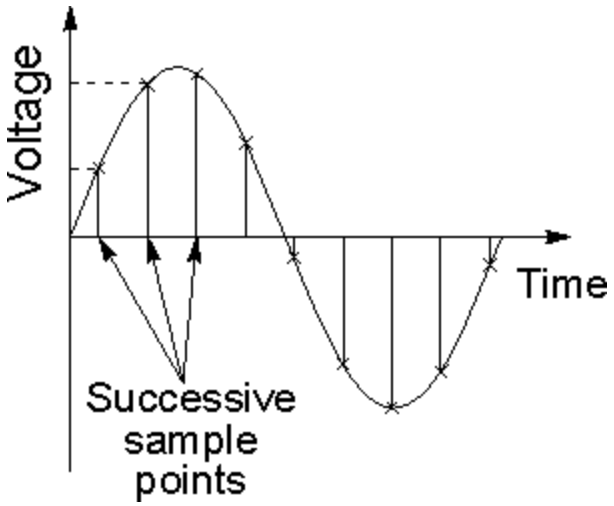


Sampling -> digitization of coordinate values

Quantization -> digitization of amplitude values

Sampling in Digital Image Processing:

- In this we digitize x-axis in sampling.
- It is done on the independent variable.
    - For e.g. if y = sinx, it is done on x variable.



- There are some variations in the sampled signal which are random in nature. These variations are due to noise.
- We can reduce this noise by more taking samples. More samples refer to collecting more data i.e. more pixels (in case of an image) which will eventually result in better image quality with less noise present.
- As we know that pixel is the smallest element in an image and for an image represented in the form of a matrix, total no. of pixels is given by:

Total number of pixels = Total number of rows X Total number of columns

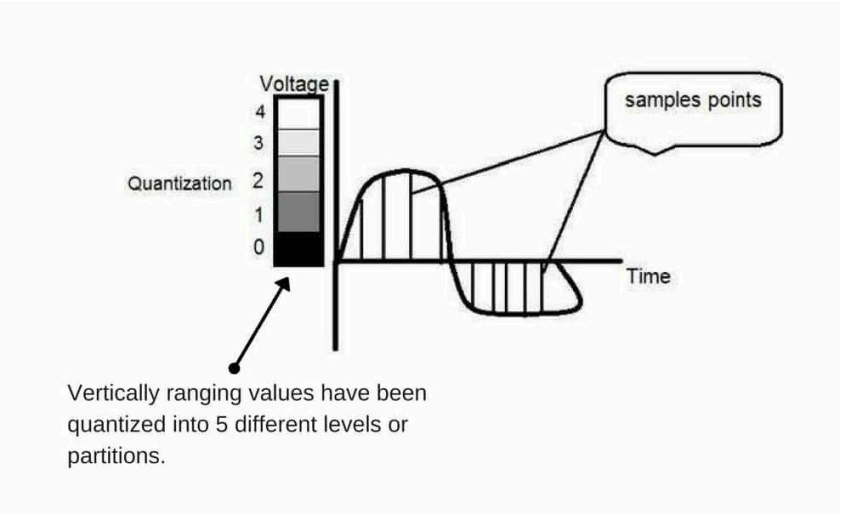
- The number of samples taken on the x-axis of a continuous signal refers to the number of pixels of that image.
- For a CCD array, if the number of sensors on a CCD array is equal to the number of pixels and number of pixels is equal to the number of samples taken, therefore we can say that number of samples taken is equal to the number of sensors on a CCD array.

No. of sensors on a CCD array = No. of pixels = No. of samples taken

- Oversampling is used for zooming. The difference between sampling and zooming is that sampling is done on signals while zooming is done on the digital image.

Quantization in Digital Image Processing:

- It is opposite of sampling as sampling is done on the x-axis, while quantization is done on the y-axis.
- Digitizing the amplitudes is quantization. In this, we divide the signal amplitude into quanta (partitions).



Relation of Quantization and gray level resolution:

Number of quantas (partitions) = Number of gray levels

- Number of gray levels here means number of different shades of gray.
- To improve image quality, we number of gray levels or gray level resolution up.
- If we increase this level to 256, it is known as the grayscale image.

$$L=2^K$$

Where,
L = gray level resolution
k = gray level
Gray level = number of bits per pixel (BPP) = number of levels per pixel

EXAMPLE:

You are given an analog grayscale image of size 10 cm x 10 cm.The resolution of the image is 100 dpi (dots per inch), meaning there are 100 pixels for every inch.You are required to:

1. Calculate the total number of pixels in the digital image after digitization.
2. Calculate the quantization levels if each pixel is stored using 8 bits (grayscale).

1. **Image Sampling (Total number of pixels)**

   First, convert the physical size of the image from centimeters to inches, since the resolution is given in dpi.

   $$1\,\text{inch} = 2.54\,\text{cm}$$

   Therefore, the size of the image in inches is:

   $$\text{Size in inches} = \frac{10\,\text{cm}}{2.54\,\text{cm/inch}} = 3.937\,\text{inches}$$

   Now, calculate the number of pixels per side of the image:

   $$\text{Pixels per side} = \text{Resolution (dpi)} \times \text{Size in inches} = 100\,\text{dpi} \times 3.937\,\text{inches} = 393.7\,\text{pixels}$$

   Since the image is square, the total number of pixels in the image is:

   $$\text{Total pixels} = 393.7 \times 393.7 \approx 155,000\,\text{pixels}$$

2. **Quantization Levels**

   In grayscale images, each pixel is represented by 8 bits (1 byte), which means it can take **2^8 = 256** different values, representing different shades of gray. Therefore, the quantization levels for each pixel are:

   $$\text{Quantization Levels} = 256$$

Program::

```
import cv2
import numpy as np

# Load an image from file
image = cv2.imread('your_image.jpg', cv2.IMREAD_GRAYSCALE)

# Resize the image to the target resolution (100 dpi in this case)
# Let's assume the image size in inches is 3.937 inches (10 cm)
dpi = 100
size_in_inches = 3.937
target_size = int(dpi * size_in_inches)

# Resize the image to the target resolution
resized_image = cv2.resize(image, (target_size, target_size))

# Reduce the bit-depth for quantization (simulate 8-bit image)
quantized_image = np.uint8(resized_image // 256)

# Display the original and processed images
```

```
cv2.imshow('Original Image', image)
cv2.imshow('Digitized Image', resized_image)
cv2.imshow('Quantized Image', quantized_image)

# Wait for a key press and close all windows
cv2.waitKey(0)
cv2.destroyAllWindows()
```

2.2.1 Image Manipulation

https://opencv-tutorial.readthedocs.io/en/latest/trans/transform.html
https://opencv-tutorial.readthedocs.io/en/latest/trans/transform.html#color-spaces
Image manipulation involves altering or enhancing images using various techniques to improve quality, adjust colors, remove artifacts, or apply artistic effects. This process is widely used in fields such as digital photography, computer vision, medical imaging, and multimedia.

Key Techniques in Image Manipulation

1. Image Scaling (Resizing)
    - Upsampling: Increases image size by adding pixels. Techniques include nearest neighbor interpolation, bilinear interpolation, and bicubic interpolation.
    - Downsampling: Reduces image size, typically by averaging pixels to retain important features while reducing resolution.
2. Cropping and Clipping
    - Cropping: Selects a specific region of interest in an image to focus on details or fit a required aspect ratio.
    - Clipping: Constrains pixel values within a specified range, preventing overexposure or underexposure.
3. Rotation and Affine Transformations
    - Rotation: Rotates the image by a specified angle. Useful for aligning images or applying perspective changes.
    - Affine Transformations: Involves scaling, rotation, translation, and shearing. Useful in image registration, alignment, and perspective correction.
4. Color Manipulation
    - Brightness and Contrast Adjustments: Modifies brightness and contrast to enhance visibility and adjust exposure.
    - Color Balancing: Adjusts color components to remove color casts and correct hues.
    - Color Space Conversion: Transforms images between color spaces (e.g., RGB to grayscale, HSV) for better analysis or specific effects.
5. Image Filtering

    - Smoothing (Blurring): Reduces noise using filters like Gaussian or median blur, making images appear softer.
    - Sharpening: Enhances edges and details using high-pass filters like Laplacian or unsharp masking.
6. Noise Removal
    - Median Filter: Reduces salt-and-pepper noise by replacing each pixel with the median value within a neighborhood.
    - Gaussian Filter: Smooths the image, reducing Gaussian noise but blurring fine details.
    - Bilateral Filter: Reduces noise while preserving edges, effective for edge-aware noise reduction.
7. Morphological Operations
    - Useful for binary images or thresholded images for tasks like removing small artifacts, edge enhancement, or feature extraction.
    - Includes dilation, erosion, opening, and closing.
8. Image Warping and Distortion Correction
    - Corrects distortions (e.g., lens distortion in photos) by remapping pixels using a transformation matrix.
    - Applications include stitching images for panoramas or creating virtual views.

Numerical Example:
You are given an image of dimensions 500x300 pixels. Calculate the new dimensions if:

1. Resizing the image by a factor of 0.5:
    - Original dimensions: 500x300 pixels
    - Resized dimensions: 0.5 * 500 = 250 pixels (width), 0.5 * 300 = 150 pixels (height)
    - New dimensions = 250x150 pixels
2. Cropping the region from (50, 50) to (200, 200):
    - Cropped region dimensions: 200 - 50 = 150 pixels (width), 200 - 50 = 150 pixels (height)
    - Cropped dimensions = 150x150 pixels

```
import cv2
# Load the image
image = cv2.imread('input_image.jpg')
# Resize the image by a factor of 0.5
resized_image = cv2.resize(image, (0, 0), fx=0.5, fy=0.5)
# Crop the image from (50, 50) to (200, 200)
cropped_image = image[50:200, 50:200]
# Save the results
cv2.imwrite('resized_image.jpg', resized_image)
cv2.imwrite('cropped_image.jpg', cropped_image)
```

2.2.2 Color Space Conversions

Different Color spaces:

Color Space	Model	Components	Range	Use Case	
RGB	Additive	Red, Green, Blue	[0, 255] for each channel	Digital displays, cameras	Additive model where colors are created by combining light in red, green, and blue.
HSV	Cylindrical	Hue, Saturation, Value	H: [0, 360], S: [0, 1], V: [0, 1]	Image editing, object detection	Intuitive representation of color for human perception, separating chromaticity and brightness.
HSL	Cylindrical	Hue, Saturation, Lightness	H: [0, 360], S: [0, 1], L: [0, 1]	Web graphics, design	Lightness component helps in easy manipulation of color brightness for design purposes.
CMYK	Subtractive	Cyan, Magenta, Yellow, Black	[0, 1] for each component	Printing, color separation	Subtractive model, used in printing, where colors are subtracted from white to create the desired color.
YCbCr	Component	Y, Cb, Cr	Y: [16, 235], Cb, Cr: [16, 240]	Video, broadcasting	Y component for luminance (brightness) and Cb, Cr for chrominance (color information) in video compression.
Lab (CIELAB)	Perceptual	L*, a*, b*	L*: [0, 100], a*: [-128, 128], b*: [-128, 128]	Color correction, matching, image processing	Perceptually uniform space designed to match human vision and be device-independent for accurate color representation.
XYZ	Tristimulus	X, Y, Z	No fixed range	Color management	Foundation for many color spaces, based on human vision sensitivity to primary colors.
YUV	Component	Y, U, V	Y: [16, 235], U/V: [16, 240]	Video compression	Separation of luminance (Y) from chrominance (U, V), optimizing video encoding and broadcasting.

Color Space Conversion involves changing an image's color model to make specific adjustments easier. The **HSV (Hue, Saturation, Value)** color space is commonly used in image processing because it separates color (hue) from intensity (value), making it easier to adjust colors and brightness independently.

In the HSV color model:

- **Hue (H)** represents the color type (e.g., red, green, blue) and ranges from 0 to 179 in OpenCV (instead of 0-360 in typical HSV color spaces).
- **Saturation (S)** controls the intensity or purity of the color, with higher values representing more vivid colors.
- **Value (V)**, often referred to as brightness, represents the intensity of light.

By converting an image from **BGR to HSV**, we can manipulate these properties independently. For example, we could adjust the **Hue** to change colors, **Saturation** to make colors more vivid or faded, and **Value** to brighten or darken the image.

```
import cv2
import numpy as np

# Load the image
image = cv2.imread('input.jpg')

# Convert the image from BGR to HSV color space
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

# Split the HSV channels
h, s, v = cv2.split(hsv_image)

# Modify Hue, Saturation, and Value independently
h = cv2.add(h, 10)  # Increase hue by 10 (wraps around, so it changes color slightly)
s = cv2.add(s, 50)  # Increase saturation to make colors more vivid
v = cv2.add(v, 30)  # Increase brightness

# Merge channels back into an HSV image
modified_hsv = cv2.merge([h, s, v])

# Convert back to BGR color space
result = cv2.cvtColor(modified_hsv, cv2.COLOR_HSV2BGR)

# Display the original and modified images
cv2.imshow("Original Image", image)
cv2.imshow("Modified Image", result)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

When to Use HSV Manipulation

- **Hue Adjustments**: To shift colors in an image without changing the intensity.
- **Saturation Adjustments**: To make colors more or less vivid.
- **Brightness Adjustments**: Modify the Value channel to brighten or darken the image without affecting color hue or saturation directly.

Using HSV gives you more control over specific color properties, making it ideal for tasks like color-based segmentation, filtering, and enhancements in image processing and computer vision.

Color Channel Manipulation

Color channel manipulation involves adjusting individual color channels (e.g., Red, Green, and Blue in the BGR color space) to alter specific aspects of an image's appearance. By changing these channels independently, you can create color effects, emphasize certain colors, or correct color imbalances.

In OpenCV, images are typically represented in the BGR (Blue, Green, Red) format, where each color is stored in a separate channel. Manipulating these channels directly can achieve various effects.

1. **Split Channels**:
2. **Adjust Channels**:
    - **Red Channel**: Increasing the red channel's intensity (adding 50) makes the image appear warmer.
    - **Blue Channel**: Reducing the blue channel (subtracting 30) makes the image appear less cool.
    - **Green Channel**: A slight increase in green (adding 10) balances the colors, giving a more natural look.
3. **Merge Channels**:
4. **Display Results**: We show both the original and modified images for comparison.

```
import cv2
import numpy as np

# Load the image
image = cv2.imread('input.jpg')

# Split the BGR channels
b, g, r = cv2.split(image)

# Modify each channel separately
# Increase intensity of Red channel to make the image warmer
r = cv2.add(r, 50)

# Decrease intensity of Blue channel to reduce coolness
b = cv2.subtract(b, 30)

# Optional: Increase intensity of Green channel slightly for a more balanced effect
g = cv2.add(g, 10)

# Merge the modified channels back
modified_image = cv2.merge([b, g, r])

# Display the original and modified images
```

```
cv2.imshow("Original Image", image)
cv2.imshow("Modified Image", modified_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Use Cases for Color Channel Manipulation

- **Color Correction**: Adjust color balance to correct lighting issues (e.g., reduce excess blue or red).
- **Creative Effects**: Intensify a specific color to create an artistic effect, such as a warmer or cooler look.
- **Highlighting Certain Features**: Emphasize specific elements in an image by making their color more intense.

Channel Adjustments and Their Effects

- **Increase Red Channel**: Adds warmth, useful for simulating sunset lighting.
- **Decrease Blue Channel**: Reduces a "cold" look, often useful in outdoor scenes with too much blue tint.
- **Increase Green Channel**: Can help balance the color in nature images, making vegetation more vibrant.

Color channel manipulation is a straightforward yet powerful technique for refining an image's appearance and creating specific visual effects.

```
import cv2

# Load the image
image = cv2.imread('input_image.jpg')

# Convert the image from RGB to HSV color space
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

# Save the converted image
cv2.imwrite('hsv_image.jpg', hsv_image)
```

Steps for RGB to HSV Conversion:

1. **Normalize the RGB values**: Convert R, G, and B to the range [0, 1] by dividing each by 255:

$$r = \frac{R}{255}, \quad g = \frac{G}{255}, \quad b = \frac{B}{255}$$

2. **Find the maximum and minimum values**:

$$C_{max} = \max(r, g, b), \quad C_{min} = \min(r, g, b)$$

$$\Delta = C_{max} - C_{min}$$

3. **Calculate Hue (H)**:

$$H = \begin{cases} 0^\circ, & \text{if } \Delta = 0 \\ 60^\circ \times \left(\frac{g-b}{\Delta}\right) & \text{if } C_{max} = r \\ 60^\circ \times \left(\frac{b-r}{\Delta} + 2\right) & \text{if } C_{max} = g \\ 60^\circ \times \left(\frac{r-g}{\Delta} + 4\right) & \text{if } C_{max} = b \end{cases}$$

   If H is negative, add 360° to it.

4. **Calculate Saturation (S)**:

$$S = \frac{\Delta}{C_{max}} \quad \text{if } C_{max} \neq 0$$

   If $C_{max} = 0$, then $S = 0$.

5. **Calculate Value (V)**:

$$V = C_{max}$$

Example (RGB = 255, 0, 0):

Let's apply this to the RGB color (255, 0, 0):

1. **Normalize**:

$$r = 1, \quad g = 0, \quad b = 0$$

2. **Find max, min, and delta**:

$$C_{max} = 1, \quad C_{min} = 0, \quad \Delta = 1$$

3. **Hue**: Since $C_{max} = r$, we use the formula:

$$H = 60^\circ \times \left(\frac{0 - 0}{1} \right) = 0^\circ$$

4. **Saturation**:

$$S = \frac{1}{1} = 1$$

5. **Value**:

$$V = 1$$

So, the HSV values for RGB(255, 0, 0) are:

- **Hue (H)** = 0°
- **Saturation (S)** = 1
- **Value (V)** = 1

These formulas are straightforward and easy to follow for converting RGB to HSV!

2.3 Image Transformations

1.GEOMETRIC Transformation
2. Intensity transformation(Point Operations)
3.Fourier Transformation

1. Geometric transformation

https://opencv-tutorial.readthedocs.io/en/latest/trans/transform.html

Transformation Type	Components	Parameters	Use Case	Unique Characteristic	Length preserved	Angle preserved
Affine Transformation	Translation, Scaling, Rotation, Shearing	Linear transformation matrix (2x2 or 3x3)	Image manipulation, object scaling, geometric transformations	Preserves parallelism and ratios of distances, but not necessarily angles or lengths.	No	No

Euclidean Transformation	Translation, Rotation	Only rotation and translation	Object alignment, rigid image registration	Preserves distances and angles, resulting in rigid body motion without distortion.	Yes	Yes
Rigid Transformation	Translation, Rotation	Rotation matrix and translation vector	Object tracking, motion analysis, robotics	Preserves both distances and angles, maintaining the object's shape and size.	Yes	Yes

2. Intensity Transformation Operations on Images

Intensity transformations are applied on images for contrast manipulation or image thresholding. These are in the spatial domain, i.e. they are performed directly on the pixels of the image at hand, as opposed to being performed on the Fourier transform of the image. The following are commonly used intensity transformations:

1. Image Negatives (Linear)
2. Log Transformations
3. Power-Law (Gamma) Transformations
4. Piecewise-Linear Transformation Functions

Image Negatives – Image negatives are discussed in this article. Mathematically, assume that an image goes from intensity levels 0 to (L-1). Generally, L = 256. Then, the negative transformation can be described by the expression s = L-1-r where r is the initial intensity level and s is the final intensity level of a pixel. This produces a photographic negative.

```
# Create a negative image
negative_image = 255 - image

cv2.imshow("Negative Image", negative_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Log Transformations –

Mathematically, log transformations can be expressed as s = clog(1+r). Here, s is the output intensity, r>=0 is the input intensity of the pixel, and c is a scaling constant. c is given by

255/(log (1 + m)), where m is the maximum pixel value in the image. It is done to ensure that the final pixel value does not exceed (L-1), or 255. Practically, log transformation maps a narrow range of low-intensity input values to a wide range of output values. Consider the following input image.Below is the code to apply log transformation to the image.

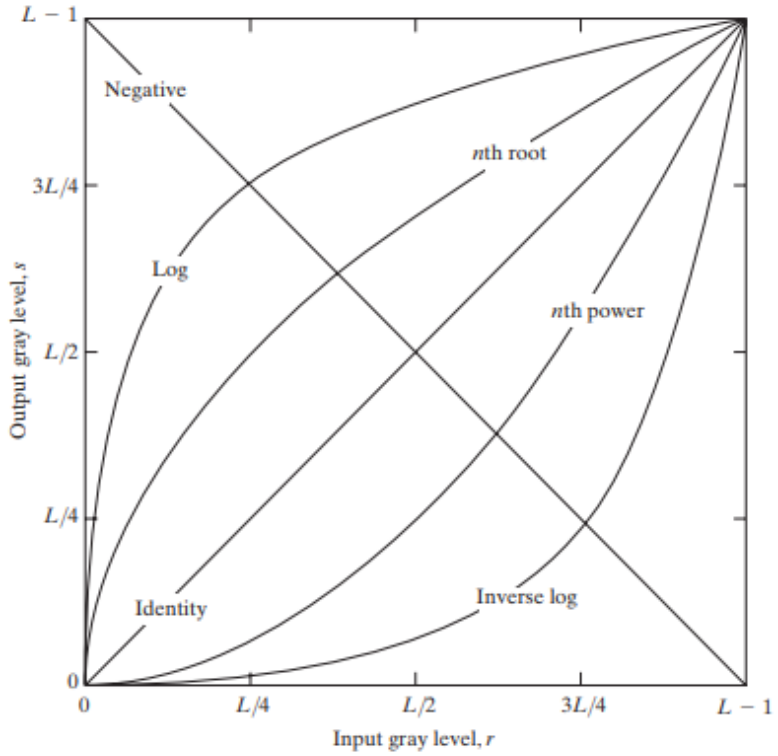
```
import cv2
import numpy as np

# Open the image.
img = cv2.imread('sample.jpg')

# Apply log transform.
c = 255/(np.log(1 + np.max(img)))
log_transformed = c * np.log(1 + img)

# Specify the data type.
log_transformed = np.array(log_transformed, dtype = np.uint8)

# Save the output.
cv2.imwrite('log_transformed.jpg', log_transformed)
```



Power-Law (Gamma) Transformation –

Power-law (gamma) transformations can be mathematically expressed as

$$s = 255 \times (255/input)^{\gamma}$$

Gamma correction is important for displaying images on a screen correctly, to prevent bleaching or darkening of images when viewed from different types of monitors with different display settings. This is done because our eyes perceive images in a gamma-shaped curve, whereas cameras capture images in a linear fashion. Below is the Python code to apply gamma correction.

```
import cv2
import numpy as np
img = cv2.imread('sample.jpg')
 # Trying 4 gamma values.
for gamma in [0.1, 0.5, 1.2, 2.2]:
    # Apply gamma correction.
    gamma_corrected = np.array(255*(img / 255) ** gamma, dtype = 'uint8')
    # Save edited images.
    cv2.imwrite('gamma_transformed'+str(gamma)+'.jpg', gamma_corrected)
```

Piecewise-Linear Transformation Functions –

These functions, as the name suggests, are not entirely linear in nature. However, they are linear between certain x-intervals. One of the most commonly used piecewise-linear transformation functions is contrast stretching. Contrast can be defined as:

Contrast = (I_max - I_min)/(I_max + I_min)

Stretched Image= (I-I_min)(Mx-Mn)/(I_max - I_min)

```
import cv2
import numpy as np

# Increase contrast
contrast = 1.5
# Change this value for different levels of contrast
contrast_image = cv2.convertScaleAbs(image, alpha=contrast)
cv2.imshow("Contrast Adjusted Image", contrast_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
—----------------------
import cv2
import numpy as np
```

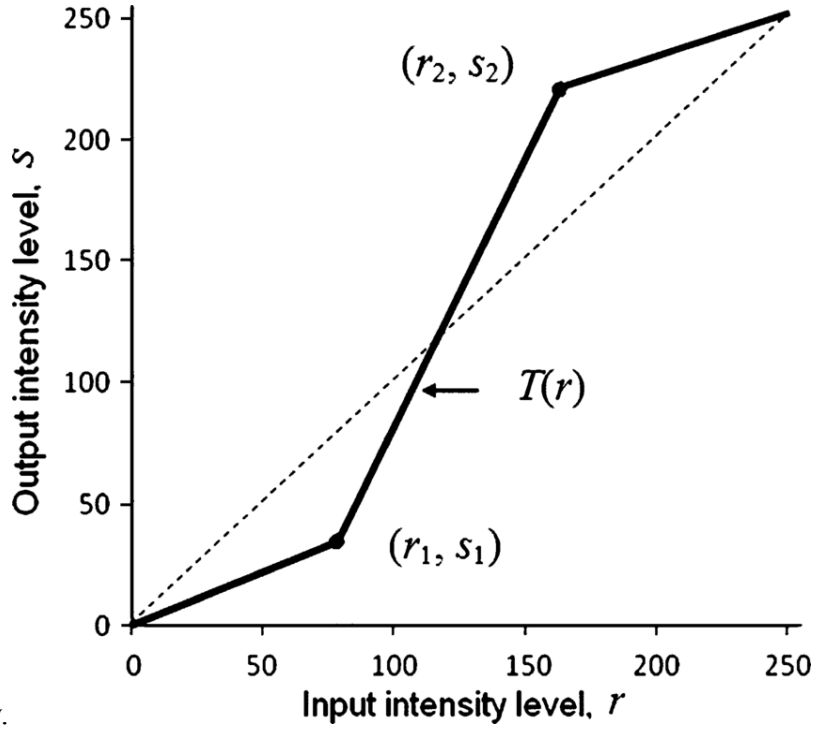
```
# Load the image in grayscale
image = cv2.imread('input.jpg', cv2.IMREAD_GRAYSCALE)

# Find the minimum and maximum pixel intensities in the image
min_val = np.min(image)
max_val = np.max(image)

# Apply contrast stretching
stretched_image = ((image - min_val) / (max_val - min_val)) * 255
stretched_image = stretched_image.astype(np.uint8)  # Convert to unsigned 8-bit integer type

cv2.imshow("Original Image", image)
cv2.imshow("Contrast Stretched Image", stretched_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

This process expands the range of intensity levels in an image so that it spans the full intensity of the camera/display.



With (r1, s1), (r2, s2) as parameters, the function stretches the intensity levels by essentially decreasing the intensity of the dark pixels and increasing the intensity of the light pixels. If r1 = s1 = 0 and r2 = s2 = L-1, the function becomes a straight dotted line in the graph (which gives no

effect). The function is monotonically increasing so that the order of intensity levels between pixels is preserved.

Numerical Example:

You are given an image with dimensions 400x300 pixels. Perform the following transformations:

1. Translate the image 50 pixels to the right and 30 pixels down.
2. Rotate the image by 45 degrees around the center.

Solution:

1. Translation:

Translation matrix:M = [[1, 0, 50], [0, 1, 30]]

2. Rotation:

M = cv2.getRotationMatrix2D(center, angle, scale)
For a 45° rotation around the center:
center = (200, 150)
angle = 45
scale = 1

```
import cv2
import numpy as np

# Load the image
image = cv2.imread('input_image.jpg')
rows, cols = image.shape[:2]

# Translation matrix (50 pixels right, 30 pixels down)
M_translation = np.float32([[1, 0, 50], [0, 1, 30]])
translated_image = cv2.warpAffine(image, M_translation, (cols, rows))

# Rotation matrix (45 degrees around the center)
M_rotation = cv2.getRotationMatrix2D((cols/2, rows/2), 45, 1)
rotated_image = cv2.warpAffine(image, M_rotation, (cols, rows))

# Save the results
cv2.imwrite('translated_image.jpg', translated_image)
cv2.imwrite('rotated_image.jpg', rotated_image)
```

Example Question: An image of size 800x600 pixels is resized to 400x300 pixels. Calculate the new resolution and how much the total number of pixels has been reduced.
Solution:

- Original resolution: 800 x 600 = 480,000 pixels
- New resolution: 400 x 300 = 120,000 pixels
- Reduction in the number of pixels = 480,000 - 120,000 = 360,000 pixels
- The percentage reduction in the total number of pixels = (360,000 / 480,000) * 100 = 75%

Example Question: An image is rotated by 90 degrees. If the original image is 1024x768 pixels, what are the dimensions of the image after rotation?

Solution:

- Original dimensions: 1024x768 pixels.
- After rotating by 90 degrees, the dimensions will be swapped:
- New dimensions=768×1024 pixels