AGH
University of Science and Technology in Krakow

DEPARTMENT OF MINING GEODESY AND ENVIRONMENTAL ENGINEERING

Theme: "Using JupyterNotebook for image processing"
Author: Tymoteusz Maj
Field of study: Remote Sensing and GIS

Kraków, 2024

Technical Report: Raster Image Representation in Python

1. Introduction

Raster data is a fundamental representation in geospatial analysis, where spatial information is stored as a grid of squares (pixels). Each pixel contains a numeric value corresponding to a specific attribute, such as brightness, color, or intensity, depending on the image type. These numeric values enable various computational manipulations and visualizations of the image. This report explores how to create and manipulate raster images using NumPy arrays and visualize them using Python libraries such as Matplotlib and Pillow (PIL).

2. Understanding Raster Data

A raster image can be viewed as a 2D array, where each element corresponds to a pixel. Each pixel holds a numeric value that determines its brightness or intensity. For standard 8-bit grayscale images, the range of pixel values is from 0 to 255:

- **0:** Represents black, the darkest shade.
- **255:** Represents white, the brightest shade.
- **Intermediate values** represent varying levels of gray between black and white.

This model of an image can be directly represented using NumPy arrays, which are highly efficient for mathematical and matrix operations, making them an excellent tool for image processing tasks.

2.1 Libraries Import

We begin by importing the necessary libraries for image generation and manipulation:

```python
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from ipywidgets import interact
import requests
from io import BytesIO
from google.colab import drive
```

3. Image Generation

3.1 Creating a Random Array

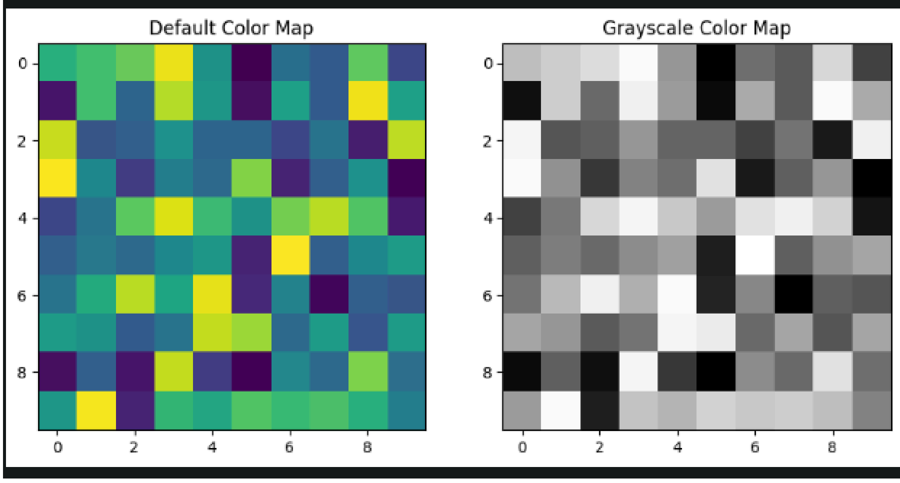
We generate a 10x10 random array with pixel values ranging from 0 to 255, simulating an 8-bit grayscale image:

```
 # Set a seed for reproducibility
 np.random.seed(24)

 # Generate a 10x10 random array with values between 0 and 255 (8-bit image)
 ar = np.random.randint(0, 255, (10, 10), dtype=np.uint8)

In [ ]:

 # Display array properties for reference
 print(ar)
 print(f'\nArray Shape: {ar.shape}\nArray Data Type: {ar.dtype}')

[[162 177 195 245 131   3  96  75 192  56]
 [ 19 179  87 225 135  13 145  77 247 145]
 [232  70  81 129  84  86  56 100  25 228]
 [251 123  48 110  92 207  28  79 130   4]
 [ 56 102 189 239 175 132 201 227 184  23]
 [ 82 107  91 121 139  29 252  80 123 143]
 [100 158 227 149 244  34 115   7  78  70]
 [143 131  76  98 231 216  89 143  73 141]
 [ 13  82  18 231  50   5 119  91 205  93]
 [135 249  31 170 153 185 171 181 163 110]]

Array Shape: (10, 10)
Array Data Type: uint8
```

3.2 Displaying the Array as an Image

We visualize the generated array using Matplotlib's plt.imshow() function, which allows for dynamic adjustments and color mapping:

```
# Display the array as an image using matplotlib's default color scheme and grayscale
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(ar)  # Default colors
plt.title("Default Color Map")
plt.subplot(1, 2, 2)
plt.imshow(ar, cmap=plt.cm.Greys_r)  # Grayscale color map
plt.title("Grayscale Color Map")
plt.show()
```



3.3 Displaying the Histogram

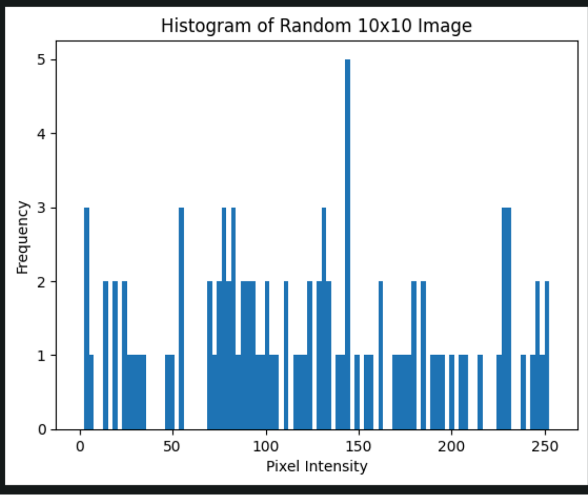
We plot a histogram of pixel values using plt.hist(), which provides insight into the pixel intensity distribution:

Displaying histogram of the image

- What is the histogram of the image?
- `plt.hist()` function hist()

In [ ]:

```python
# Plot histogram of pixel values
plt.hist(ar.ravel(), bins=100, range=(0, 255))
plt.title("Histogram of Random 10x10 Image")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.show()
```



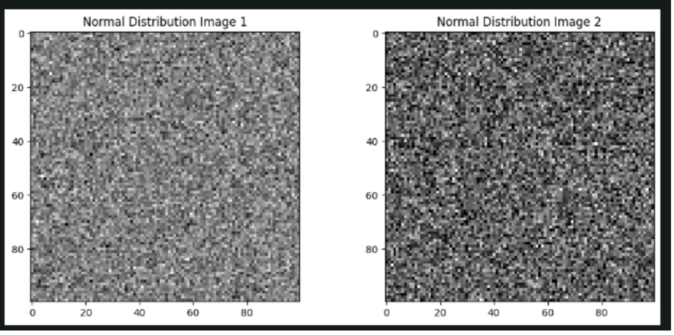
4. Experimenting with Normal Distribution

We expand our analysis by generating larger arrays with normal distributions (100x100 and 500x500) and visualizing their histograms:

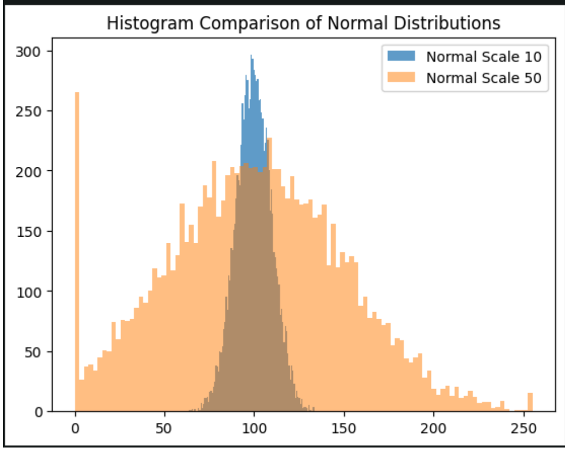
```python
# Generate larger arrays with normal distribution and display them
ar_normal = np.random.normal(loc=100, scale=10, size=(100, 100))
ar_normal = np.clip(ar_normal, 0, 255)  # Clip values between 0 and 255 for valid pix

ar_normal_150 = np.random.normal(loc=100, scale=50, size=(100, 100))
ar_normal_150 = np.clip(ar_normal_150, 0, 255)
```

In [ ]:

```python
# Display images and histograms for normal distributions
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.imshow(ar_normal, cmap='gray')
plt.title("Normal Distribution Image 1")
plt.subplot(1, 2, 2)
plt.imshow(ar_normal_150, cmap='gray')
plt.title("Normal Distribution Image 2")
plt.show()
```



```python
# Display histograms for both normal distributions
plt.hist(ar_normal.flatten(), bins=100, alpha=0.7, label='Normal Scale 10')
plt.hist(ar_normal_150.flatten(), bins=100, alpha=0.5, label='Normal Scale 50')
plt.title("Histogram Comparison of Normal Distributions")
plt.legend()
plt.show()
```



5. Creating Images with Varying Resolutions

We create random images of different resolutions (20x20, 100x100, and 500x500 pixels) and visualize them:

Playing with spatial resolution

Creating 3 single-channel images with different shape:

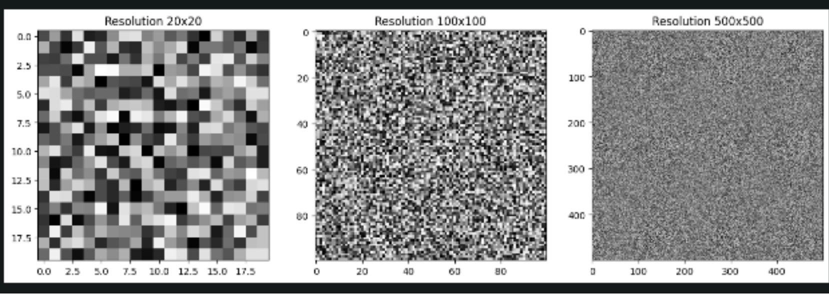
- 20 x 20 pixels
- 50 x 50 pixels
- 500 x 500 pixels

In [ ]:

```python
# Experiment with images of varying resolutions
r20 = np.random.randint(0, 255, (20, 20), dtype=np.uint8)
r100 = np.random.randint(0, 255, (100, 100), dtype=np.uint8)
r500 = np.random.randint(0, 255, (500, 500), dtype=np.uint8)
```

In [ ]:

```python
# Titles for the images with different resolutions
titles = ['Resolution 20x20', 'Resolution 100x100', 'Resolution 500x500']

# Display the images with different resolutions
plt.figure(figsize=(15, 5))
for i, img in enumerate([r20, r100, r500], 1):
    plt.subplot(1, 3, i)
    plt.imshow(img, cmap='gray')
    plt.title(titles[i - 1])
plt.show()
```



6. Image Transformation and Manipulation

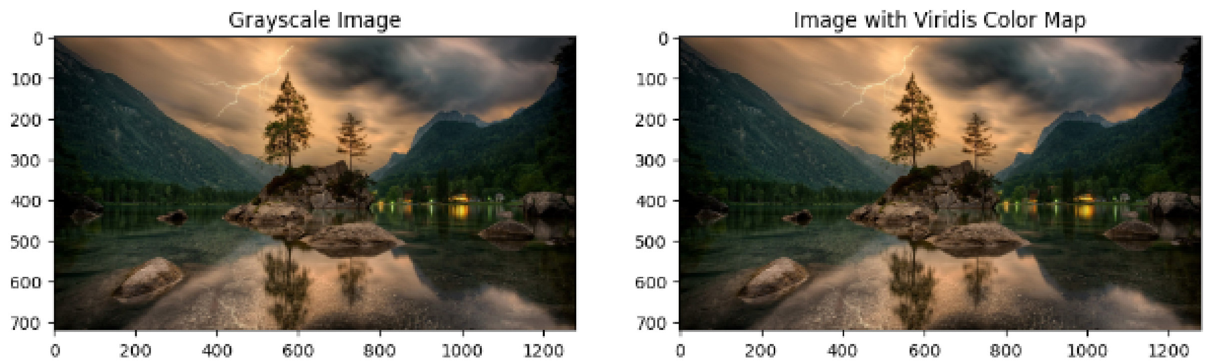
6.1 Loading an Image from the Internet

We load an image from a given URL and convert it into a NumPy array for further manipulation:

```
In [58]:

# # Load an Image from the Internet
image_url = 'https://cdn.pixabay.com/photo/2018/01/14/23/12/nature-3082832_1280.jpg'
response = requests.get(image_url)  # Send a GET request to the URL

# # Check if the request was successful
if response.status_code == 200:
    try:
        image = Image.open(BytesIO(response.content))  # Open the image from the resp
    except UnidentifiedImageError:
        print("Error: The image could not be identified. Please check the URL or the
else:
    print(f"Error: Unable to retrieve image. Status code: {response.status_code}")
```

6.2 Visualizing the Loaded Image

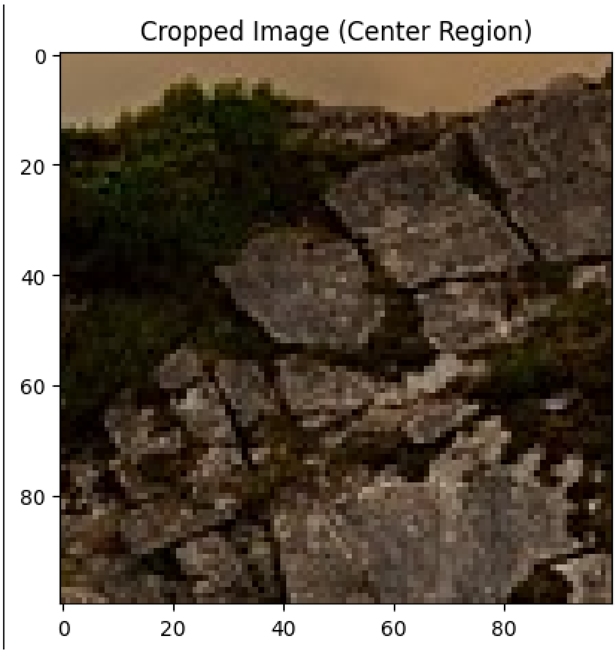
We visualize the loaded image in different color maps:



6.3 Cropping the Image

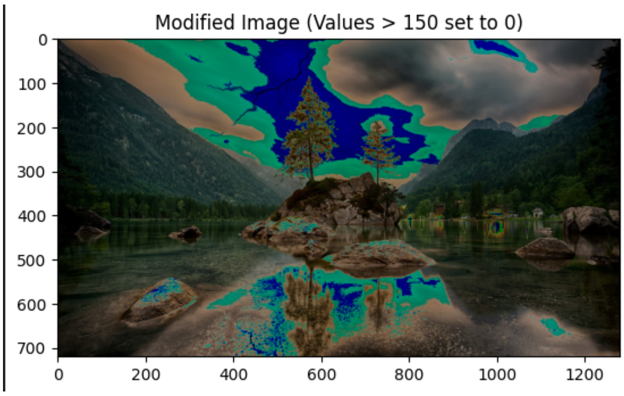
We crop a specific region from the loaded image, focusing on a central square area:

```
# Crop the center square
cropped_img = img_array[center_y - half_size:center_y + half_size, center_x - hal
plt.imshow(cropped_img, cmap='gray')
plt.title('Cropped Image (Center Region)')
plt.show()
```



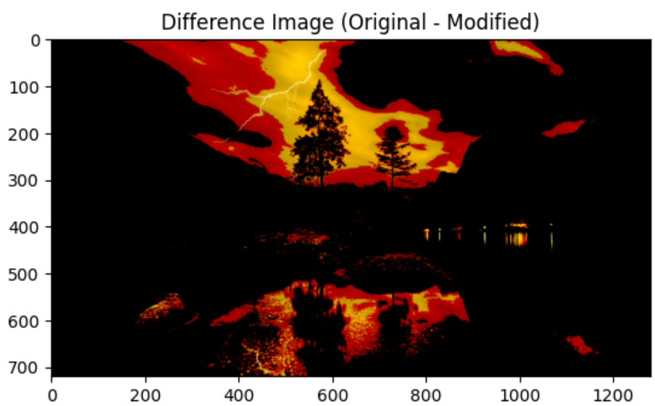
6.4 Modifying the Image

We set pixel values greater than 150 to 0, highlighting areas of interest:



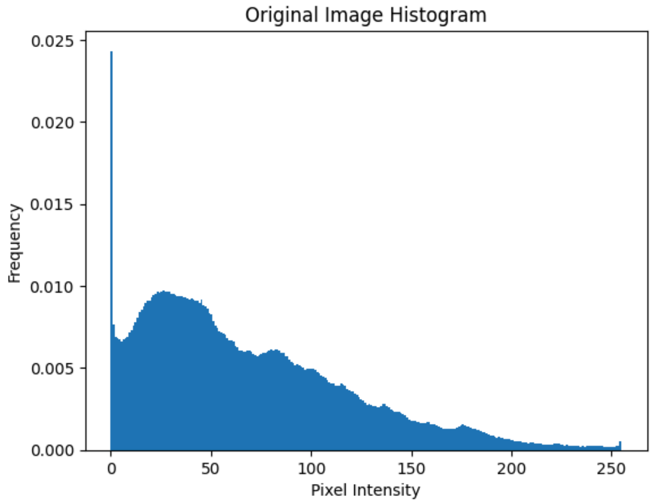
6.5 Calculating the Difference Image

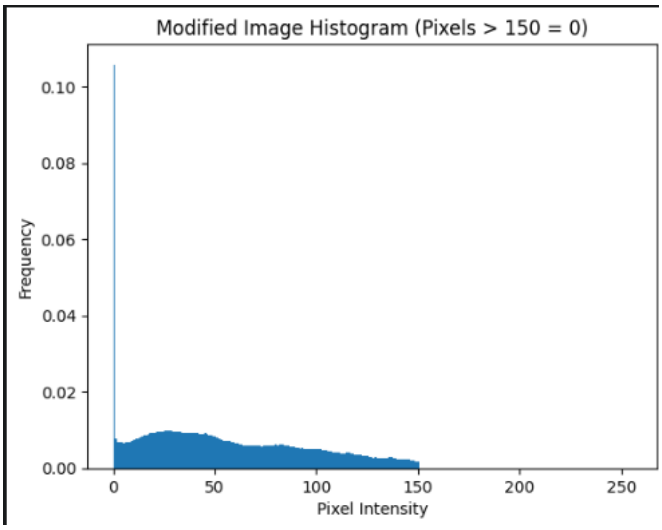
We calculate the difference between the original and modified images to highlight the changes:

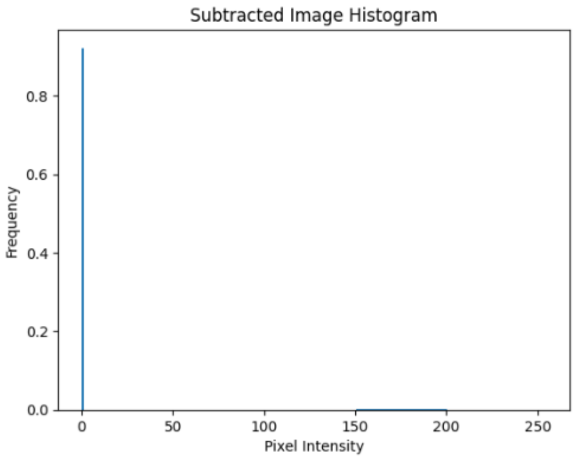


6.6 Histogram Analysis

We plot histograms for different image states to analyze pixel distributions:

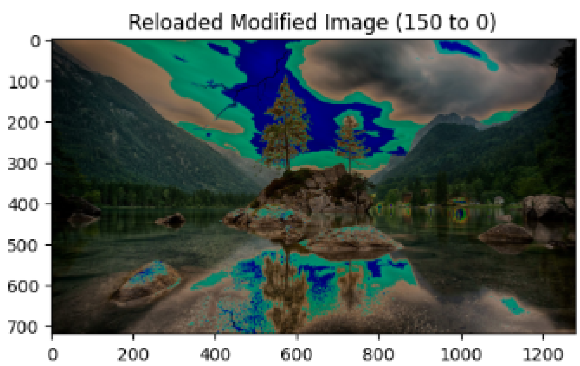


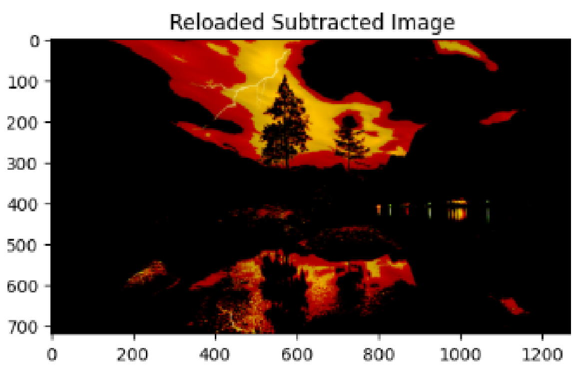




6.7 Saving and Reloading Images

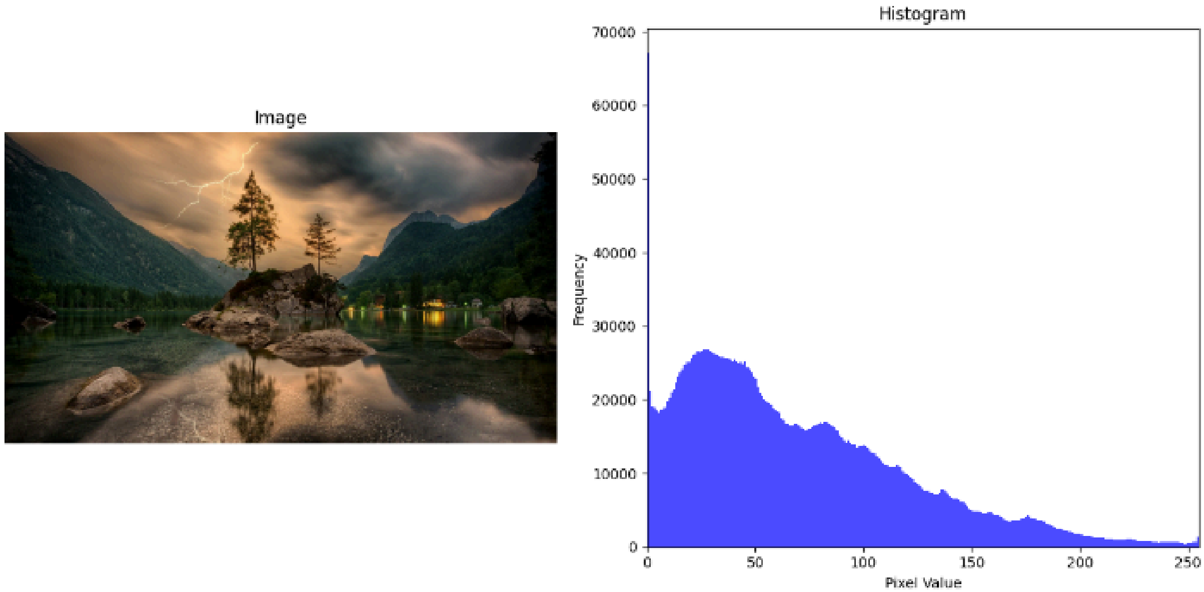
We save the modified and difference images and then reload them to verify integrity:





6.8 Visualizing Image Alongside Its Histogram

Finally, we display the modified image and its histogram side by side for comprehensive analysis:



7. Conclusion

The implementation of raster image representation and manipulation in Python showcases the power and flexibility of using libraries like NumPy, Matplotlib, and Pillow. This approach allows for efficient image processing techniques, including random generation, image loading, manipulation, and visualization, making it suitable for a wide range of applications in geospatial analysis and beyond. The ability to visualize and analyze pixel intensity distributions through histograms further enhances the understanding of image characteristics, aiding in informed decision-making during image analysis tasks.