EC7212 – COMPUTER VISION AND IMAGE PROCESSING

ASSIGNMENT 01

NAME :DISSANAYAKE D.K.R.C.K.

REG NO : EG/2020/3910

SEMESTER : 07

DATE : 21/06/2025

Table of Contents

1 Introduction

Digital image processing plays a fundamental role in modern computer vision and artificial intelligence systems. It involves the application of algorithms to improve, analyze, and transform digital images for various practical applications such as medical imaging, object detection, and machine learning.

This assignment explores four foundational operations in image processing using Python:

1. **Intensity Level Reduction**

   This task reduces the number of grayscale intensity levels in an image from 256 to a lower power-of-two range. It demonstrates quantization techniques and their impact on image quality.

2. **Spatial Averaging (Smoothing)**

   In this task, spatial filters such as 3×3, 10×10, and 20×20 neighborhood averaging are applied to images. These filters help in reducing noise and small variations by averaging surrounding pixel values.

3. **Image Rotation**

   Images are rotated by 45° and 90°, requiring appropriate handling of geometry, interpolation, and image boundaries. This task shows how image transformation can be handled programmatically while preserving visual content.

4. **Spatial Resolution Reduction via Block Averaging**

   This final task involves dividing an image into non-overlapping blocks (3×3, 5×5, 7×7) and replacing each block with its average value. It simulates lowering the spatial resolution of an image, useful for tasks like compression and scaling.

All tasks were implemented using Python libraries such as **NumPy, Pillow, OpenCV,** and **Matplotlib**, and run within **Jupyter Notebooks** for clear visual demonstration. The combination of synthetic and real image manipulation provided a hands-on understanding of core image processing principles.

2 Methodology

The implementation of this assignment was carried out in a structured and modular approach using Python. Each image processing task was developed and tested separately in its own Jupyter Notebook file for clarity and ease of experimentation. The project followed a consistent methodology across all tasks:

2.1 Development Environment

The project was organized in a clearly defined folder structure, separating source code (notebooks/), input/output images (data/, outputs/), and supporting files (README.md, requirements.txt).

All programs were written using Python 3, leveraging popular libraries such as:

- **NumPy** for matrix operations
- **Pillow (PIL)** and **OpenCV** for image handling
- **Matplotlib** for visualization
- **Jupyter Notebook** for step-by-step execution and output presentation

2.2 Folder Structure

```
AI-Image-Processing-Python/
│
├─── data/                          # Input images
│    └─── input_image.jpg
│
├─── notebooks/                     # Task notebooks
│    ├─── task1_intensity_reduction.ipynb
│    ├─── task2_spatial_average.ipynb
│    ├─── task3_rotation.ipynb
│    └─── task4_spatial_resolution_reduction.ipynb
│
├─── outputs/                       # Generated output images
│    ├─── task1/
│    ├─── task2/
│    ├─── task3/
│    └─── task4/
│
├─── requirements.txt              # Python dependencies
└─── README.md                     # This file
```

2.3 Image Input

A single input image was used across all tasks, loaded and pre-processed where necessary.

For grayscale-based operations (like intensity reduction and spatial filtering), the image was explicitly converted to grayscale to simplify processing.

2.4 Result Visualization and Saving

For each step, the intermediate and final outputs were visualized using matplotlib and saved in the outputs/ directory for comparison.

Titles and axis formatting were used to make visual analysis clearer.

3 Implementation

3.1 To reduce the number of intensity levels in an image from 256 to 2, in integer powers of 2. The desired number of intensity levels needs to be a variable input to your program.

- A image was loaded and processed by reducing its pixel intensity levels from 256 to smaller power-of-two levels (e.g., 2, 4, 8, 16...).
- A mathematical scaling technique was used to map pixel values to the desired levels.
- The program accepted the number of levels as a user input, allowing flexible testing and comparison.
- The output demonstrated the visual effect of quantization and intensity loss.

Code and Results

Task 1 : To reduce the number of intensity levels in an image from 256 to 2, in integer powers of 2. The desired number of intensity levels needs to be a variable input to your program.

Import Libraries

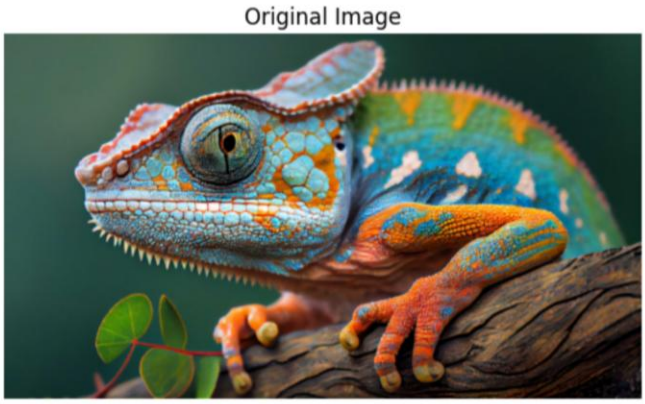
In [3]:
```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import os
```

Define the Reduction Function

In [4]:
```
def reduce_intensity_levels(img_array, levels):
    """
    Reduce intensity levels of an image array to 'levels'.
    'levels' must be a power of 2 and less than or equal to 256.
    """
    assert (levels & (levels - 1) == 0) and levels <= 256, "Levels must be a power of 2 and <= 256"

    factor = 256 // levels  # e.g., for levels=8, factor=32
    quantized_img = (img_array // factor) * factor
    return quantized_img.astype(np.uint8)
```

Load Image

In [5]:
```
image_path = '../data/input_image.jpg'  # Change this if your image name is different

# Load the image and convert to grayscale
img = Image.open(image_path).convert('RGB')
img_array = np.array(img)

# Display the original image
plt.imshow(img_array)
plt.title("Original Image")
plt.axis('off')
plt.show()
```



Set Desired Intensity Levels

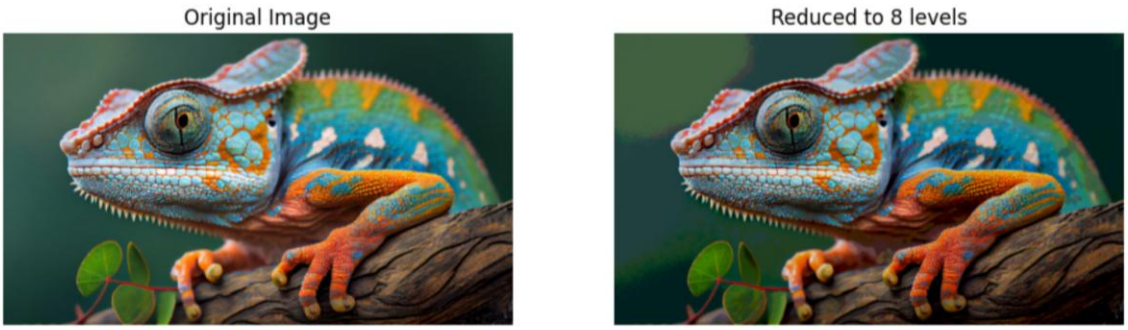
In [6]:
```
desired_levels = 8  # Try 2, 4, 8, 16, 32, 64, 128, 256
```

Apply Intensity Reduction

In [7]:
```
reduced_img_array = reduce_intensity_levels(img_array, desired_levels)

# Convert the result back to PIL Image
reduced_img = Image.fromarray(reduced_img_array)
```

Show Original vs Reduced Image

In [8]:
```
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(img_array)
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title(f"Reduced to {desired_levels} levels")
plt.imshow(reduced_img_array)
plt.axis('off')

plt.show()
```



Save Output Image

In [9]:
```
output_dir = '../outputs/task1'
os.makedirs(output_dir, exist_ok=True)

output_path = os.path.join(output_dir, f'reduced_{desired_levels}_levels.jpg')
reduced_img.save(output_path)

print(f"Saved reduced image at: {output_path}")
```

```
Saved reduced image at: ../outputs/task1\reduced_8_levels.jpg
```

3.2 Load an image and then perform a simple spatial 3x3 average of image pixels. Repeat the process for a 10x10 neighborhood and again for a 20x20 neighborhood.

- The same image was used as input.
- Three different averaging filters were implemented using local neighborhoods of sizes 3×3, 10×10, and 20×20.
- Each neighborhood was applied across the image using a convolution-like sliding window approach.
- The averaged pixel values replaced the original ones, smoothing the image and reducing local noise.
- Results were visualized to observe how increasing the kernel size affected image sharpness.

Code and Results

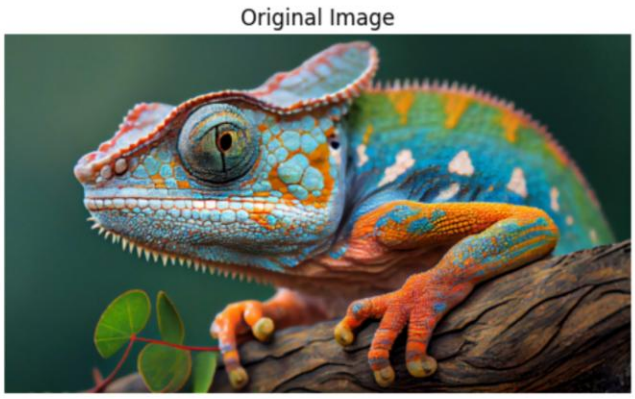
Task 2 : Load an image and then perform a simple spatial 3x3 average of image pixels. Repeat the process for a 10x10 neighborhood and again for a 20x20 neighborhood.

Import Libraries

In [11]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
import os
```

Load Image

In [12]:
```
image_path = '../data/input_image.jpg'

# Load image in RGB mode using Pillow
img = Image.open(image_path).convert('RGB')
img_array = np.array(img)

plt.imshow(img_array)
plt.title("Original Image")
plt.axis('off')
plt.show()
```

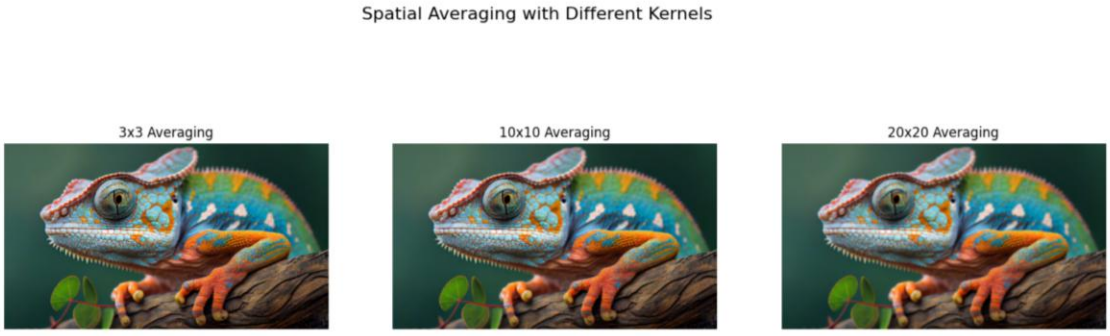


Define Average Filtering Function

In [13]:
```
def apply_average_filter(img_array, kernel_size):

    """Apply average (mean) filter of given kernel size to the image."""

    return cv2.blur(img_array, (kernel_size, kernel_size))
```

Apply and Display Results for Each Kernel Size

In [14]:
```
kernel_sizes = [3, 10, 20]
results = {}

plt.figure(figsize=(18, 6))

for i, k in enumerate(kernel_sizes):
    smoothed = apply_average_filter(img_array, k)
    results[k] = smoothed

    plt.subplot(1, 3, i+1)
    plt.imshow(smoothed)
    plt.title(f"{k}x{k} Averaging")
    plt.axis('off')

plt.suptitle("Spatial Averaging with Different Kernels", fontsize=16)
plt.show()
```



Save All Output Images

In [15]:
```
output_dir = '../outputs/task2'
os.makedirs(output_dir, exist_ok=True)

for k, img_out in results.items():
    output_path = os.path.join(output_dir, f'average_{k}x{k}.jpg')
    Image.fromarray(img_out).save(output_path)

print("All smoothed images saved to:", output_dir)
```

```
All smoothed images saved to: ../outputs/task2
```

3.3 Rotate an image by 45 and 90 degrees.

- The image was rotated by 45° and 90° using OpenCV's built-in warpAffine and rotate functions.
- The 45° rotation required calculating a transformation matrix and adjusting the image canvas to avoid clipping.
- The 90° rotation used a simple and accurate method for axis-aligned rotation.
- The results were displayed to illustrate geometric transformations without loss of image data.

Code and Results

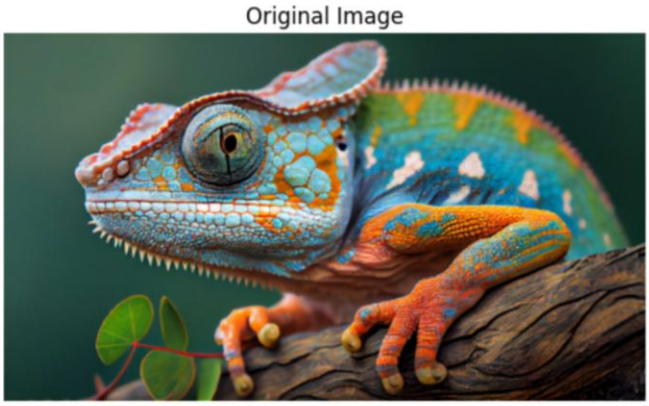
Task 3 : Rotate an image by 45 and 90 degrees.

Import Required Libraries

In [7]:
```
import cv2
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import os
```

Load the Image

In [8]:
```
image_path = '../data/input_image.jpg'  # Adjust if needed

# Load image in RGB
img = cv2.cvtColor(cv2.imread(image_path), cv2.COLOR_BGR2RGB)

plt.imshow(img)
plt.title("Original Image")
plt.axis('off')
plt.show()
```



Define Rotation Function

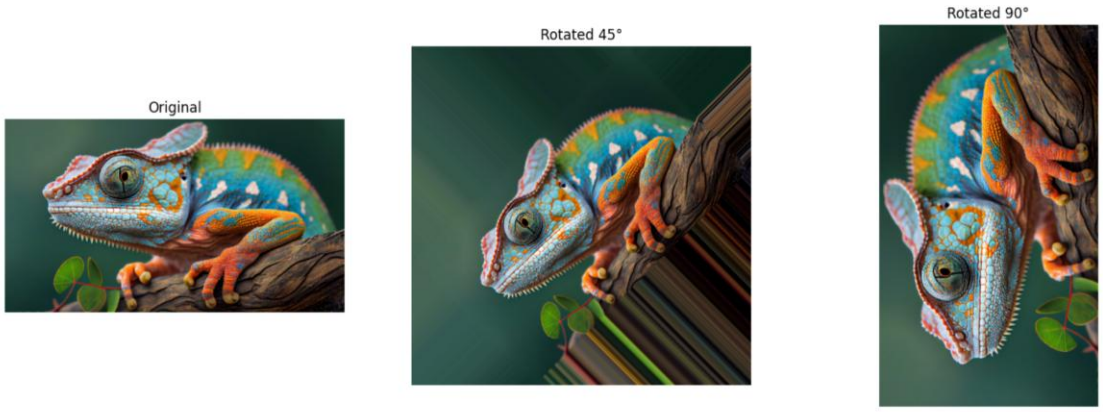
In [9]:
```python
def rotate_image(image, angle):

    """Rotate an image around its center by given angle. The canvas size is adjusted to fit the whole image."""

    (h, w) = image.shape[:2]
    center = (w // 2, h // 2)

    # Get the rotation matrix
    M = cv2.getRotationMatrix2D(center, angle, 1.0)

    # Calculate the new bounding dimensions
    cos = np.abs(M[0, 0])
    sin = np.abs(M[0, 1])
    new_w = int((h * sin) + (w * cos))
    new_h = int((h * cos) + (w * sin))

    # Adjust the rotation matrix to consider translation
    M[0, 2] += (new_w / 2) - center[0]
    M[1, 2] += (new_h / 2) - center[1]

    # Perform the rotation
    rotated = cv2.warpAffine(image, M, (new_w, new_h), borderMode=cv2.BORDER_REPLICATE)
    return rotated
```

Apply Rotations

In [10]:
```python
rotated_45 = rotate_image(img, 45)
rotated_90 = rotate_image(img, 90)
```

Display Results

In [11]:
```python
plt.figure(figsize=(18, 6))

plt.subplot(1, 3, 1)
plt.imshow(img)
plt.title("Original")
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(rotated_45)
plt.title("Rotated 45°")
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(rotated_90)
plt.title("Rotated 90°")
plt.axis('off')

plt.show()
```



Save Rotated Images

In [12]:
```python
output_dir = '../outputs/task3'
os.makedirs(output_dir, exist_ok=True)

Image.fromarray(rotated_45).save(os.path.join(output_dir, 'rotated_45.jpg'))
Image.fromarray(rotated_90).save(os.path.join(output_dir, 'rotated_90.jpg'))

print("Rotated images saved to:", output_dir)
```

```
Rotated images saved to: ../outputs/task3
```

3.4 For every 3×3 block of the image (without overlapping), replace all the corresponding 9 pixels by their average. This operation simulates reducing the image spatial resolution. Repeat this for 5×5 blocks and 7×7 blocks.

- The image was divided into non-overlapping blocks of size 3×3, 5×5, and 7×7.
- Within each block, the average RGB value was computed and reassigned to every pixel in that block.
- This simulated a reduction in spatial resolution while maintaining the overall structure and color of the image.
- The process was applied to color images, requiring separate averaging of each channel (Red, Green, Blue).

Code and Results

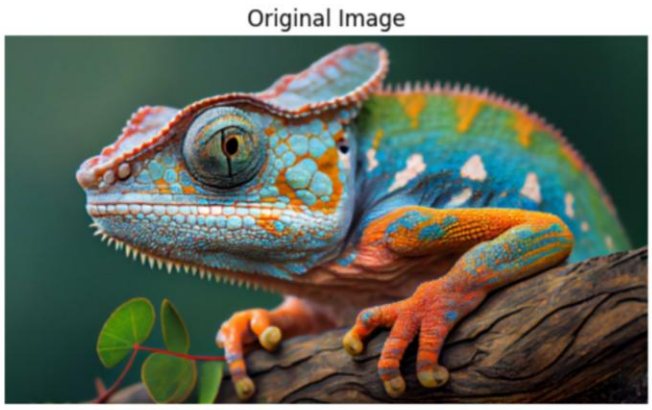
Task 4 : For every 3×3 block of the image (without overlapping), replace all the corresponding 9 pixels by their average. This operation simulates reducing the image spatial resolution. Repeat this for 5×5 blocks and 7×7 blocks.

Import Libraries

In [27]:
```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import os
```

Load the Image

In [28]:
```
image_path = '../data/input_image.jpg'

img = Image.open(image_path).convert('RGB')  # RGB
img_array = np.array(img)

plt.imshow(img_array)
plt.title("Original Image")
plt.axis('off')
plt.show()
```



Define Block Averaging Function

In [29]:
```python
def block_average_color(img_array, block_size):
    """
    Apply block averaging to a color image in non-overlapping blocks.
    Each block's R, G, B channels are averaged separately.
    """
    h, w, c = img_array.shape
    new_img = img_array.copy()

    for i in range(0, h - block_size + 1, block_size):
        for j in range(0, w - block_size + 1, block_size):
            block = img_array[i:i+block_size, j:j+block_size, :]
            avg_color = block.mean(axis=(0, 1)).astype(np.uint8)
            new_img[i:i+block_size, j:j+block_size, :] = avg_color

    return new_img
```

Apply the Operation for 3×3, 5×5, and 7×7 Blocks

In [30]:
```python
block_sizes = [3, 5, 7]
results = {}

plt.figure(figsize=(18, 6))

for idx, bsize in enumerate(block_sizes):
    reduced = block_average_color(img_array, bsize)
    results[bsize] = reduced

    plt.subplot(1, 3, idx + 1)
    plt.imshow(reduced)
    plt.title(f"{bsize}x{bsize} Block Averaging")
    plt.axis('off')

plt.suptitle("Spatial Resolution Reduction (Color)", fontsize=16)
plt.show()
```

Save Outputs

In [31]:
```python
output_dir = '../outputs/task4'
os.makedirs(output_dir, exist_ok=True)

for bsize, result in results.items():
    Image.fromarray(result).save(os.path.join(output_dir, f'block_avg_{bsize}x{bsize}.jpg'))

print("Block-averaged images saved to:", output_dir)
```

```
Block-averaged images saved to: ../outputs/task4
```

4 Github Repository

You can visit and check the Full project from this Link

https://github.com/chathuradissanayake/AI-Image-Processing-Python.git