محمدمهدی نوروزی-9923085

گزارش پروژه پردازش تصویر با استفاده از Python و OpenCV

سوال ۱

مقدمه

در این پروژه، هدف ما بررسی و شناسایی تغییرات اعمال شده بر روی یک تصویر اصلی است. پنج تغییر مختلف بر روی تصویر اصلی اعمال شده‌اند و ما نیاز داریم تا این تغییرات را پیدا کنیم. برای انجام این کار از کتابخانه‌های OpenCV و NumPy و matplotlib در Python استفاده کرده‌ایم. در ادامه به تشریح مراحل و روش‌های مورد استفاده می‌پردازیم.

مراحل و روش‌های انجام کار

۱. بارگذاری تصاویر

در ابتدا تصاویر اصلی و تغییر یافته را بارگذاری می‌کنیم.

```
img1 = cv2.imread('./Original_image.jpg', cv2.IMREAD_GRAYSCALE)

img2 = cv2.imread('./transformed_image.jpg', cv2.IMREAD_GRAYSCALE)
```

۲. اعمال بازتاب

اولین تغییری که بر روی تصویر اصلی اعمال می‌کنیم، بازتاب تصویر است. این کار با استفاده از تابع `cv2.flip` انجام می‌شود.

```
def apply_reflection(image):

   reflected_image = cv2.flip(image, 1)

   return reflected_image
```

۳. اعمال تغییر برشی

در این مرحله، تغییر برشی (Shear) بر روی تصویر بازتابی اعمال می‌شود. ماتریس تغییر برشی ایجاد و سپس با استفاده از تابع `cv2.warpAffine` اعمال می‌شود. دلیل استفاده از این تبدیل این است که در تصویر نهایی انحراف خاصی دارد که تنها با چرخش قابل انجام نیست. به عبارت دیگر اگر این مرحله انجام نشود ابعاد x و y دوتصویر درنهایت منطبق نخواهد شد.

```
def apply_shear_transformation(image, shx=0, shy=0):
```

```
    rows, cols = image.shape

    # Define the shear transformation matrix

    shear_matrix = np.array([[1, shx, 0],

                             [shy, 1, -200]], dtype=np.float32)

    # Apply the shear transformation using the transformation matrix

    transformed_image = cv2.warpAffine(image, shear_matrix, (cols, rows))

    return transformed_image
```

۴. مقیاس‌دهی افقی

در این مرحله تصویر برشی مقیاس‌دهی افقی می‌شود. ابعاد جدید تصویر محاسبه و سپس با استفاده از تابع `cv2.resize` تغییر اندازه داده می‌شود. دلیل استفاده از این تبدیل این است که تصویر نهایی دست ابعادی مانند تصور اولیه ندارد و در جهت محور x ابعادش افزایش یافته است. لذا این تغییر را اعمال میکنیم.

```
def scale_image_horizontal(image, s):

    # Get original dimensions

    h, w = image.shape

    h, w = int(h*s[0]), int(w*s[1])

    # Resize the image

    resized_image = cv2.resize(image, (w, h), interpolation=cv2.INTER_LINEAR)

    return resized_image
```

۵. اعمال تبدیل آفین

در این مرحله تبدیل آفین شامل چرخش، مقیاس‌دهی و انتقال بر روی تصویر اعمال می‌شود. ماتریس تغییر آفین با استفاده از تابع `cv2.getRotationMatrix2D` ایجاد و سپس با استفاده از `cv2.warpAffine` اعمال می‌شود. البته از انتقال این تابع استفاده نشده است.

```
def apply_affine_transformation(image, angle, scale, tx, ty):

    # Get the image dimensions
```

```
    rows, cols = image.shape

    # Compute the center of the image

    center = (cols / 2, rows / 2)

    # Compute the transformation matrix

    rotation_matrix = cv2.getRotationMatrix2D(center, angle, scale)

    # Apply the translation to the transformation matrix

    rotation_matrix[0, 2] += tx  # adding translation in x direction

    rotation_matrix[1, 2] += ty  # adding translation in y direction

    # Apply the afficv2.findContours(gray_image, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)ne transformation using the transformation matrix

    transformed_image = cv2.warpAffine(image, rotation_matrix, (cols, rows))

    return transformed_image
```

۶. جابجایی تصویر

در نهایت، تصویر تغییر یافته جابجا می‌شود. ماتریس جابجایی ایجاد و با استفاده از `cv2.warpAffine` اعمال می‌شود.

```
def move_image(image, tx, ty, cols, rows):

    move_matrix = np.array([[1, 0, tx],

                            [0, 1, ty]], dtype=np.float32)

    transformed_image = cv2.warpAffine(image, move_matrix, (cols, rows))

    return transformed_image
```

۷. مقایسه تصاویر

برای مقایسه تصویر نهایی با تصویر تغییر یافته از دو روش `bitwise_or` استفاده می‌کنیم.

```
def bitwise_or_images(image1, image2):
```

```
    if image1.shape != image2.shape:
        print("Error: The dimensions of the images do not match")
        return
    # Perform bitwise OR operation
    result = cv2.bitwise_and(image1, cv2.bitwise_not(image2))
    return result
```

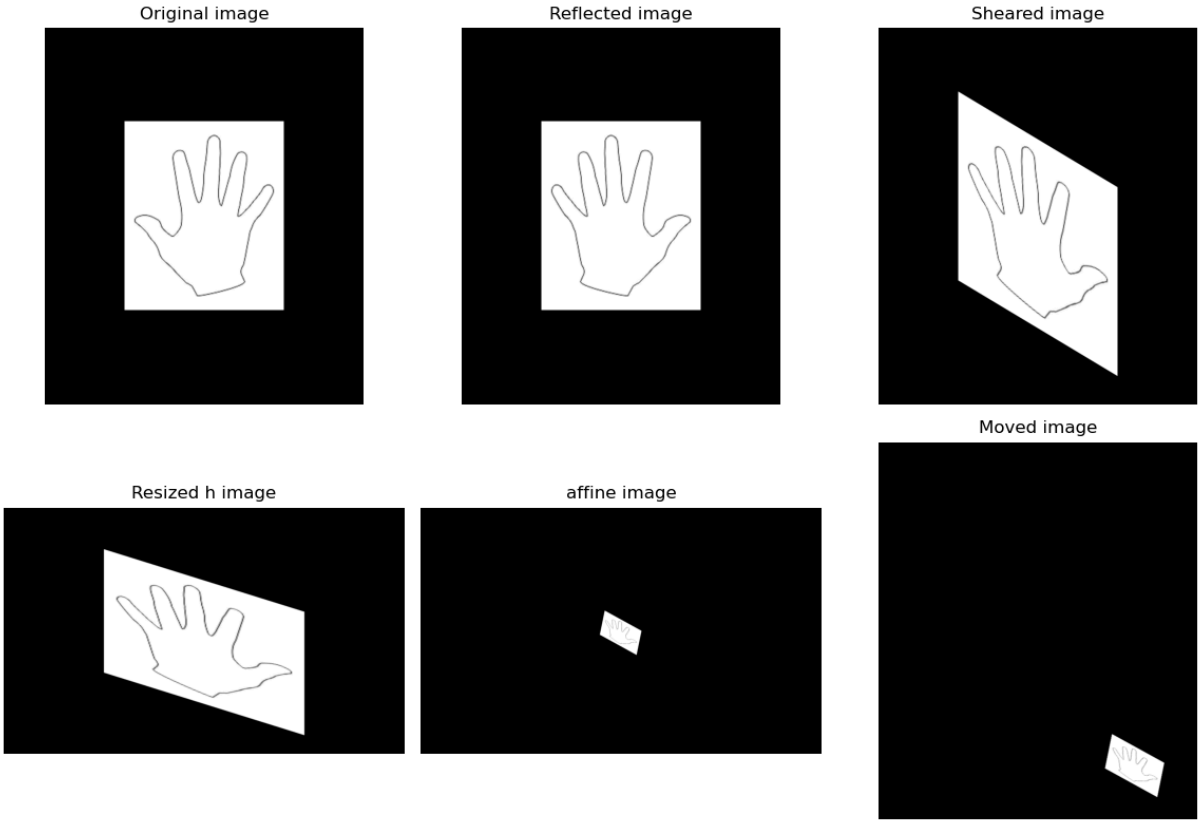
۸. نمایش تصاویر

در نهایت، تصاویر مختلف با استفاده از `matplotlib` نمایش داده می‌شوند.

```
# Create a figure with 2 rows and 3 columns
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
# Display images in the subplots
axes[0, 0].imshow(img1, cmap='gray')
axes[0, 0].set_title("Original Image")
axes[0, 0].axis('off')  # Hide axes
axes[0, 1].imshow(moved_image, cmap='gray')
axes[0, 1].set_title("Result")
axes[0, 1].axis('off')
axes[0, 2].imshow(img2, cmap='gray')
axes[0, 2].set_title("Test")
axes[0, 2].axis('off')
axes[1, 0].imshow(compare1, cmap='gray')
axes[1, 0].set_title("Test & (~Result)")
axes[1, 0].axis('off')
```

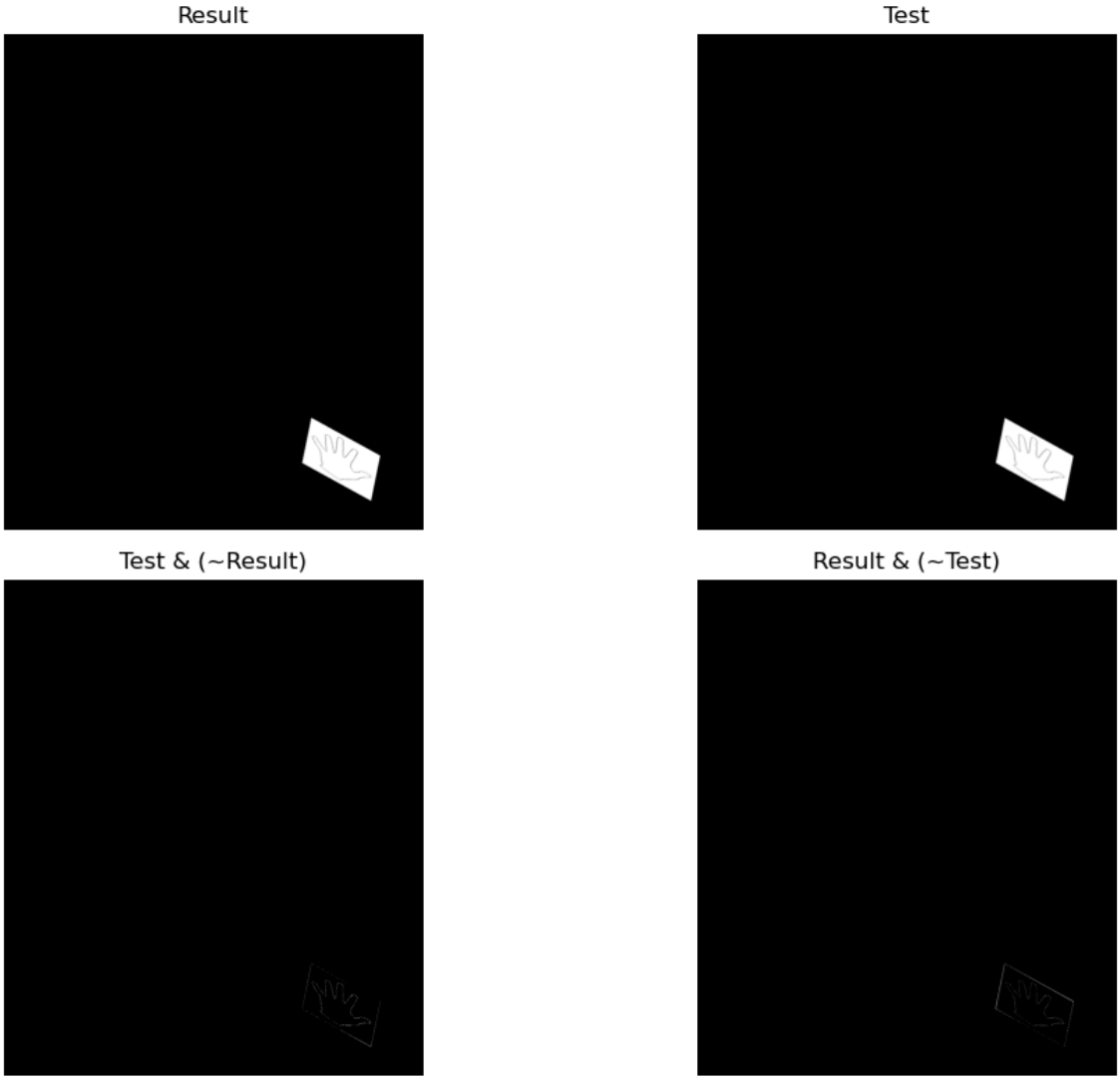
```
axes[1, 1].imshow(compare2, cmap='gray')

axes[1, 1].set_title("Result & (~Test)")

axes[1, 1].axis('off')

# Hide the last subplot (bottom-right) as we don't have a sixth image

axes[1, 2].axis('off')

# Adjust layout

plt.tight_layout()

plt.show()
```

نتیجه هر تبدیل



بررسی نتیجه نهایی

تصویر result تصویری است که در این پروژه به آن رسیدیم. تصویر test تصویری است که باید به آن میرسیدیم. برای مقایسه این دو تصویر همانطور که قبلا بیان شدن not یک تصویر را با دیگری and میکنیم. اگر دوتصویر کامل روی هم منطبق شده باشند تصویر مقایسه باید کامل مشکی باشد. یکبار باید test را با not تصویر result مقایسه کنیم و بار دیگر تصویر result را با not تصویر test. با اینکار دقیق میتوان مقایسه را انجام داد. همانطور که مشاهده میشود دو تصویر بسیار ه هم نزدیک هستند.



نتیجه‌گیری

در این سوال با استفاده از روش‌های مختلف پردازش تصویر، پنج تغییر بر روی یک تصویر اصلی اعمال و سپس این تغییرات با تصویر نهایی مقایسه شدند. این تغییرات شامل بازتاب، تغییر برشی، مقیاس‌دهی افقی، تبدیل آفین و جابجایی بودند. نتیجه نهایی نشان داد که این تغییرات به درستی اعمال شده‌اند و تصاویر با هم منطبق هستند.

سوال ۲

مقدمه

در این پروژه هدف اعمال چندین تابع پردازش تصویر بر روی یک تصویر ورودی است. این توابع شامل کشش کنتراست، منفی‌سازی، اعمال توان دوم، ریشه دوم و دیگر تبدیل‌ها می‌باشند. برای انجام این کار از کتابخانه‌های OpenCV و NumPy و matplotlib در Python استفاده شده است. در این گزارش به تشریح مراحل و توابع مورد استفاده در این پروژه می‌پردازیم.

توابع پردازش تصویر

۱. تابع T1: منفی‌سازی تصویر

این تابع به منظور منفی‌سازی تصویر استفاده می‌شود، به این معنی که هر پیکسل به مقدار معکوس خود تبدیل می‌شود. در اینجا مقدار هر پیکسل از 255 کم می‌شود تا مقدار معکوس به دست آید.

```python
def T1(image):

    L = 256  # Assuming 8-bit grayscale image

    # Define the transformation function

    def transform_pixel(pixel):

        return -1*pixel + (L-1)

    # Vectorize the transformation function

    vectorized_transform = np.vectorize(transform_pixel)

    # Apply the transformation

    stretched_image = vectorized_transform(image)

    # Clip values to ensure they are within [0, 255] and convert back to uint8

    stretched_image = np.clip(stretched_image, 0, 255).astype(np.uint8)

    return stretched_image
```

شرح:

- در این تابع، ابتدا تصویر به صورت 8 بیتی و خاکستری فرض می‌شود.
- تابع `transform_pixel` مقدار هر پیکسل را معکوس می‌کند.
- سپس با استفاده از `np.vectorize` این تابع به تمام پیکسل‌های تصویر اعمال می‌شود.

- مقادیر پیکسل‌ها به بازه [0, 255] محدود شده و به نوع داده `uint8` تبدیل می‌شوند.

۲. تابع T2: اعمال فیلتر با شرایط مشخص

این تابع تصویر را بر اساس مقادیر مشخصی تغییر می‌دهد. پیکسل‌هایی که در محدوده‌ی مشخصی هستند به صفر تبدیل می‌شوند و بقیه پیکسل‌ها بدون تغییر باقی می‌مانند.

```python
def T2(image, r1, r2):
    L = 255  # Assuming 8-bit grayscale image
    # Define the transformation function
    def transform_pixel(pixel):
        if pixel <= r1*L:
            return pixel
        elif pixel <= r2*L:
            return 0
        else:
            return pixel
    # Vectorize the transformation function
    vectorized_transform = np.vectorize(transform_pixel)
    # Apply the transformation
    stretched_image = vectorized_transform(image)
    # Clip values to ensure they are within [0, 255] and convert back to uint8
    stretched_image = np.clip(stretched_image, 0, 255).astype(np.uint8)
    return stretched_image
```

شرح:

- این تابع دو مقدار `r1` و `r2` را به عنوان ورودی دریافت می‌کند که محدوده‌ای از پیکسل‌ها را مشخص می‌کنند.
- تابع `transform_pixel` مقادیر پیکسل‌هایی که در این محدوده قرار دارند را به صفر تبدیل می‌کند.
- با استفاده از `np.vectorize`، این تابع بر روی تمامی پیکسل‌های تصویر اعمال می‌شود.
- مقادیر نهایی به بازه [0, 255] محدود شده و به نوع `uint8` تبدیل می‌شوند.

۳. تابع T3: اعمال فیلتر با شرایط مشخص

این تابع نیز تصویر را بر اساس مقادیر مشخصی تغییر می‌دهد. پیکسل‌هایی که در محدوده‌ی مشخصی هستند به مقدار ماکسیمم (255) یا 0 تبدیل می‌شوند و بقیه پیکسل‌ها بدون تغییر باقی می‌مانند.

```
def T3(image, r1, r2):

    L = 255  # Assuming 8-bit grayscale image

    # Define the transformation function

    def transform_pixel(pixel):

        if pixel <= r1*L:

            return 0

        elif pixel <= r2*L:

            return pixel

        else:

            return L

    # Vectorize the transformation function

    vectorized_transform = np.vectorize(transform_pixel)

    # Apply the transformation

    stretched_image = vectorized_transform(image)

    # Clip values to ensure they are within [0, 255] and convert back to uint8

    stretched_image = np.clip(stretched_image, 0, 255).astype(np.uint8)

    return stretched_image
```

شرح:

- این تابع نیز مانند T2 دو مقدار `r1` و `r2` را به عنوان ورودی دریافت می‌کند.
- تابع `transform_pixel` پیکسل‌هایی که در محدوده مشخص شده قرار دارند را به مقدار ماکسیمم (255) تبدیل می‌کند.
- با استفاده از `np.vectorize`، این تابع بر روی تمامی پیکسل‌های تصویر اعمال می‌شود.
- مقادیر نهایی به بازه [0, 255] محدود شده و به نوع `uint8` تبدیل می‌شوند.

۴. تابع T4: کشش کنتراست با استفاده از نقاط کلیدی

این تابع کشش کنتراست تصویر را با استفاده از نقاط کلیدی مشخص اعمال می‌کند. این نقاط شامل `r1`, `r2` و `s1` هستند.

```
def T4(image, r1, r2, s1):

    L = 255  # Assuming 8-bit grayscale image

    # Define the transformation function

    def transform_pixel(pixel):

        if pixel <= r1*L:

            return (s1/r1)*pixel

        elif pixel <= r2*L:

            return pixel

        else:

            return ((1 - s1)/(1- r2))*(pixel-r2) + s1

    # Vectorize the transformation function

    vectorized_transform = np.vectorize(transform_pixel)

    # Apply the transformation

    stretched_image = vectorized_transform(image)

    # Clip values to ensure they are within [0, 255] and convert back to uint8
```

```
    stretched_image = np.clip(stretched_image, 0, 255).astype(np.uint8)

    return stretched_image
```

شرح:

- این تابع از سه مقدار `r1`, `r2` و `s1` برای کشش کنتراست استفاده می‌کند.
- تابع `transform_pixel` کشش کنتراست را بر اساس این مقادیر اعمال می‌کند.
- با استفاده از `np.vectorize`، این تابع بر روی تمامی پیکسل‌های تصویر اعمال می‌شود.
- مقادیر نهایی به بازه [0, 255] محدود شده و به نوع `uint8` تبدیل می‌شوند.

۵. تابع T5: اعمال ریشه دوم

این تابع ریشه دوم هر پیکسل را اعمال می‌کند. ابتدا تصویر نرمال شده و سپس ریشه دوم اعمال می‌شود.

```
def T5(image):
    # Normalize the pixel values to the range [0, 1]
    normalized_image = image / 255.0
    # Apply the "2th root" transformation (square root of the pixel values)
    transformed_image = np.sqrt(normalized_image)
    # Scale the transformed image back to the range [0, 255]
    scaled_transformed_image = np.uint8(transformed_image * 255)
    return scaled_transformed_image
```

شرح:

- تصویر به بازه [0, 1] نرمال می‌شود.
- تابع `np.sqrt` ریشه دوم هر پیکسل را محاسبه می‌کند.
- تصویر نهایی به بازه [0, 255] مقیاس‌بندی شده و به نوع `uint8` تبدیل می‌شود.

۶. تابع T6: اعمال توان دوم

این تابع توان دوم هر پیکسل را اعمال می‌کند. ابتدا تصویر نرمال شده و سپس توان دوم اعمال می‌شود.

```
def T6(image):
    # Normalize the pixel values to the range [0, 1]
    normalized_image = image / 255.0
    # Apply the "2th power" transformation (square the pixel values)
    transformed_image = np.power(normalized_image, 2)
    # Scale the transformed image back to the range [0, 255]
    scaled_transformed_image = np.uint8(transformed_image * 255)
    return scaled_transformed_image
```

شرح:

- تصویر به بازه [0, 1] نرمال می‌شود.
- تابع `np.power` توان دوم هر پیکسل را محاسبه می‌کند.
- تصویر نهایی به بازه [0, 255] مقیاس‌بندی شده و به نوع `uint8` تبدیل می‌شود.

بارگذاری و اعمال توابع بر روی تصویر

در نهایت، تصویر اصلی بارگذاری شده و توابع مختلف بر روی آن اعمال می‌شوند. سپس نتایج به همراه تصویر اصلی نمایش داده می‌شوند.

```
image = cv2.imread('image1.jfif', cv2.IMREAD_GRAYSCALE)

images = {
    "Original": image,
    "T1": T1(image),
    "T2": T2(image, 0.2, 0.55),
    "T3": T3(image, 0.4, 0.55),
```

```
    "T4": T4(image, 0.2, 0.55, 0.3),
    "T5": T5(image),
    "T6": T6(image)
}
# Plot all images in one window with the original image on top
plt.figure(figsize=(12, 8))
# Plot the original image
ax = plt.subplot(3, 3, 2)
plt.title("Original")
plt.imshow(images["Original"], cmap='gray')
plt.axis('off')
# Draw a box around the original image
rect = patches.Rectangle((0, 0), images["Original"].shape[1],
images["Original"].shape[0], linewidth=3, edgecolor='red', facecolor='none')
ax.add_patch(rect)
# Plot the transformed images
for i, (title, img) in enumerate(images.items()):
    if title != "Original":
        ax = plt.subplot(3, 3, i + 3)
        plt.title(title)
        plt.imshow(img, cmap='gray')
        plt.axis('off')
        # Draw a box around each transformed image
        rect = patches.Rectangle((0, 0), img.shape[1], img.shape[0],
linewidth=3, edgecolor='red', facecolor='none')
```

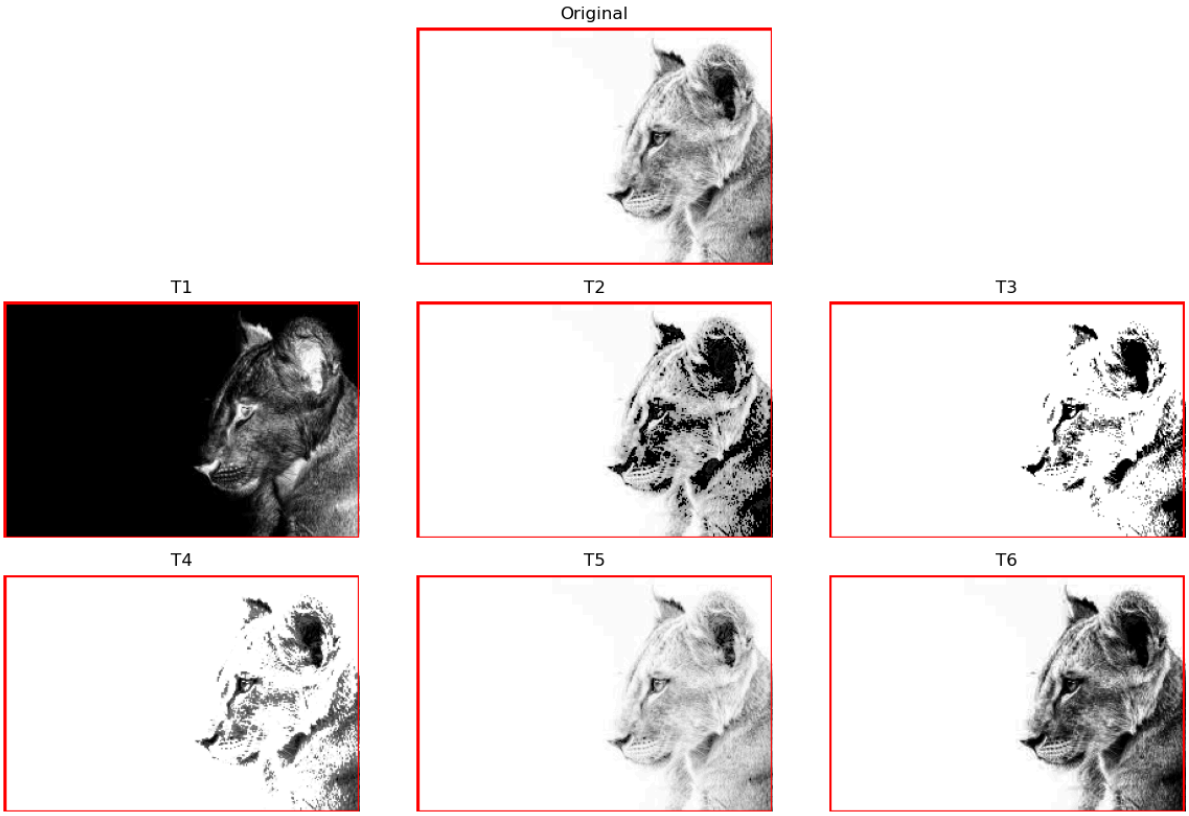
```
        ax.add_patch(rect)

plt.tight_layout()

plt.show()
```

شرح:

- تصویر اصلی با استفاده از OpenCV بارگذاری می‌شود.
- توابع پردازش تصویر مختلف بر روی این تصویر اعمال می‌شوند و نتایج در دیکشنری `images` ذخیره می‌شوند.
- سپس تصاویر اصلی و تغییر یافته با استفاده از matplotlib در یک پنجره نمایش داده می‌شوند.
- برای تاکید بیشتر، یک کادر قرمز دور هر تصویر کشیده می‌شود.

نتایج و بررسی

تبدیل T1 پرواضح است که مکمل 255 را محاسبه میکند. نتیجه آن هم به راحتی میتوان حدس زد. تبدیل T2 پیکسل هایی از تصویر را مشکلی میکند. تبدیل T3 فقط در بخشی از مقدار پیکسل ها خطی است، یعنی تغییری روی تصویر اعمال نمیشود در خارج از این بازه تصویر 0 و 1 میشود. تبدیل T3 در یک بازه مقدار خاصی را به پیکسل ها اختصاص میدهد. در دوبازه خارج از مقدار ثابت پیکسل ها به بازه بزرگتری مپ میشود و باعث میشد کنتراست تصویر در این بازه ها افزایش یابد. تبدیل T5 ریشه دوم پیکسل هارا محاسبه کرده و در تصویر خروجی قرار میدهد. تبدیل T6 توان دوم هر پیکسل را محاسبه کرده و به خروجی انتساب میدهد.



نتیجه‌گیری

در این پروژه، با استفاده از روش‌های مختلف پردازش تصویر، چندین تغییر بر روی یک تصویر اصلی اعمال و سپس نتایج حاصل مقایسه شدند. این تغییرات شامل منفی‌سازی، کشش کنتراست، فیلترهای مختلف و اعمال توان و ریشه بودند. نتیجه نهایی نشان داد که این تغییرات به درستی اعمال شده‌اند و تصاویر تغییر یافته با تصویر اصلی تفاوت‌های قابل توجهی دارند. این پروژه به خوبی نشان می‌دهد که چگونه می‌توان با استفاده از ابزارهای ساده پردازش تصویر، تغییرات مختلفی را بر روی تصاویر اعمال کرد و نتایج جالب و مفیدی به دست آورد.

سوال ۳

مقدمه

در این پروژه، دو تصویر را با استفاده از تکنیک‌های هیستوگرام مورد تحلیل قرار می‌دهیم. ابتدا تصویر `trees.jpeg` را می‌خوانیم و کانال‌های RGB آن را جدا کرده و نمایش می‌دهیم و هیستوگرام هر کانال و هیستوگرام کلی تصویر رنگی را ترسیم می‌کنیم. سپس تصویر دیگری به نام `abraham.jpg` را می‌خوانیم، هیستوگرام آن را محاسبه می‌کنیم، روش هیستوگرام اکولایزیشن را بر روی آن اعمال می‌کنیم و تغییرات تصویر و هیستوگرام را مشاهده می‌کنیم.

بخش اول: تحلیل تصویر `trees.jpeg`

گام ۱: خواندن تصویر

ابتدا تصویر `trees.jpeg` را با استفاده از OpenCV می‌خوانیم. OpenCV به صورت پیش‌فرض تصاویر را در فرمت BGR می‌خواند.

```
import cv2

import numpy as np

import matplotlib.pyplot as plt

# Step 1: Read the image

image = cv2.imread('trees.jpeg')
```

گام ۲: تبدیل BGR به RGB

از آنجا که Matplotlib تصاویر را در فرمت RGB نمایش می‌دهد، نیاز به تبدیل تصویر از BGR به RGB داریم.

```
# Step 2: Convert image from BGR (OpenCV format) to RGB (Matplotlib format)

image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

گام ۳: نمایش تصویر اصلی

تصویر اصلی را با استفاده از Matplotlib نمایش می‌دهیم.

```
# Step 3: Display the original image

plt.figure(figsize=(10, 7))

plt.subplot(2, 2, 1)

plt.imshow(image_rgb)

plt.title('Original Image')

plt.axis('off')
```

گام ۴: جدا کردن کانال‌های رنگ

کانال‌های RGB تصویر را جدا می‌کنیم.

```
# Step 4: Separate the color channels

R, G, B = image_rgb[:,:,0], image_rgb[:,:,1], image_rgb[:,:,2]
```

گام ۵: نمایش هر کانال رنگ

هر کانال رنگ را به صورت جداگانه نمایش می‌دهیم.

```
# Step 5: Display each color channel

plt.subplot(2, 2, 2)

plt.imshow(R, cmap='Reds')

plt.title('Red Channel')

plt.axis('off')

plt.subplot(2, 2, 3)

plt.imshow(G, cmap='Greens')

plt.title('Green Channel')

plt.axis('off')

plt.subplot(2, 2, 4)

plt.imshow(B, cmap='Blues')

plt.title('Blue Channel')

plt.axis('off')

plt.tight_layout()

plt.show()
```

گام ۶: محاسبه هیستوگرام‌ها

هیستوگرام‌های هر کانال رنگ و تصویر کلی را محاسبه می‌کنیم.

```
# Step 6: Compute histograms

hist_R = cv2.calcHist([R], [0], None, [256], [0, 256])

hist_G = cv2.calcHist([G], [0], None, [256], [0, 256])

hist_B = cv2.calcHist([B], [0], None, [256], [0, 256])

hist_total = cv2.calcHist([image_rgb], [0, 1, 2], None, [256, 256, 256], [0,
256, 0, 256, 0, 256])
```

گام ۷: ترسیم هیستوگرام‌ها

هیستوگرام‌های هر کانال و هیستوگرام کلی را ترسیم می‌کنیم.

```
# Step 7: Plot histograms

plt.figure(figsize=(15, 5))

plt.subplot(1, 4, 1)

plt.plot(hist_R, color='r')

plt.title('Red Channel Histogram')

plt.xlim([0, 256])

plt.subplot(1, 4, 2)

plt.plot(hist_G, color='g')

plt.title('Green Channel Histogram')

plt.xlim([0, 256])

plt.subplot(1, 4, 3)

plt.plot(hist_B, color='b')

plt.title('Blue Channel Histogram')

plt.xlim([0, 256])

plt.subplot(1, 4, 4)
```

```
plt.hist(image_rgb.ravel(), bins=256, color='black', alpha=0.5,
label='Overall')

plt.title('Overall Histogram')

plt.xlim([0, 256])

plt.tight_layout()

plt.show()
```

بخش دوم: تحلیل تصویر `abraham.jpg`

گام ۱: خواندن تصویر

تصویر `abraham.jpg` را در حالت خاکستری (grayscale) می‌خوانیم.

```
# Step 1: Read the image

image = cv2.imread('abraham.jpg', cv2.IMREAD_GRAYSCALE)
```

گام ۲: نمایش تصویر اصلی

تصویر اصلی خاکستری را نمایش می‌دهیم.

```
# Step 2: Display the original image

plt.figure(figsize=(10, 7))

plt.subplot(2, 2, 1)

plt.imshow(image, cmap='gray')

plt.title('Original Image')

plt.axis('off')
```

گام ۳: ترسیم هیستوگرام تصویر اصلی

هیستوگرام تصویر اصلی را محاسبه و ترسیم می‌کنیم.

```
# Step 3: Draw histogram of the original image
hist_orig = cv2.calcHist([image], [0], None, [256], [0, 256])
plt.subplot(2, 2, 2)
plt.plot(hist_orig, color='black')
plt.title('Histogram of Original Image')
plt.xlim([0, 256])
```

گام ۴: اعمال هیستوگرام اکولایزیشن

روش هیستوگرام اکولایزیشن را برای افزایش کنتراست تصویر اعمال می‌کنیم.

```
# Step 4: Apply Histogram Equalization
equalized_image = cv2.equalizeHist(image)
```

گام ۵: نمایش تصویر اکولایزشده

تصویر اکولایزشده را نمایش می‌دهیم.

```
# Step 5: Display the equalized image
plt.subplot(2, 2, 3)
plt.imshow(equalized_image, cmap='gray')
plt.title('Equalized Image')
plt.axis('off')
```

گام ۶: ترسیم هیستوگرام تصویر اکولایزشده

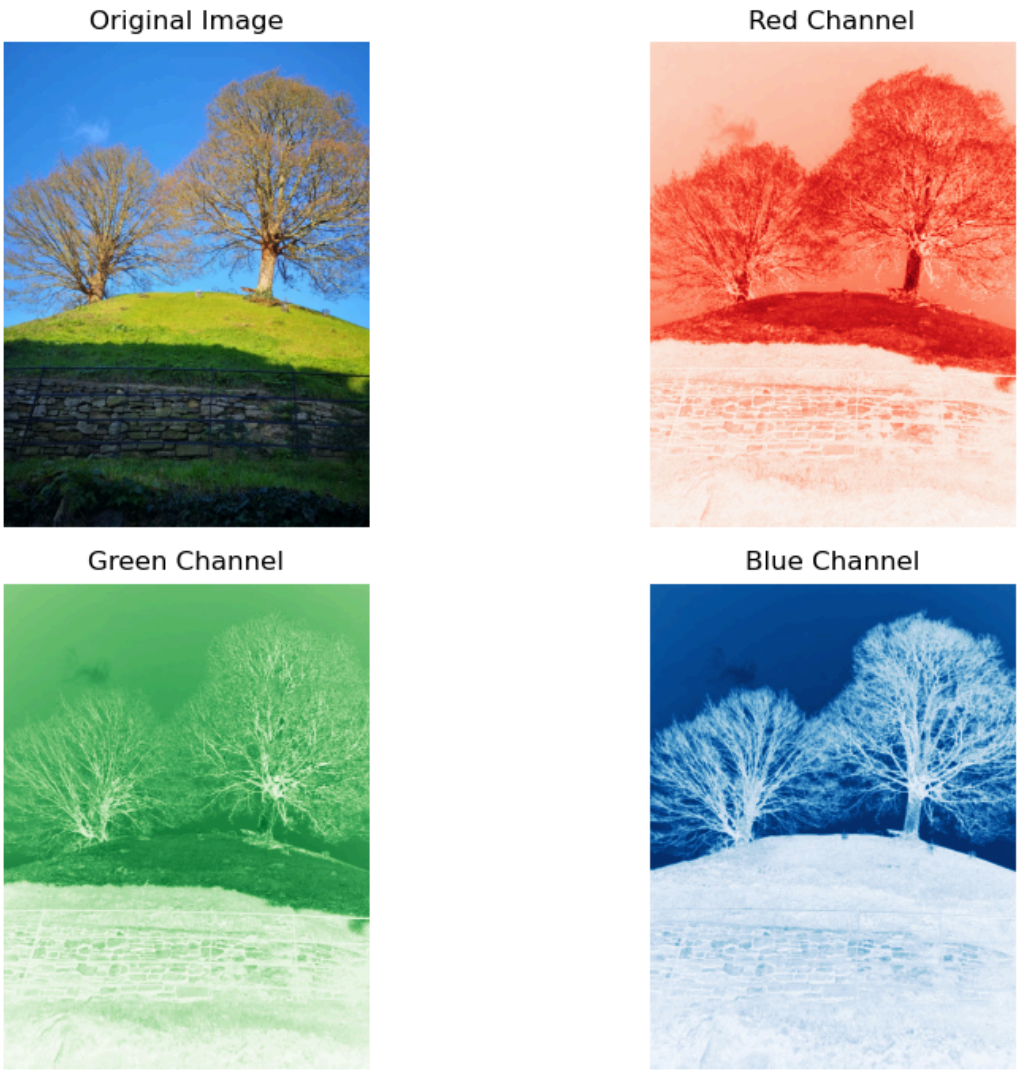
هیستوگرام تصویر اکولایزشده را محاسبه و ترسیم می‌کنیم.

```
# Step 6: Draw histogram of the equalized image
hist_equalized = cv2.calcHist([equalized_image], [0], None, [256], [0, 256])
```

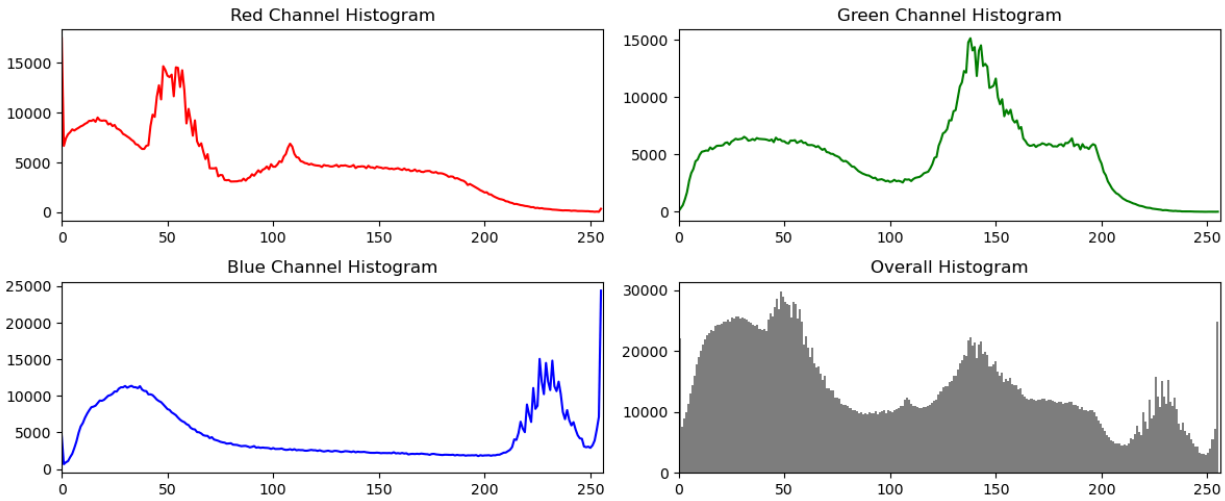
```
plt.subplot(2, 2, 4)
plt.plot(hist_equalized, color='black')
plt.title('Histogram of Equalized Image')
plt.xlim([0, 256])

plt.tight_layout()
plt.show()
```

نتیجه بخش اول

تصویر اصلی و تصاویر تک کاناله به صورت زیر هستند. تصاویر تک کاناله فقط intensity یک کانال را نشان میدهد.

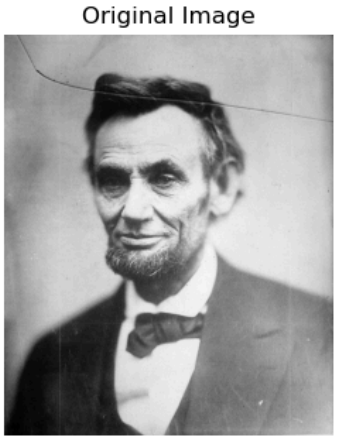


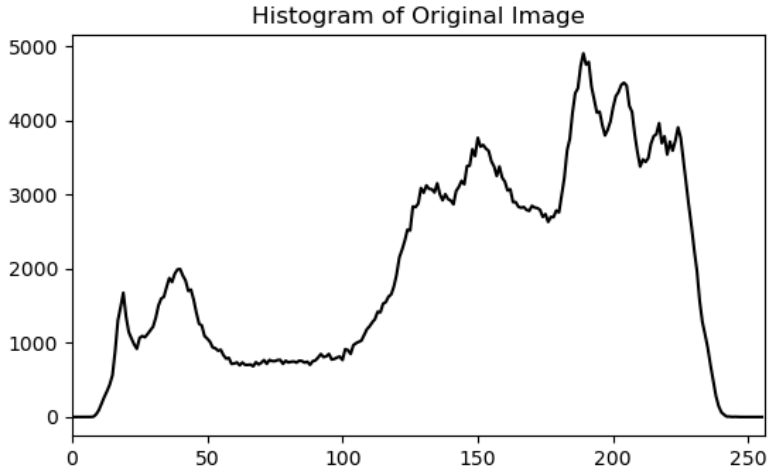
هیستوگرام کانال‌ها و هستوگرام کلی. هستوگرام های کانال‌ها براساس intensity هر کانال بدست آمده است. اما هستوگرام کلی جمع تمام هستوگرام ها است.

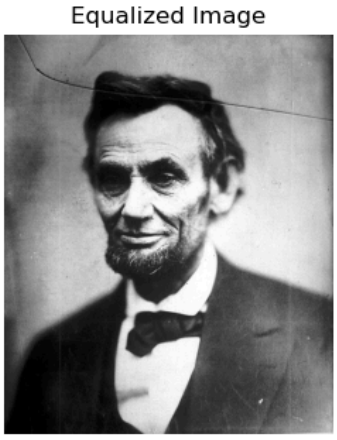


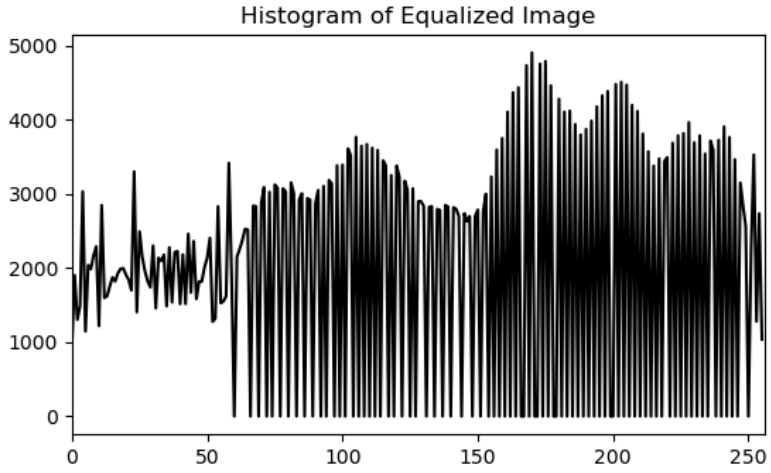
نتیجه بخش دوم

نتایج و تصاویر خواسته شده به صورت زیر است. برای پی بردن به اثر equalization باید نمودار CDF را رسم کنیم.

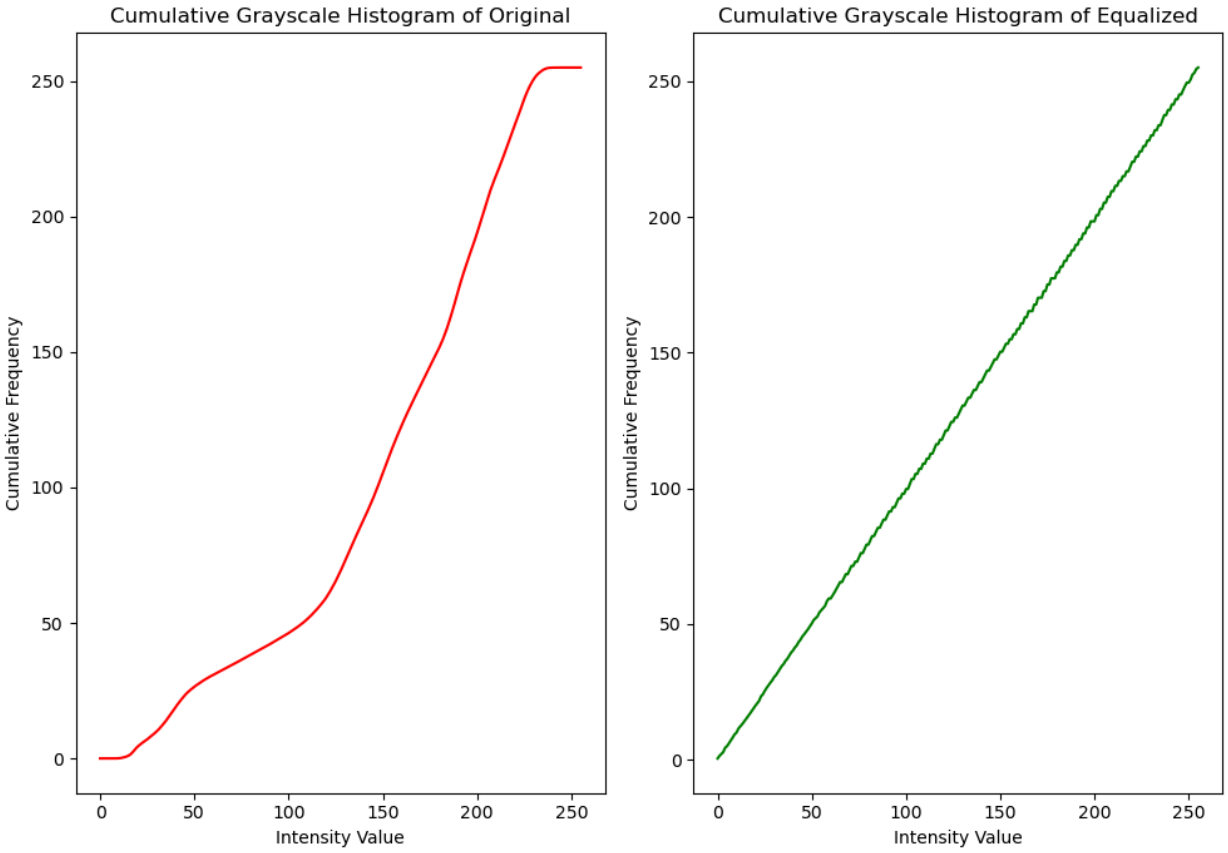








نمودار CDF که به خوبی تاثیر equalization مشاهده میشود. باعث افزایش کنتراست تصویر شده.



تحلیل و نتیجه‌گیری

پس از اعمال هیستوگرام اکولایزیشن، کنتراست تصویر بهبود یافته و هیستوگرام تصویر از حالت فشرده به حالت گسترده‌تری تبدیل شده است. این تغییرات نشان‌دهنده توزیع یکنواخت‌تر سطوح خاکستری در تصویر اکولایزشده می‌باشد. این بهبود کنتراست، جزئیات بیشتری از تصویر را نمایان می‌سازد و تصویر بهتری برای تحلیل و پردازش‌های بعدی فراهم می‌آورد.

سوال ۴

مقدمه

در این پروژه، هدف ما بررسی تاثیر فیلترهای مختلف نرم‌سازی و تشخیص لبه بر روی یک تصویر هوایی می‌باشد. ابتدا تصویر `AerialView.jpeg` را می‌خوانیم و سپس فیلترهای گوسین، میانه و شارپنینگ را بر روی آن اعمال می‌کنیم. پس از آن، الگوریتم‌های تشخیص لبه Sobel و Canny را روی تصویر اصلی و تصاویر فیلترشده اعمال می‌کنیم تا بهترین ترکیب فیلتر نرم‌سازی و الگوریتم تشخیص لبه را بر اساس معیارهای مختلف مقایسه کنیم.

بخش اول: اعمال فیلترهای مختلف

گام ۱: خواندن تصویر

ابتدا تصویر `AerialView.jpeg` را به صورت خاکستری (grayscale) می‌خوانیم.

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

# Step 1: Read the image
image = cv2.imread('AerialView.jpeg', cv2.IMREAD_GRAYSCALE)
```

گام ۲: اعمال فیلتر گوسین

فیلتر گوسین با کرنل 5x5 بر روی تصویر اعمال می‌شود تا نویزها کاهش یابند.

```
# Step 2: Apply Gaussian Filter
gaussian_blur = cv2.GaussianBlur(image, (5, 5), 0)
```

گام ۳: اعمال فیلتر میانه

فیلتر میانه با اندازه کرنل 5 بر روی تصویر اعمال می‌شود تا نویزهای نمکی و فلفلی کاهش یابند.

```
# Step 3: Apply Median Filter
median_blur = cv2.medianBlur(image, 5)
```

گام ۴: اعمال فیلتر شارپنینگ

فیلتر شارپنینگ با استفاده از یک کرنل مشخص برای افزایش وضوح تصویر اعمال می‌شود.

```
# Step 4: Apply Sharpening Filter
sharpening_kernel = np.array([[-1, -1, -1],
                              [-1, 9, -1],
                              [-1, -1, -1]])
sharpened = cv2.filter2D(image, -1, sharpening_kernel)
```

بخش دوم: اعمال الگوریتم‌های تشخیص لبه

گام ۵: تشخیص لبه Sobel

الگوریتم Sobel را برای تشخیص لبه‌های افقی و عمودی و ترکیب آن‌ها اعمال می‌کنیم.

```
# Step 5: Apply Sobel Edge Detection
```

```
sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)
sobel_combined = cv2.magnitude(sobel_x, sobel_y)
```

گام ۶: تشخیص لبه Canny

الگوریتم Canny را برای تشخیص لبه‌های تصویر اعمال می‌کنیم.

```
# Step 6: Apply Canny Edge Detection
canny_edges = cv2.Canny(image, 100, 200)
```

گام ۷: اعمال Sobel و Canny بر روی تصویر فیلتر شده با گوسین

الگوریتم‌های Sobel و Canny را بر روی تصویر گوسین بلور اعمال می‌کنیم.

```
# Step 7: Apply Sobel and Canny on Gaussian Blurred Image
sobel_gaussian = cv2.magnitude(cv2.Sobel(gaussian_blur, cv2.CV_64F, 1, 0,
ksize=3), cv2.Sobel(gaussian_blur, cv2.CV_64F, 0, 1, ksize=3))
canny_gaussian = cv2.Canny(gaussian_blur, 100, 200)
```

گام ۸: اعمال Sobel و Canny بر روی تصویر فیلتر شده با میانه

الگوریتم‌های Sobel و Canny را بر روی تصویر میانه بلور اعمال می‌کنیم.

```
# Step 8: Apply Sobel and Canny on Median Blurred Image
sobel_median = cv2.magnitude(cv2.Sobel(median_blur, cv2.CV_64F, 1, 0, ksize=3),
cv2.Sobel(median_blur, cv2.CV_64F, 0, 1, ksize=3))
canny_median = cv2.Canny(median_blur, 100, 200)
```

گام ۹: اعمال Sobel و Canny بر روی تصویر فیلتر شده با شارپنینگ

الگوریتم‌های Sobel و Canny را بر روی تصویر شارپن‌شده اعمال می‌کنیم.

```
# Step 9: Apply Sobel and Canny on Sharpened Image
sobel_sharpened = cv2.magnitude(cv2.Sobel(sharpened, cv2.CV_64F, 1, 0,
ksize=3), cv2.Sobel(sharpened, cv2.CV_64F, 0, 1, ksize=3))
canny_sharpened = cv2.Canny(sharpened, 100, 200)
```

نمایش نتایج

تصاویر فیلتر شده و نتایج الگوریتم‌های تشخیص لبه را نمایش می‌دهیم.

```
# Display the results
plt.figure(figsize=(20, 15))

plt.subplot(3, 4, 1)
plt.imshow(gaussian_blur, cmap='gray')
plt.title('Gaussian Blurred')
plt.axis('off')

plt.subplot(3, 4, 2)
plt.imshow(median_blur, cmap='gray')
plt.title('Median Blurred')
plt.axis('off')

plt.subplot(3, 4, 3)
plt.imshow(sharpened, cmap='gray')
plt.title('Sharpened')
plt.axis('off')

plt.subplot(3, 4, 4)
plt.imshow(image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(3, 4, 5)
plt.imshow(sobel_gaussian, cmap='gray')
plt.title('Sobel on Gaussian')
plt.axis('off')

plt.subplot(3, 4, 6)
plt.imshow(canny_gaussian, cmap='gray')
plt.title('Canny on Gaussian')
plt.axis('off')

plt.subplot(3, 4, 7)
plt.imshow(sobel_median, cmap='gray')
plt.title('Sobel on Median')
plt.axis('off')

plt.subplot(3, 4, 8)
plt.imshow(canny_median, cmap='gray')
```

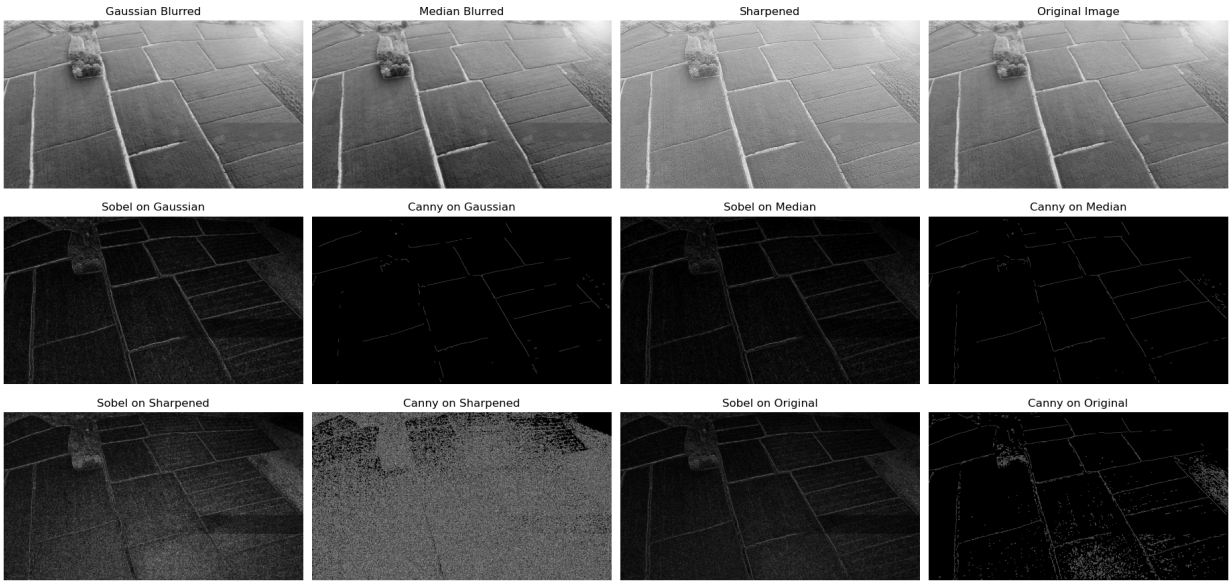
```
plt.title('Canny on Median')
plt.axis('off')

plt.subplot(3, 4, 9)
plt.imshow(sobel_sharpened, cmap='gray')
plt.title('Sobel on Sharpened')
plt.axis('off')

plt.subplot(3, 4, 10)
plt.imshow(canny_sharpened, cmap='gray')
plt.title('Canny on Sharpened')
plt.axis('off')

plt.subplot(3, 4, 11)
plt.imshow(sobel_combined, cmap='gray')
plt.title('Sobel on Original')
plt.axis('off')

plt.subplot(3, 4, 12)
plt.imshow(canny_edges, cmap='gray')
plt.title('Canny on Original')
plt.axis('off')

plt.tight_layout()
plt.show()
```

تحلیل و نتیجه‌گیری

برای مقایسه نتایج الگوریتم‌های تشخیص لبه Sobel و Canny با استفاده از تصاویر فیلتر شده مختلف، معیارهای زیر را در نظر می‌گیریم:

1. **ضخامت لبه‌های تشخیص داده شده:**
   - الگوریتم Canny به طور کلی لبه‌های نازک‌تری نسبت به Sobel تولید می‌کند.
   - لبه‌های تولید شده توسط Sobel معمولاً ضخیم‌تر هستند که می‌تواند به تشخیص بهتر ساختارهای بزرگتر کمک کند.
2. **عملکرد الگوریتم در تشخیص لبه‌های افقی و عمودی:**
   - الگوریتم Sobel به خوبی قادر به تشخیص لبه‌های افقی و عمودی به صورت جداگانه است.
   - الگوریتم Canny نیز در تشخیص لبه‌های افقی و عمودی عملکرد خوبی دارد اما با دقت بیشتری در لبه‌های نازک‌تر.
3. **دقت لبه‌های تشخیص داده شده:**

- الگوریتم Canny به دلیل استفاده از روش‌های چندمرحله‌ای، دقت بالاتری در تشخیص لبه‌های واقعی تصویر دارد.
- الگوریتم Sobel ممکن است لبه‌های کاذب بیشتری تولید کند، اما برای تصاویر با کنتراست پایین می‌تواند مفید باشد.



نتیجه نهایی

برای تصویر `AerialView.jpeg`:

- **بهترین فیلتر نرم‌سازی**: فیلتر گوسین به دلیل کاهش نویز و حفظ جزئیات تصویر بهتر عمل می‌کند.
- **بهترین الگوریتم تشخیص لبه**: الگوریتم Canny به دلیل دقت بالاتر و تولید لبه‌های نازک‌تر، بهترین عملکرد را در تشخیص لبه‌های تصویر دارد.

این نتایج می‌توانند بسته به نوع تصویر و نیازهای خاص پروژه متغیر باشند، اما در این آزمایش خاص، ترکیب فیلتر گوسین و الگوریتم Canny بهترین نتایج را ارائه داده است.