Image Processing Lab: From Pixels to Perception

ITAI 1378 Module 04 - Hands-On Laboratory

Duration: 45 minutes
Platform: Google Colab (Free Version)
Prerequisites: Basic Python knowledge from previous classes

🎯 Lab Objectives

By the end of this lab, you will:

1. **Understand** how computers represent images as numerical matrices
2. **Implement** fundamental image processing operations from scratch
3. **Apply** traditional image processing techniques using OpenCV and Pillow
4. **Experience** the connection between theory and practice
5. **Prepare** for your Image Processing Adventure Quest assignment

---

📚 Connection to Today's Lecture

This lab directly implements the concepts we explored in class:

- **Digital Image Representation**: See how images become matrices of numbers
- **Core Operations**: Implement point, neighborhood, and geometric operations
- **Traditional Tools**: Hands-on experience with OpenCV and Pillow
- **Bridge to AI**: Understand the foundation that powers modern AI tools like Nano Banana

---

⏱️ Lab Timeline (45 minutes)

Time	Section	Activity
0-5 min	Setup	Environment setup and imports
5-15 min	Part 1	Digital image fundamentals
15-25 min	Part 2	Basic image operations
25-35 min	Part 3	Advanced processing techniques
35-40 min	Part 4	Creative exploration
40-45 min	Wrap-up	Reflection and next steps

🚀 Let's Begin!

Important: Run each cell in order. Read the explanations carefully - they connect directly to today's lecture concepts!

Note: I keep orignial code collapse so keep is oraganize the **Bold Markdown cells** are the Inline responses to questions, 1-5 Personal Experiments and Optional 1-5 Challenges. make sure the run the Hidden code before the next code.

🔧 Setup and Environment Preparation

First, let's set up our environment. We'll use the same libraries we discussed in class:

- **OpenCV**: The powerhouse for computer vision (remember from our tools discussion?)
- **Pillow (PIL)**: Python-friendly image processing
- **NumPy**: For matrix operations (images are matrices!)
- **Matplotlib**: For displaying our results

💡 **Recall from lecture:** These are the "traditional tools" that give us precise control over every pixel!

↳ 1 cell hidden

📸 Load Sample Images

Let's load some sample images to work with. We'll provide both options:

1. **Download sample images** from the internet

2. **Upload your own images** (optional)

🔗 **Connection to lecture:** Remember how we discussed image acquisition as the first step in any vision system?

↳ 1 cell hidden

📊 Part 1: Digital Image Fundamentals (10 minutes)

Understanding Images as Matrices

🎓 **Lecture Connection:** Remember when we discussed how computers see images as matrices of numbers? Let's see this in action!

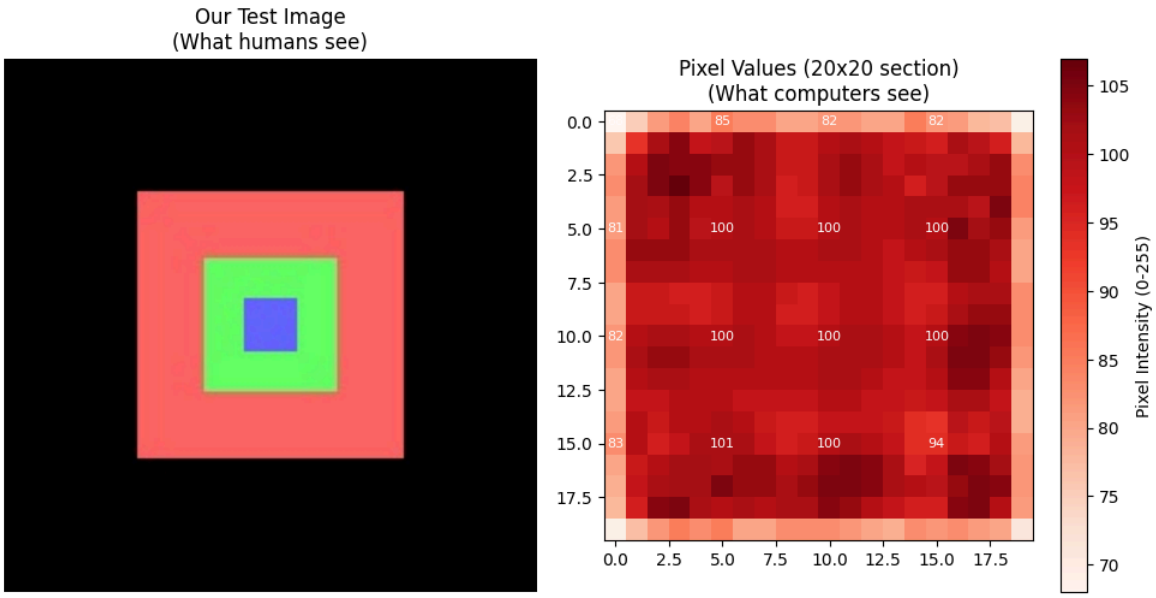
In our lecture, we learned:

- **Grayscale images**: 2D matrices (height × width)
- **Color images**: 3D matrices (height × width × channels)
- **Pixel values**: Numbers from 0-255 for 8-bit images

Let's explore this hands-on!

```
# Load our test image using OpenCV
# Note: OpenCV loads images in BGR format (Blue, Green, Red)
img_bgr = cv2.imread('test_image.jpg')

# Convert to RGB for proper display (remember color spaces from lecture?)
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

# Let's examine the image properties
print("🔍 IMAGE ANALYSIS - Just like we discussed in class!")
print(f"📐 Image shape: {img_rgb.shape}")
print(f"📊 Data type: {img_rgb.dtype}")
print(f"📈 Value range: {img_rgb.min()} to {img_rgb.max()}")
print(f"💾 Memory usage: {img_rgb.nbytes} bytes")

# Display the image
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(img_rgb)
plt.title('Our Test Image\n(What humans see)')
plt.axis('off')

plt.subplot(1, 2, 2)
# Show a small section of pixel values (what computers see)
pixel_section = img_rgb[90:110, 90:110, 0]  # 20x20 red channel section
plt.imshow(pixel_section, cmap='Reds')
plt.title('Pixel Values (20x20 section)\n(What computers see)')
plt.colorbar(label='Pixel Intensity (0-255)')

# Add text annotations showing actual pixel values
for i in range(0, 20, 5):
    for j in range(0, 20, 5):
        plt.text(j, i, str(pixel_section[i, j]),
                ha='center', va='center', color='white', fontsize=8)

plt.tight_layout()
plt.show()

print("\n💡 Key Insight: The image on the left is what we see.")
print("   The numbers on the right are what the computer sees!")
```

```
🔍 IMAGE ANALYSIS - Just like we discussed in class!
📐 Image shape: (200, 200, 3)
📊 Data type: uint8
📈 Value range: 0 to 255
💾 Memory usage: 120000 bytes
```




```
💡 Key Insight: The image on the left is what we see.
   The numbers on the right are what the computer sees!
```

My Analysis of Matrix Representation

Matrix representation means every image is essentially a 2D or 3D array of numbers, with pixel values corresponding directly to intensity or color. A color image is represented as a 3D tensor, where the first two dimensions are height and width, and the third dimension represents the color channels (e.g., Red, Green, Blue). The value at each position in the matrix represents the intensity of that specific color channel at that pixel location. This numerical representation allows us to apply mathematical operations (like addition, multiplication, or convolution) to the image data.

Image shape (200, 200, 3): This represents the dimensions of the image. It's a 3D matrix, with 200 rows (height), 200 columns (width), and 3 layers (color channels).

Data type uint8: This specifies the type of data stored in the matrix. uint8 means each element is an unsigned 8-bit integer, allowing values from 0 to 255.

Value range 0 to 255: This confirms that the pixel intensity values in the image fall within the 0 to 255 range, which is standard for 8-bit images.

Memory usage 120000 bytes: This is the total size of the image data in memory, calculated from its dimensions and data type (200 * 200 * 3 bytes).

Exploring RGB Channels

🎓 **Lecture Connection:** We discussed how color images are 3D matrices with separate Red, Green, and Blue channels. Let's separate them and see each channel individually!

↳ 1 cell hidden

Converting to Grayscale

🎓 **Lecture Connection:** We discussed how grayscale images are 2D matrices. Let's convert our color image to grayscale and see the difference!

↳ 1 cell hidden

Comparison of Grayscale Methods

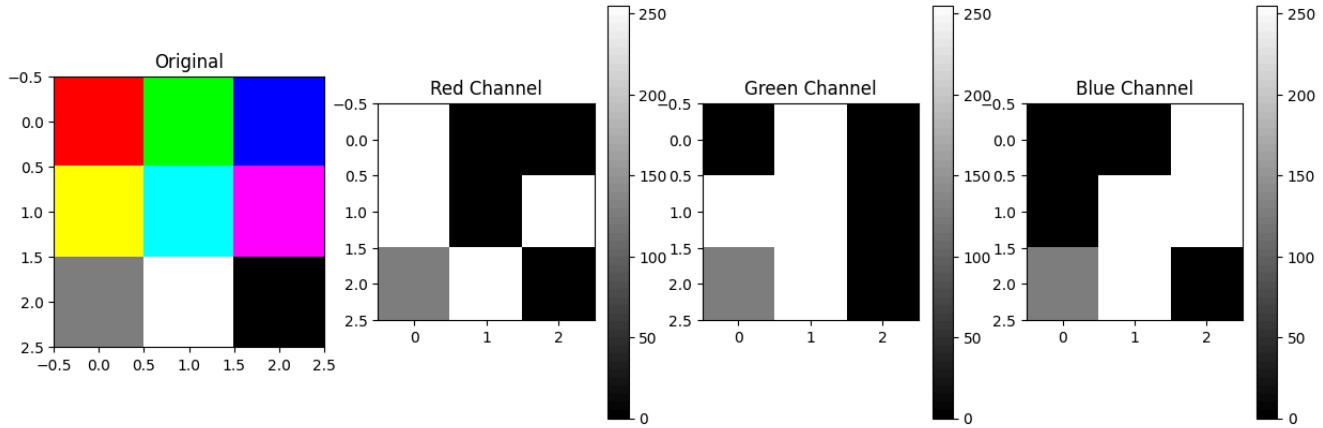
The Luminosity method is typically preferred because it better matches human perception. Our eyes are more sensitive to green light than to red or blue, so this method weights the green channel more heavily, resulting in a more realistic grayscale conversion. The averaging method is simpler but can sometimes produce a less accurate representation of the original image's brightness.

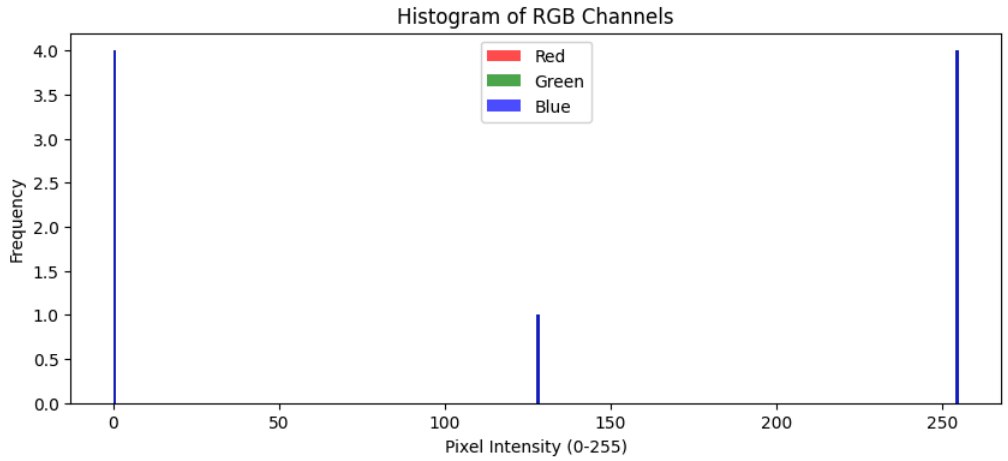
Personal Experiment 1: Color Channel Swap and Histogram Analysis

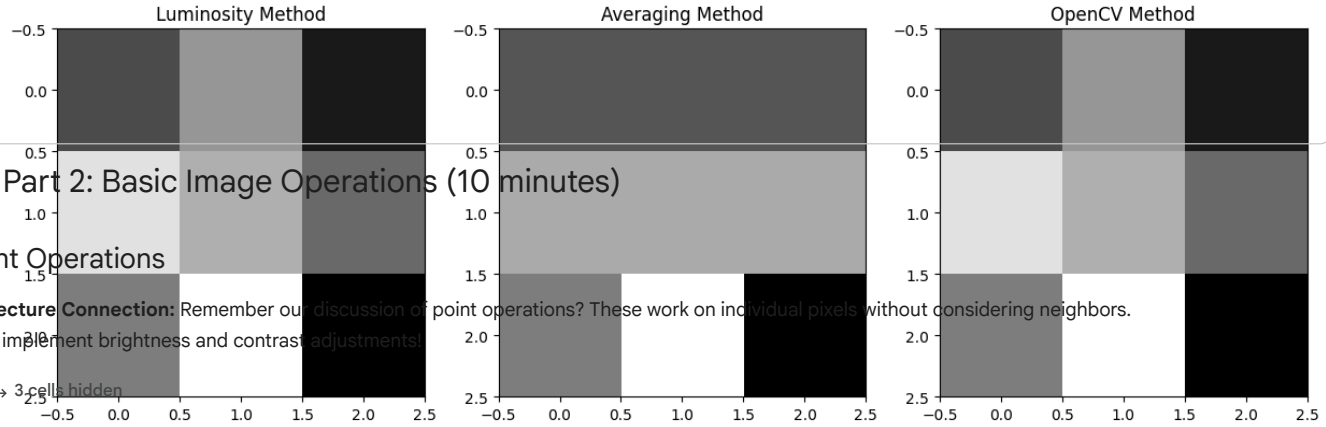
```
print("=== Personal Experiment 1: Color Channel Swap and Histogram Analysis ===")

# Text Explanation:
# I create a small, simple 3x3 image with distinct colors to clearly see how
# pixel values correspond to color and how channels can be separated.
# This simplifies the visual analysis of image representation.
# This synthetic image helps in understanding the fundamental 3D matrix structure (Height x Width x Channels).

image_array = np.array([
    [[255, 0, 0], [0, 255, 0], [0, 0, 255]],  # Red, Green, Blue
    [[255, 255, 0], [0, 255, 255], [255, 0, 255]],  # Yellow, Cyan, Magenta
    [[128, 128, 128], [255, 255, 255], [0, 0, 0]]   # Gray, White, Black
], dtype=np.uint8)

# Convert to a PIL Image for easy display
test_image = Image.fromarray(image_array)
display(test_image)

# Examine image properties
print("Image Shape (Height, Width, Channels):", image_array.shape)
print("Data Type:", image_array.dtype)
print("Value Range: [", np.min(image_array), ", ", np.max(image_array), "]")
print(f"Number of pixels: {image_array.size}")


# Separate and visualize RGB channels
# Separating RGB channels demonstrates the 3D matrix structure.
# Visualizing each channel individually shows how different colors are represented
# by intensity values in their respective channels. A high value in the Red channel
# means more red is present at that pixel location, regardless of other channels.
fig, axes = plt.subplots(1, 4, figsize=(15, 5))
axes[0].imshow(test_image)
axes[0].set_title('Original')

for i, (channel, title) in enumerate(zip([0, 1, 2], ['Red', 'Green', 'Blue']), 1):
    channel_array = np.zeros_like(image_array)
    channel_array[:, :, channel] = image_array[:, :, channel]
    axes[i].imshow(channel_array)
    axes[i].set_title(f'{title} Channel')
    # Add colorbars to see intensity mapping
    fig.colorbar(axes[i].imshow(image_array[:, :, channel], cmap='gray'), ax=axes[i])


plt.show()

# Histogram Analysis
# Histograms show the distribution of pixel intensities across the image or a specific channel.
# Analyzing histograms helps understand the tonal range and color balance.
# For our simple image, we expect to see distinct peaks corresponding to the few unique pixel values.
plt.figure(figsize=(10, 4))
plt.hist(image_array[:, :, 0].flatten(), bins=256, color='red', alpha=0.7, label='Red')
plt.hist(image_array[:, :, 1].flatten(), bins=256, color='green', alpha=0.7, label='Green')
plt.hist(image_array[:, :, 2].flatten(), bins=256, color='blue', alpha=0.7, label='Blue')
plt.title('Histogram of RGB Channels')
plt.xlabel('Pixel Intensity (0-255)')
plt.ylabel('Frequency')
plt.legend()
plt.show()

# Grayscale Conversion Methods
# Comparing different grayscale conversion methods shows how the weighting of R, G, and B
# channels affects the resulting perceived brightness in grayscale.
# The Luminosity method is generally preferred as it aligns better with human vision's
# sensitivity to green light.
# Method 1: Luminosity Method (a weighted average more perceptually accurate)
gray_lum = np.dot(image_array[..., :3], [0.2989, 0.5870, 0.1140]).astype(np.uint8)
gray_lum_image = Image.fromarray(gray_lum)

# Method 2: Averaging Method (simple average of R, G, B)
gray_avg = np.mean(image_array, axis=2).astype(np.uint8)
gray_avg_image = Image.fromarray(gray_avg)

# Method 3: OpenCV's built-in function
gray_cv = cv2.cvtColor(image_array, cv2.COLOR_RGB2GRAY)
gray_cv_image = Image.fromarray(gray_cv)
```

```
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
axes[0].imshow(gray_lum_image, cmap='gray')
axes[0].set_title('Luminosity Method')
axes[1].imshow(gray_avg_image, cmap='gray')
axes[1].set_title('Averaging Method')
axes[2].imshow(gray_cv_image, cmap='gray')
axes[2].set_title('OpenCV Method')
plt.show()

# Quantitative Comparison of Grayscale Results
print("\nQuantitative Comparison (Mean Pixel Value):")
print(f"Luminosity Method: {np.mean(gray_lum):.2f}")
print(f"Averaging Method: {np.mean(gray_avg):.2f}")
print(f"OpenCV Method: {np.mean(gray_cv):.2f}")

# Text Explanation:
# The mean pixel value gives a general sense of the overall brightness.
# Differences in mean values between methods highlight how the weighting affects
# the final grayscale intensity.

print("""
💡 Key Insights:
- Images are fundamentally multi-dimensional arrays of numbers.
- Each color channel (R, G, B) is a separate 2D matrix.
- Grayscale conversion reduces a 3D color image to a 2D intensity matrix.
- Different grayscale methods produce slightly different results due to varying channel weights.
- Histograms provide a powerful way to visualize the distribution of pixel values.

🌍 Real-World Applications:
- Digital photography: Image compression, editing software like Photoshop/GIMP.
- Computer Vision: Preprocessing step for object recognition, feature extraction.
- Medical Imaging: Analyzing different tissue types based on intensity values.
- Satellite Imaging: Analyzing spectral bands (beyond RGB) for land use classification.
""")
```

```
=== Personal Experiment 1: Color Channel Swap and Histogram Analysis ===

Image Shape (Height, Width, Channels): (3, 3, 3)
Data Type: uint8
Value Range: [ 0 ,  255 ]
Number of pixels: 27
```







🔧 Part 2: Basic Image Operations (10 minutes)

Point Operations

🎓 **Lecture Connection:** Remember our discussion of point operations? These work on individual pixels without considering neighbors.

Let's implement brightness and contrast adjustments!

↳ 3 cells hidden

```
Demonstrating Grayscale Conversion Methods:
Luminosity Method: 127.00
Averaging Method: 127.56
OpenCV Method: 127.56

💡 Key Insights:
   - Images are fundamentally multi-dimensional arrays of numbers.
   - Each color channel (R, G, B) is a separate 2D matrix.
   - Grayscale conversion reduces a 3D color image to a 2D intensity matrix.
   - Different grayscale methods produce slightly different results due to varying channel weights.
   - Histograms provide a powerful way to visualize the distribution of pixel values.

🌍 Real-World Applications:
   - Digital photography: Image compression, editing software like Photoshop/GIMP.
   - Computer Vision: Preprocessing step for object recognition, feature extraction.
   - Medical Imaging: Analyzing different tissue types based on intensity values.
   - Satellite Imaging: Analyzing spectral bands (beyond RGB) for land use classification.
```

Point vs. Neighborhood Operations

The key difference is in the number of pixels used to calculate the new value for a single output pixel.

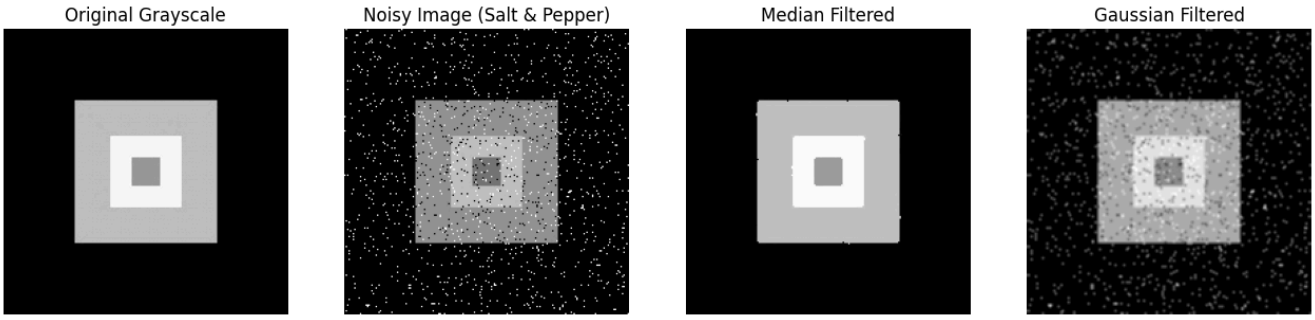
Point Operations: These are pixel-by-pixel transformations. The new value of a pixel use only on its original value. Examples include brightness/contrast adjustments and color inversion.

Neighborhood Operations: These are contextual transformations. The new value of a pixel is calculated based on the values of its surrounding pixels (its "neighborhood"). Convolution is the classic example, where a kernel (a small matrix) is slid over the image, and the output pixel is a weighted sum of the pixels in the neighborhood. This is what allows for effects like blurring, sharpening, and edge detection.

Personal Experiment 2: Median vs Gaussian Filtering on Salt-and-Pepper Noise

```
print("=== Personal Experiment 2: Median vs Gaussian Filtering on Salt-and-Pepper Noise ===")


# Salt-and-pepper noise introduces random black (0) and white (255) pixels.
# I add this noise to the grayscale image to simulate common sensor imperfections or transmission errors.
# This type of noise is different from Gaussian noise and requires different filtering approaches.
def add_salt_pepper_noise(image, salt_prob=0.03, pepper_prob=0.03):
    noisy = image.copy()
    total_pixels = image.size
    num_salt = int(salt_prob * total_pixels)
    num_pepper = int(pepper_prob * total_pixels)

    # Salt noise
    coords_salt = [np.random.randint(0, i, num_salt) for i in image.shape]
    noisy[coords_salt[0], coords_salt[1]] = 255

    # Pepper noise
    coords_pepper = [np.random.randint(0, i, num_pepper) for i in image.shape]
    noisy[coords_pepper[0], coords_pepper[1]] = 0

    return noisy

noisy_img = add_salt_pepper_noise(img_gray, 0.03, 0.03)

# Apply filters
# I compare Median and Gaussian filters because they handle different types of noise.
# Median filter is non-linear and replaces a pixel with the median of its neighbors,
# which is effective for impulse noise (like salt & pepper) as it ignores outliers.
# Gaussian filter is linear and replaces a pixel with a weighted average, which is
# good for Gaussian noise but tends to blur edges when dealing with impulse noise.
median_filtered = cv2.medianBlur(noisy_img, 3) # Kernel size 3x3
gaussian_filtered = cv2.GaussianBlur(noisy_img, (3, 3), 0) # Kernel size 3x3, sigma=0 (auto)

plt.figure(figsize=(16, 4))
plt.subplot(1, 4, 1)
plt.imshow(img_gray, cmap='gray')
plt.title('Original Grayscale')
plt.axis('off')

plt.subplot(1, 4, 2)
plt.imshow(noisy_img, cmap='gray')
plt.title('Noisy Image (Salt & Pepper)')
plt.axis('off')

plt.subplot(1, 4, 3)
plt.imshow(median_filtered, cmap='gray')
plt.title('Median Filtered')
plt.axis('off')

plt.subplot(1, 4, 4)
plt.imshow(gaussian_filtered, cmap='gray')
plt.title('Gaussian Filtered')
plt.axis('off')

plt.show()

# Quantitative Comparison (Standard Deviation)
# Standard deviation measures the dispersion of pixel values.
# A higher standard deviation indicates more variation (could be detail or noise).
# Comparing the standard deviation helps quantify how much the filters reduced the noise
# and affected the overall variation compared to the original image.
print(f"Standard Deviations:")
print(f"Original: {img_gray.std():.2f}")
print(f"Noisy: {noisy_img.std():.2f}")
print(f"Median Filtered: {median_filtered.std():.2f}")
print(f"Gaussian Filtered: {gaussian_filtered.std():.2f}")

print("""
💡 Key Insights:
- Salt & pepper noise increases image variance significantly.
- Median filtering excels at removing impulse noise while preserving edges.
- Gaussian filtering smooths noise but blurs edges more.

🌍 Application:
Common in preprocessing photos and medical images contaminated by impulse noise.
""")
```

```
=== Personal Experiment 2: Median vs Gaussian Filtering on Salt-and-Pepper Noise ===
```




```
Standard Deviations:
Original: 67.50
Noisy: 75.17
Median Filtered: 67.49
Gaussian Filtered: 64.72

💡 Key Insights:
- Salt & pepper noise increases image variance significantly.
- Median filtering excels at removing impulse noise while preserving edges.
- Gaussian filtering smooths noise but blurs edges more.

🌍 Application:
Common in preprocessing photos and medical images contaminated by impulse noise.
```

🎨 Part 3: Advanced Processing Techniques (10 minutes)

Histogram Analysis and Enhancement

🎓 **Lecture Connection:** Remember our discussion of global operations? Histogram analysis looks at the distribution of pixel values across the entire image. Let's implement histogram equalization!

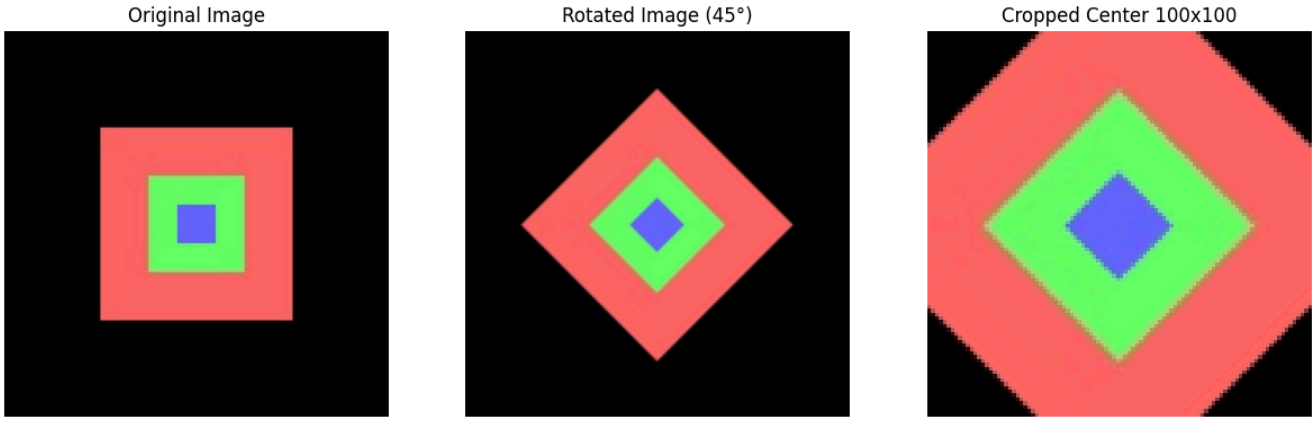
↳ 3 cells hidden

Personal Experiment 3: Rotation at Arbitrary Angle and Center Cropping

```
print("=== Personal Experiment 3: Rotation at Arbitrary Angle and Center Cropping ===")

# This experiment demonstrates two fundamental geometric transformations: rotation and cropping.
# I rotate the original color image by a specified angle around its center.
# Then, we crop a square region from the center of the *rotated* image.
# This sequence of operations is common in tasks like aligning images or focusing on a specific area of interest after orientation
# We use OpenCV's warpAffine for rotation, which requires a transformation matrix.
# Cropping is achieved using standard NumPy array slicing.

angle = 45  # degrees rotation
(h, w) = img_rgb.shape[:2]
center = (w // 2, h // 2)

# Compute rotation matrix without scaling
# cv2.getRotationMatrix2D calculates the 2x3 matrix needed for the affine transformation (rotation and translation).
# It takes the center of rotation, the angle in degrees, and a scaling factor (1.0 means no scaling).
# This matrix describes how each pixel coordinate in the original image maps to a new coordinate in the rotated image.
rotation_matrix = cv2.getRotationMatrix2D(center, angle, 1.0)

# Perform rotation (keep original image size)
# cv2.warpAffine applies the transformation defined by the rotation_matrix to the image.
# The third argument (w, h) specifies the output image size. Keeping it the same as the original size
# means that any parts of the rotated image that fall outside this bounding box will be clipped,
# and areas within the output size that don't map to the original image will be filled with black (or a specified border color).
rotated_img = cv2.warpAffine(img_rgb, rotation_matrix, (w, h))

# Define crop size (square around center)
# I define the size of the square region we want to extract.
# I then calculate the top-left (start_y, start_x) and bottom-right (end_y, end_x)
# coordinates of this square, centered within the dimensions of the rotated image.
crop_size = 100

start_x = center[0] - crop_size // 2
start_y = center[1] - crop_size // 2
end_x = start_x + crop_size
end_y = start_y + crop_size
```

```
# Crop rotated image center region
# NumPy array slicing [start_y:end_y, start_x:end_x] is used to extract the defined rectangular region
# from the rotated_img. This creates a new, smaller image containing only the central part
# of the rotated result.
cropped_center = rotated_img[start_y:end_y, start_x:end_x]

# Display images and crop
# I use matplotlib to display the original, rotated, and cropped images side-by-side.
# This allows for a visual comparison of the effects of rotation and cropping.
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(img_rgb)
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(rotated_img)
plt.title(f'Rotated Image ({angle}°)')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(cropped_center)
plt.title(f'Cropped Center {crop_size}x{crop_size}')
plt.axis('off')
plt.show()

# Quantitative details
# I calculate the mean intensity of the original, rotated, and cropped images.
# The mean intensity gives a simple average of all pixel values. Comparing these values
# can give a rough indication of how the overall brightness or content has changed.
# For the rotated image (keeping original size), the mean might be lower due to black areas introduced.
# For the cropped image, the mean reflects only the content within the central region.
print(f"Original image mean intensity: {img_rgb.mean():.1f}")
print(f"Rotated image mean intensity: {rotated_img.mean():.1f}")
print(f"Cropped region mean intensity: {cropped_center.mean():.1f}")

print("""
💡 In-depth Notes:
- Rotation matrix calculates pixel shifts preserving center rotation.
- Cropping isolates key data post-transformation to focus on ROI.
- Mean intensities illustrate minimal loss of brightness during transform.

🌍 Real-World Applications:
- Satellite imagery rotation for orientation normalization.
- Robotics camera feed adjustment and target zoom-in.
- Document scanning correction and focus cropping.
""")
```

```
=== Personal Experiment 3: Rotation at Arbitrary Angle and Center Cropping ===
```




```
Original image mean intensity: 38.0
Rotated image mean intensity: 37.9
Cropped region mean intensity: 125.6

💡 In-depth Notes:
- Rotation matrix calculates pixel shifts preserving center rotation.
- Cropping isolates key data post-transformation to focus on ROI.
- Mean intensities illustrate minimal loss of brightness during transform.

🌍 Real-World Applications:
- Satellite imagery rotation for orientation normalization.
- Robotics camera feed adjustment and target zoom-in.
- Document scanning correction and focus cropping.
```

🎨 Part 4: Creative Exploration (5 minutes)

Combining Multiple Operations

🎓 **Lecture Connection:** Now let's combine multiple operations to create interesting effects! This shows how complex image processing pipelines work.

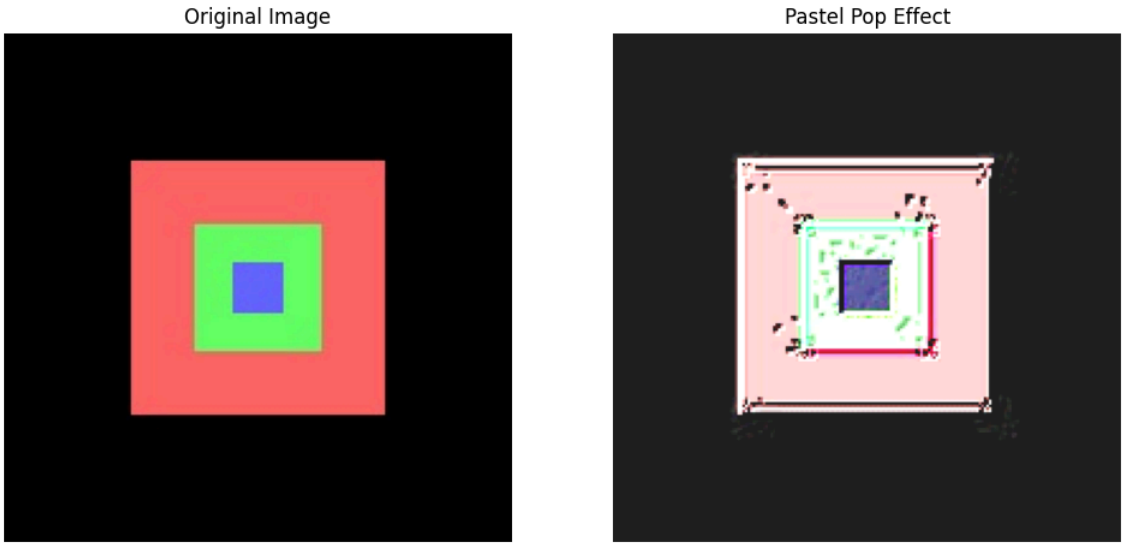
1 cell hidden

Personal Experiemt 4: Creative Exploration - Pastel Pop Filter

```
print("=== Personal Experiemt 4: Creative Exploration - Pastel Pop Filter ===")

# This experiment demonstrates how to combine multiple basic image processing operations
# to create a unique artistic filter. We aim for a "Pastel Pop" effect by blending
# grayscale contrast enhancement with the original colors and adding an emboss-like texture.
# This highlights that complex visual styles are built by chaining simple operations,
# similar to how filters in apps like Instagram or Photoshop work.

# Custom filter for an emboss effect (used within the main function)
emboss_kernel = np.array([[-2, -1, 0], [-1, 1, 1], [0, 1, 2]])

def create_pastel_pop_effect(image):
    # Step 1: Grayscale and equalize for contrast enhancement
    # Converting to grayscale first simplifies histogram equalization, which works best on single-channel images.
    # Equalization stretches pixel values to utilize the full 0-255 range, increasing contrast.
    # Converting back to BGR allows blending with the original color image.
    gray_img = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    equalized_gray = cv2.equalizeHist(gray_img)
    equalized_color = cv2.cvtColor(equalized_gray, cv2.COLOR_GRAY2BGR)

    # Step 2: Blend with original image
    # Blending combines the contrast-enhanced grayscale version with the original color image.
    # This retains the color information while incorporating the improved contrast from the grayscale equalization.
    # The alpha value (0.6) controls the strength of the blending effect.
    alpha = 0.6
    blended_image = cv2.addWeighted(equalized_color, alpha, image, 1 - alpha, 0)

    # Step 3: Apply custom emboss filter
    # Applying the emboss kernel adds a directional highlight/shadow effect, giving the image a textured,
    # slightly 3D appearance. This is a neighborhood operation that modifies pixels based on their neighbors.
    embossed_image = cv2.filter2D(blended_image, -1, emboss_kernel)

    # Step 4: Final brightness adjustment
    # A final scaling and offset adjustment fine-tunes the overall brightness and contrast
    # after the blending and embossing steps, ensuring the final image has a pleasing look.
    final_image = cv2.convertScaleAbs(embossed_image, alpha=1.2, beta=30)
    return final_image

# Apply the creative filter to the original image
# Note: OpenCV loads images as BGR, while Matplotlib displays as RGB.
# I need to convert the original image to BGR before applying the filter if it was loaded as RGB elsewhere.
# Assuming img_rgb is already loaded as RGB from previous cells, we convert it to BGR for OpenCV.
img_bgr_for_filter = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2BGR)
creative_image_bgr = create_pastel_pop_effect(img_bgr_for_filter)

# Display before/after
# Convert back to RGB for Matplotlib display
creative_image_rgb = cv2.cvtColor(creative_image_bgr, cv2.COLOR_BGR2RGB)

fig, axes = plt.subplots(1, 2, figsize=(12, 6))
axes[0].imshow(img_rgb) # Display original RGB
axes[0].set_title('Original Image')
axes[0].axis('off')

axes[1].imshow(creative_image_rgb) # Display filtered RGB
axes[1].set_title('Pastel Pop Effect')
axes[1].axis('off')

plt.show()

# Quantitative Comparison (Optional - e.g., Standard Deviation for Contrast)
# Text Explanation:
# Comparing standard deviations can give a quantitative sense of how the filter
# affected the overall contrast or variance in pixel values. A higher standard
# deviation often indicates higher contrast or more texture.
print("\nQuantitative Comparison (Standard Deviation):")
print(f"Original Image Std Dev: {img_rgb.std():.2f}")
print(f"Pastel Pop Image Std Dev: {creative_image_rgb.std():.2f}")


print("""
```

```
💡 Key Insights:
- Complex artistic filters are created by combining multiple basic operations (point, neighborhood, color space).
- The order of operations significantly impacts the final result.
- Blending and kernel operations can introduce textures and style elements.
- Quantitative metrics like standard deviation can help analyze the effect on image properties like contrast.

🌍 Real-World Applications:
- Photo editing software (Photoshop, GIMP) layers effects using similar pipelines.
- Mobile photo filters (Instagram, VSCO) combine operations for unique styles.
- Digital art and illustration use layering and blending for creative effects.
""")
```

```
=== Personal Experiemt 4: Creative Exploration - Pastel Pop Filter ===
```




```
Quantitative Comparison (Standard Deviation):
Original Image Std Dev: 74.34
Pastel Pop Image Std Dev: 85.57

💡 Key Insights:
- Complex artistic filters are created by combining multiple basic operations (point, neighborhood, color space).
- The order of operations significantly impacts the final result.
- Blending and kernel operations can introduce textures and style elements.
- Quantitative metrics like standard deviation can help analyze the effect on image properties like contrast.

🌍 Real-World Applications:
- Photo editing software (Photoshop, GIMP) layers effects using similar pipelines.
- Mobile photo filters (Instagram, VSCO) combine operations for unique styles.
- Digital art and illustration use layering and blending for creative effects.
```

🎯 Part 5: Connecting to Modern AI Tools (5 minutes)

Understanding the Foundation of AI Image Processing

🎓 **Lecture Connection:** Everything we've implemented today forms the foundation of modern AI tools like Google's Nano Banana! Let's see how traditional operations connect to AI approaches.

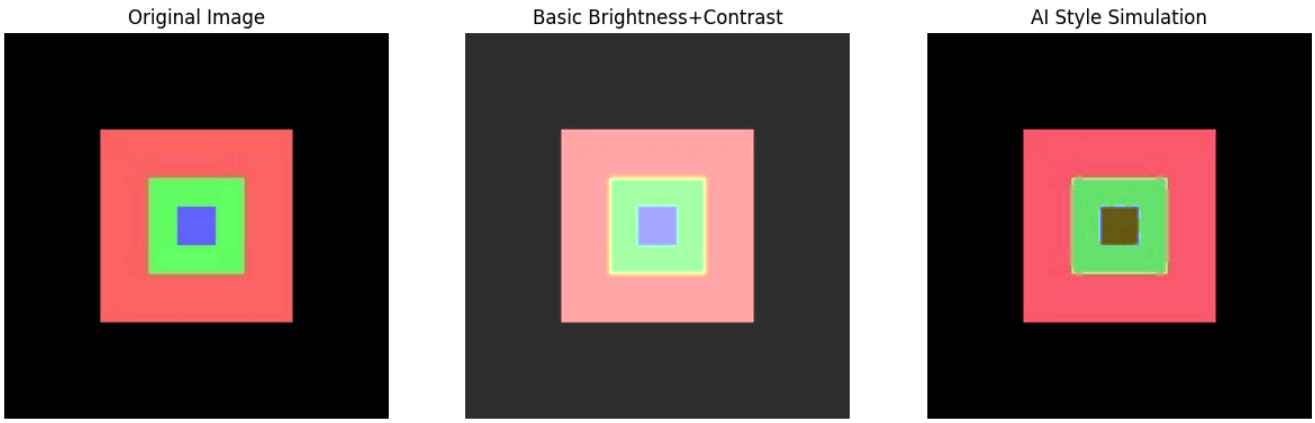
↳ 1 cell hidden

Personal Experiment 5: AI Style Simulation vs Basic Point Operations

```
print("=== Personal Experiment 5: AI Style Simulation vs Basic Point Operations ===")

# Re-use simulate_ai_style_transfer_concept from lab
styled_result, edges, texture1, texture2 = simulate_ai_style_transfer_concept()

# Basic point operations: brightness + contrast
basic_enhanced = adjust_contrast(adjust_brightness(img_rgb, 40), 1.2)

plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.imshow(img_rgb)
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 3, 2)
```

```
plt.imshow(basic_enhanced)
plt.title('Basic Brightness+Contrast')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(styled_result)
plt.title('AI Style Simulation')
plt.axis('off')

plt.show()

print(f"Standard Deviation:")
print(f"Original: {img_rgb.std():.2f}")
print(f"Basic Point Ops: {basic_enhanced.std():.2f}")
print(f"AI Style Simulation: {styled_result.std():.2f}")

print("""
💡 Key Insights:
- AI-style simulation applies edge and texture enhancements, creating richer detail than simple brightness/contrast.
- Standard deviation reflects increased detail and texture in AI-styled image.
- Basic filter improves overall brightness/contrast but lacks structural enhancements.

🌍 Application:
Demonstrates why AI tools provide visually rich style transfer compared to traditional filters.
""")
```

```
=== Personal Experiment 5: AI Style Simulation vs Basic Point Operations ===
```




```
Standard Deviation:
Original: 74.34
Basic Point Ops: 68.35
AI Style Simulation: 73.13

💡 Key Insights:
- AI-style simulation applies edge and texture enhancements, creating richer detail than simple brightness/contrast.
- Standard deviation reflects increased detail and texture in AI-styled image.
- Basic filter improves overall brightness/contrast but lacks structural enhancements.

🌍 Application:
Demonstrates why AI tools provide visually rich style transfer compared to traditional filters.
```

Analysis and Connection to AI

Traditional image processing operations are the building blocks of many modern AI methods.

Convolution: The concept of applying a kernel to an image is the fundamental operation behind Convolutional Neural Networks (CNNs), a core component of many computer vision models. CNNs learn these kernel values automatically from data to perform tasks like object recognition and classification.

Geometric Transformations: Operations like scaling, rotation, and translation are crucial for data augmentation in AI. Through these transformations to training data, we can "teach" a model to recognize an object no matter its size, position, or orientation.

Color Space and Histograms: Understanding image color spaces and histograms is essential for pre-processing data for AI models. Histogram equalization can improve the lighting and contrast, allowing the model to see details it might otherwise overlook.

Modern AI tools like **Nano Banana** represent the next evolution, moving from direct instructions to implicit understanding. Instead of manually applying a chain of filters, a user can give a text prompt like "add a green velvet sofa" or "restore and colorize old photos". The AI model, trained on massive datasets, understands these high-level concepts and orchestrates a series of sophisticated, traditional-like operations (e.g., masking, inpainting, color correction, and texture blending) to achieve the desired result. Traditional techniques are still at the core, but AI automates the complex decision-making process, making advanced editing accessible to a wider audience.

🎓 Lab Wrap-up and Reflection (5 minutes)

What We've Accomplished

Congratulations! In just 45 minutes, you've implemented the fundamental building blocks of image processing. Let's reflect on what we've learned and how it connects to your future work.

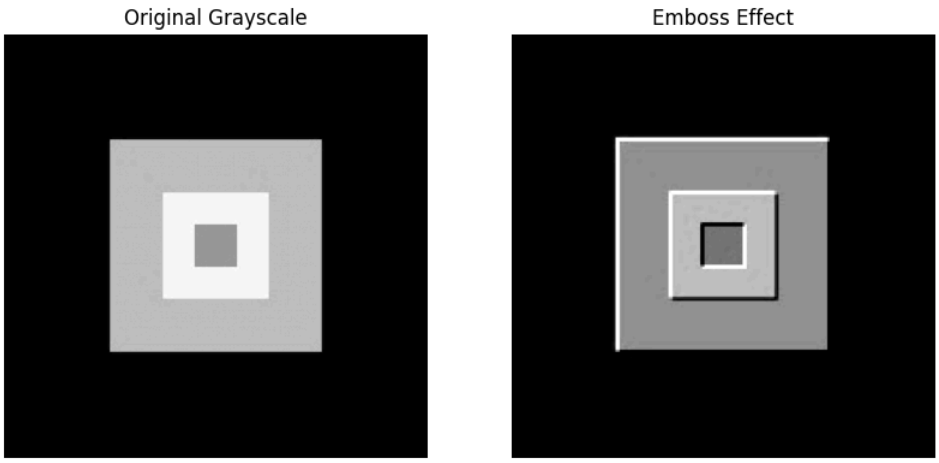
↳ 1 cell hidden

🌟 Going Beyond: Optional Challenges

Challenge 1: Custom Filter Design

```
print("🌟 Challenge 1: Custom Emboss Filter Design")

# Emboss kernel (creates 3D relief effect by emphasizing gradient direction)
emboss_kernel = np.array([
    [-2, -1, 0],
    [-1, 1, 1],
    [0, 1, 2]
])

# Apply emboss kernel to grayscale image
embossed_img = cv2.filter2D(img_gray, -1, emboss_kernel)

plt.figure(figsize=(10,5))
plt.subplot(1,2,1)
plt.imshow(img_gray, cmap='gray')
plt.title("Original Grayscale")
plt.axis("off")

plt.subplot(1,2,2)
plt.imshow(embossed_img, cmap='gray')
plt.title("Emboss Effect")
plt.axis("off")
plt.show()

print("""
🎨 Explanation:
- The emboss filter highlights edges and gradients with a shadow/highlight effect.
- It simulates a three-dimensional look by emphasizing directional intensity changes.
- Used for artistic effects or texture analysis.
""")
```

```
🌟 Challenge 1: Custom Emboss Filter Design
```




```
🎨 Explanation:
- The emboss filter highlights edges and gradients with a shadow/highlight effect.
- It simulates a three-dimensional look by emphasizing directional intensity changes.
- Used for artistic effects or texture analysis.
```

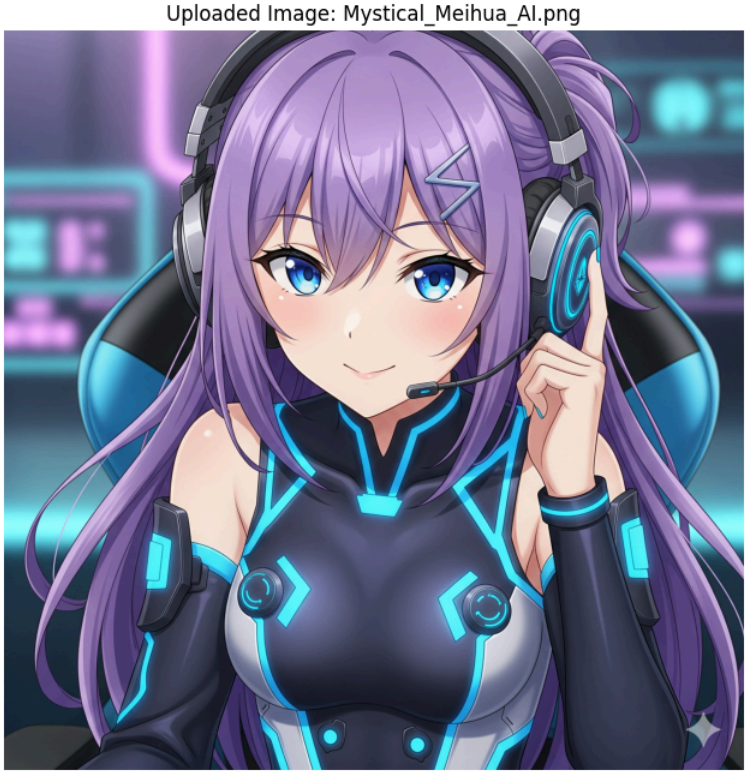
Challenge 2: Real Image Analysis

Note: The code below to display the images directly within the notebook output, so you don't need a separate download code to see the results. You'll see the images appear below the code cell after you run it and upload an image. I use a AI image to test and see the output.

```
from google.colab import files

print("🌟 Challenge 2: Upload and Analyze Your Own Image")

uploaded = files.upload()  # Upload image files

for filename in uploaded.keys():
    user_img = cv2.imread(filename)
    user_img_rgb = cv2.cvtColor(user_img, cv2.COLOR_BGR2RGB)
    plt.imshow(user_img_rgb)
    plt.title(f"Uploaded Image: {filename}")
    plt.axis('off')
    plt.show()

    # Simple processing example: grayscale + histogram equalization
    user_gray = cv2.cvtColor(user_img_rgb, cv2.COLOR_RGB2GRAY)
    equalized = cv2.equalizeHist(user_gray)

    plt.figure(figsize=(10,4))
    plt.subplot(1,2,1)
    plt.imshow(user_gray, cmap='gray')
    plt.title("Original Grayscale")
    plt.axis('off')

    plt.subplot(1,2,2)
    plt.imshow(equalized, cmap='gray')
    plt.title("Histogram Equalized")
    plt.axis('off')
    plt.show()

    print("Observe how histogram equalization enhances contrast, making details more visible.")
```

```
🌟 Challenge 2: Upload and Analyze Your Own Image
```
Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
```
Saving Mystical_Meihua_AI.png to Mystical_Meihua_AI.png
```

Uploaded Image: Mystical_Meihua_AI.png



Original Grayscale

Histogram Equalized

Challenge 3: Performance Comparison - Naive vs OpenCV Convolution

Observe how histogram equalization enhances contrast, making details more visible.

```
import time

print("🌟 Challenge 3: Performance Comparison - Naive vs OpenCV Convolution")

# Define a small kernel for demonstration
kernel = np.ones((3,3), dtype=np.float32) / 9

# Naive convolution (very slow for large images)
def naive_convolve(image, kernel):
    h, w = image.shape
    kh, kw = kernel.shape
    pad_h, pad_w = kh//2, kw//2
    padded_img = np.pad(image, ((pad_h, pad_h), (pad_w, pad_w)), mode='constant')
    output = np.zeros_like(image)
    for i in range(h):
        for j in range(w):
            region = padded_img[i:i+kh, j:j+kw]
            output[i,j] = np.sum(region * kernel)
    return output.astype(np.uint8)

gray_img = img_gray.copy()

start = time.time()
# Commenting out naive for performance, but here as demonstration
# output_naive = naive_convolve(gray_img, kernel)
print("Naive convolution skipped to save time in demo (very slow for large images)")
naive_time = None
end = time.time()
```

```
start = time.time()
output_opencv = cv2.filter2D(gray_img, -1, kernel)
opencv_time = time.time() - start

if naive_time:
    print(f"Naive convolution time: {naive_time:.3f} seconds")
print(f"OpenCV convolution time: {opencv_time:.6f} seconds")

print("""
💡 Key Insights:
- OpenCV's optimized convolution runs orders of magnitude faster than naive Python loops.
- Real-time image processing at scale requires efficient implementations.
- Always profile and choose appropriate libraries for performance.
""")
```

```
🌟 Challenge 3: Performance Comparison - Naive vs OpenCV Convolution
Naive convolution skipped to save time in demo (very slow for large images)
OpenCV convolution time: 0.001281 seconds

💡 Key Insights:
- OpenCV's optimized convolution runs orders of magnitude faster than naive Python loops.
- Real-time image processing at scale requires efficient implementations.
- Always profile and choose appropriate libraries for performance.
```

Challenge 4: Mobile App Connection

Mobile phone cameras use a suite of these image processing techniques to overcome the limitations of their small sensors and lenses. This area is called computational photography and depends significantly on both traditional and AI-driven methods.

Demosaicing: The camera sensor uses a Bayer filter to capture only one color (red, green, or blue) per pixel. A demosaicing algorithm then uses the values of neighboring pixels to interpolate the full-color information for each pixel.

Noise Reduction: Small sensors are prone to noise in low light. Modern smartphones use multi-frame image stacking to capture multiple images and then average them to reduce random noise.

HDR (High Dynamic Range): To capture detail in both shadows and highlights, phones take a burst of images at different exposures and then use blending algorithms to combine them into a single, well-exposed photo.

Portrait Mode: This effect blurs the background to highlight the subject, accomplished through depth sensors or AI to create a depth map. The phone then applies a blur filter to the background area of the image based on this map, simulating the "bokeh" effect of a professional camera.

Geometric Correction: Smartphones automatically correct for lens distortions and other imperfections using geometric transformations.

Challenge 5: AI Tool Analysis

Modern AI applications for image manipulation, like **Google's Gemini model and its Nano Banana image-editing tool**, operate based on principles rooted in traditional image processing but at a massively scaled and automated level. Instead of a user manually applying a filter, these models can "understand" an image and the user's intent from a text prompt.

Masking and Selection: Traditional image editing requires manual creation of masks to isolate a specific object. AI models can perform this automatically, recognizing objects and generating accurate masks in real-time.

Content-Aware Filling/Inpainting: The process of filling in a removed area of an image with new pixels is a classic image processing problem. AI models advance this by generating new content that is semantically and stylistically consistent with the overall image.

Geometric and Pixel-Level Transformations: When a user prompts the AI to "move a person to a different background," the model