Introduction to Image Analysis

Compulsory Assignment 1

Ugnius Alekna

October 2023

1 Introduction

The assignment concerns the practical exercises we've covered in the first five weeks of the course. This project serves as a summary of the course material we've learned so far and involves the hands-on implementation of these topics. As a Mathematics student, I have had an option to choose between Python and C++ as the programming language for this assignment. I have decided to use Python and this choice will be consistent for Compulsory Assignments 2 and 3.

This serves as a written report of Compulsory Assignment 1 that provides a brief description of each topic covered, details on how I implemented them, and a discussion of the choices I made during the implementation. Both the report and the implemented code will be submitted. I will also explain how to run each example on the command line in an Ubuntu Linux environment.

The data used for this assignment is available in a zip file provided under "Files" in the "General" channel in the MIF-IA Teams Channel.

2 Assignment tasks

2.1 Image Loading

Purpose.

Acquire a practical understanding of different image formats, their memory structure, and the techniques for combining/ manipulating them. Visualizing the images represented in either RGB or grayscale, using 8 bits per channel always.

Image Investigation.

`'tiffinfo'` is a command line tool used to extract and display detailed information about TIFF (Tagged Image File Format) images. It provides insights into the image's technical specifications and metadata.

In order to use `'tiffinfo'` tool, `'libtiff-tools'` package must be installed on your machine. Using the `'tiffinfo'` tool to investigate an image is a straightforward process. It can be achieved by opening terminal window, navigating to the directory of an image, and running the following commands:

```
> cd path_to_data\data
> tiffinfo example_image.tif
```

Let's use this tool for an image `'Kidney1.tif'`.

```
> C:\Users\path_to_data\data> tiffinfo Kidney1.tif
TIFF Directory at offset 0xd3808 (866312)
Subfile Type: (0 = 0x0)
```

```
Image Width: 1120 Image Length: 2508 Image Depth: 1
Tile Width: 240 Tile Length: 240
Bits/Sample: 8
Compression Scheme: JPEG
Photometric Interpretation: RGB color
YCbCr Subsampling: 2, 2
Samples/Pixel: 3
Planar Configuration: single image plane
ImageDescription: Aperio Image Library v12.0.15
18288x40236 [0,100 17927x40136] (240x240) -> 1120x2508 JPEG/RGB Q=92
JPEG Tables: (574 bytes)
```

Most of the information obtained, such as image dimensions, `Bits/Sample`, `Compression Scheme`, `Photometric Interpretation`, etc. is self-explanatory. A few more interesting observations could include:

- `Tile Width, Tile Length` shows that the selected tif image is stored in memory as a collection of smaller, fixed-size blocks (tiles).
- `YCbCr Subsampling: 2, 2` indicates that color information is averaged within 2x2 blocks of pixels for image size reduction.
- `Planar Configuration` means that the image is stored in a single-plane configuration, i.e. all channels are stored together; sometimes referred to as *chunky*.
- `18288x40236 [0,100 17927x40136] (240x240) -> 1120x2508 JPEG/RGB Q=92`. This line describes the transformation process applied to the image. `18288x40236` is the original resolution of an image; `[0,100 17927x40136]` is a cropping of an image, where `[0,100]` are starting pixel coordinates; `(240x240)` is the tile size used; `1120x2508` is the resolution of an image after transformation; `JPEG/RGB Q=92` is the compression algorithm and the color format used. The `Q=92` is the `JPEG` compression quality setting (larger `Q` means lower compression rate).

Let us now analyze `'Region_001_FOV_00041_DAPI_Gray.tif'` using the tool.

```
> C:\Users\path_to_data\data> tiffinfo Region_001_FOV_00041_FITC_Gray.tif
TIFF Directory at offset 0x10196e (1055086)
Image Width: 1392 Image Length: 1024
Resolution: 96, 96 pixels/inch
Bits/Sample: 8
Sample Format: unsigned integer
```

```
Compression Scheme: LZW
Photometric Interpretation: min-is-black
Orientation: row 0 top, col 0 lhs
Samples/Pixel: 1
Rows/Strip: 128
Planar Configuration: single image plane
PageName: Region_001_FOV_00041_FITC_Gray.tif
```

We can see that some of the technical details displayed, for example, `Compression Scheme`, `Photometric Interpretation`, `Samples/Pixel`, `Planar Configuration` are different. Some new data could be seen as well, being `Sample Format`, `Orientation`, `Rows/Strip`, etc. A few observations could be made from the given information:

- `Resolution` along with image dimensions provide information about the physical size of the image when printed or displayed.
- `Samples/Pixel` and `Sample Format` tell us that it is a grayscale (single-channel) image, providing 256 levels of intensity.
- `Orientation` is specified to ensure that the image is displayed correctly with the appropriate coordinate system.
- `Rows/Strip` means that the selected tif image is stored in memory as sections of data (strips).

Image Loading.

`Libtiff` is a Python library that wraps the `Libtiff` C/C++ library, enabling Python users to read and write TIFF files. The library provides essential features for handling TIFF images, such as file access, information tags, etc. TIFF files are converted into NumPy arrays for further processing. `Libtiff` library allows users to load whole images using the `'read_image()'` method, or sub-rectangles from TIFF file using the `'read_tile()'` method. Here is an example demonstrating how to load a whole TIFF image and a sub-rectangle of the image into a NumPy array. The following Python code fragments can be accessed in `image_io.py` file located in `code/exercise_1/functions` directory.

1. Load a whole image: To load a whole image, `'open()'` method is used to load the desired TIFF file, and `'read_image()'` method is used to read the data to a NumPy array.

```
from libtiff import TIFF
def read_image(file_path):
    # Load TIFF file data
    tifImg = TIFF.open(file_path)
```

```
    # Read TIFF file data to an array
    image = TIFF.read_image(tifImg)
    tifImg.close()
    return image
```

2. Load a sub-image: To load a sub-image, X and Y coordinates (of the top-left corner of the tile) must be provided as arguments to the method `'read_one_tile()'`. In most cases, the convention is that (0, 0) corresponds to the top-left corner of an image, the X-axis increases from left to right, while the Y-axis - from top to bottom. If different, `'tiffinfo'` tool can reveal the orientation of a TIFF image using the `Orientation` tag.

```
def read_one_tile(file_path, x_coord, y_coord):
    # Load TIFF file data
    tifImg = TIFF.open(file_path)
    # Read one tile of TIFF file to an array
    tile = TIFF.read_one_tile(tifImg, x_coord, y_coord)
    tifImg.close()
    return tile
```

3. Visualize an image: To visualize an image using Matplotlib in Python, `'imshow()'` function is used, once the data is read into a NumPy array.

```
import matplotlib.pyplot as plt
def show_image(image):
    # Display the image
    plt.imshow(image, interpolation='nearest')
    plt.axis('off')
    plt.show()
```

In order to see how the `'show_image()'` function displays an image, the following lines of code

```
image = read_image('../../data/Kidney1.tif')
show_image(image)
```

were executed to display the 'Kidney1.tif' image. Figure 1 provides a visual representation of the function output.

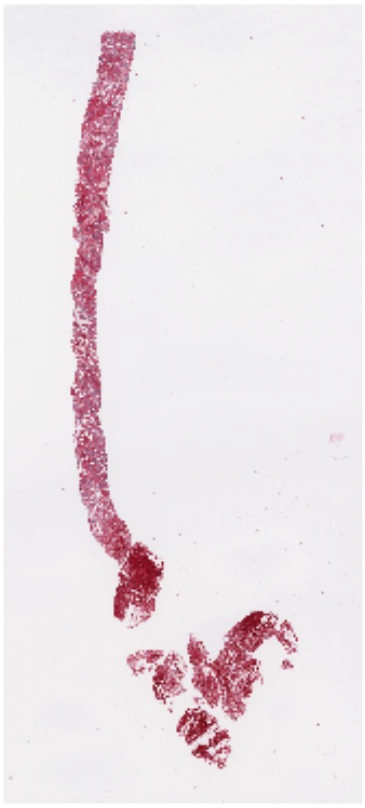


Figure 1: Visual representation of an image.

When displaying grayscale images using the `'imshow()'` function, unexpected colorization might occur, where shades of gray appear with added color. It can be seen in Figure 2. This happens because Matplotlib's `'imshow()'` function, by default, applies a colormap to the grayscale image. To verify if an image is grayscale, `'tiffinfo` tool may be used to extract the `'SamplesPerPixel'` information from the TIFF file.

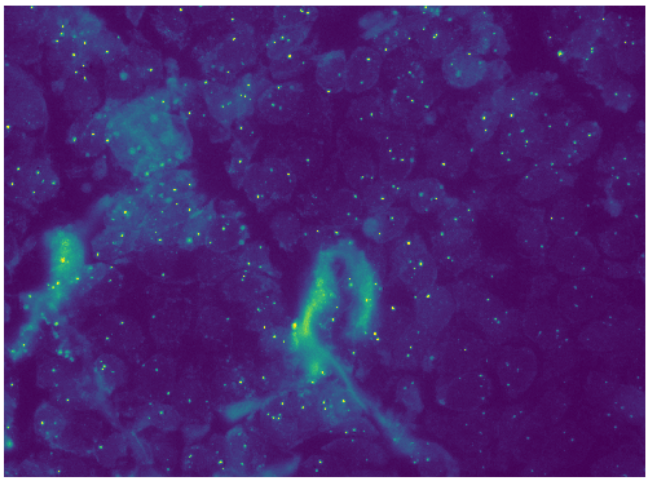


Figure 2: Unexpected colorization.

To achieve such outputs, one would have to run the following commands on the Ubuntu command line:

```
> cd code/exercise_1
> python load_visualize.py
```

4. Load and visualize sub-images: In order to examine SVS files, such as 'Kidney2 RGB2 20x.svs', the process of loading and visualizing sub-images involves opening a file and reading directories of that file. This can be achieved using `'SetDirectory()'` method.

```
tifImg_Kidney2 = TIFF.open('data/Kidney2_RGB2_20x.svs')
# Default directory 0
tifImg_Kidney2.SetDirectory(1)
image_Kidney2_dir1 = TIFF.read_image(tifImg_Kidney2)
show_image(image_Kidney2_dir1)
tifImg_Kidney2.close()
```

In a similar manner, 3 sub-images that can be seen in Figure 3, were obtained. To generate these results, you need to execute the following commands in the Ubuntu command line:

```
> cd code/exercise_1
> python load_subimages_svs.py
```

(a) Thumbnail image (b) Slide label image (c) Macro camera image

Figure 3: Different sub-images of the SVS file

The internal TIFF file structure within the SVS file format is designed to store extremely high-resolution images typically used in digital pathology. The first image in the SVS file is always the baseline image (full resolution). The image is tiled, so that extremely large images could be loaded and displayed efficiently by loading only those tiles that are absolutely necessary, rather than loading a single large image. SVS file has a few additional images that include the slide label, overview image, and a couple of smaller, scaled copies of the scanned image.

Image Combinations: The process of creating a contiguous RGB image from three fluorescence images involves several steps. First, the grayscale images are read from their respective TIFF files. Then, an empty RGB image is created with the same dimensions as the grayscale images to serve as a canvas for merging these images. The grayscale images are assigned to the respective RGB channels. Finally, a new TIFF file is opened for writing in write mode to save the combined image to disk. The resulting image can be seen in Figure 4.

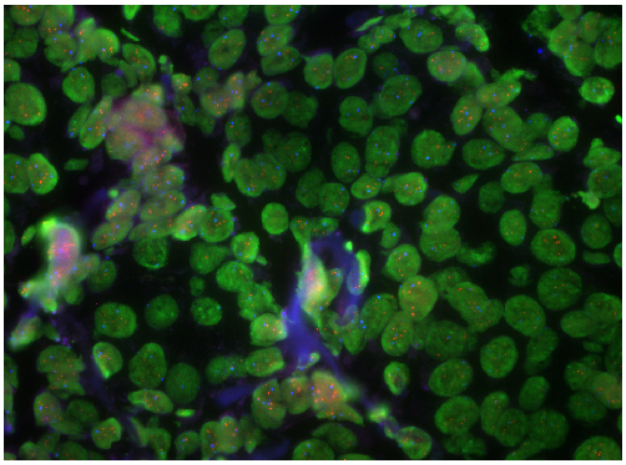


Figure 4: Combined RGB image.

To execute these steps, one can run a Python script on a command line in Ubuntu Linux system, using:

```
> cd code/exercise_1
> python combine_grayscale.py
```

2.2 Intensity Transformations

Purpose:
Gain practical experience with:

- Image intensity histograms.
- Image intensity transformations and how they affect the image by looking at the intensity histograms.
- Precision for intermediate calculations.

Power Law for Intensities.

Power law intensity transformation is a technique used to modify the contrast and brightness of an image. It is based on a mathematical power function that alters the pixel intensity values according to a law $T(r) = r^\gamma$, where $\gamma > 0$ is a predefined value, r is the initial intensity of a pixel and $T(r)$ is the intensity of a pixel after applying transformation T.

The implementation of the power law intensity transformation mainly involves two functions:

1. Lookup table generation. A lookup table is created for the power law transformation. The table maps each possible input intensity (in a range from 0 to 255) to its transformed value calculated by the power law. The values are then normalized to fit within the 8-bit range.

```
def intensityLUT(transform, gamma=None, ...):
    ...
    # Check the type of transformation requested
    if transform=='powerlaw':
        # Create a lookup table for powerlaw transformation
        lut = np.power(np.arange(0, 256, 1), gamma)
        lut = (255 * (lut - lut.min()) / (lut.max() - lut.min())).astype(np.uint8)
    ...
    return lut
```

2. Transformation. In this function, a duplicate is created in order to avoid modifying the original data. Power law transformation is applied to the duplicate image by mapping each pixel's intensity to the corresponding value in the lookup table.

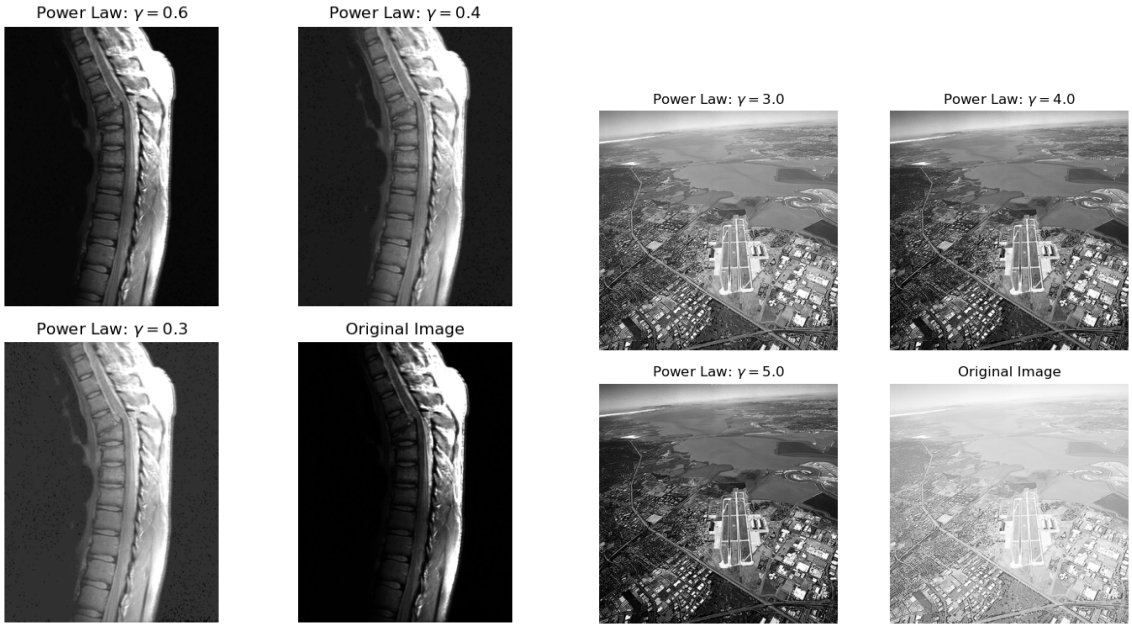
```
def intensityPowerlawLUT(image, gamma):
    # Create a duplicate of original image
    duplicate_image = np.copy(image)
    # Create a lookup table for the powerlaw transform
    lut = intensityLUT(transform='powerlaw',gamma=gamma)
    # Apply the power law transform using a lookup table
    duplicate_image = lut[image]
    return duplicate_image
```

It is important to note that using integer values for intermediate calculations isn't possible in this context. The reason is that the power law function, when applied, transforms integer values into real numbers. Therefore, it is necessary to use floating-point values (`float64`) for these calculations.

- $\gamma < 1$. When γ is set to a value less than 1, the power law transformation increases the image's brightness, as a narrow range of low-intensity values is mapped to a larger range. It

is great for enhancing image features and details but may lead to exaggeration in very bright regions. Figure 5(a) visualizes the effect of $\gamma < 1$, where brightness correction is made to highlight darker areas of the image.

- $\gamma > 1$. When γ is greater than 1, the power law transformation decreases the brightness of an image, because high-intensity values are mapped to a wider range. It enhances the darker details, leading to a more vivid appearance of an image. Figure 5(b) displays the effect of $\gamma > 1$, where the 'washed-out' appearance is removed using the transformation,



(a) Contrast enhancement (b) Brightness correction

Figure 5: Power law transformations with different γ values

To obtain such results of power law transformations, the following lines must be executed in the Ubuntu command line:

```
> cd code/exercise_2
> python intensity_powerlaw.py
```

Piece-wise Linear Intensity Transformations.

Piece-wise linear intensity transformations are employed for enhancing or modifying the contrast and brightness of an image. The technique involves dividing the intensity range into segments or pieces and applying different linear transformations for each segment. These linear transformations stretch or squish the range of intensity values, allowing for adjustments in image contrast and brightness. Piece-wise linear transformations also include thresholding, where certain intensity values are mapped to specific desired values, creating binary regions within the image.

- Histogram stretching. Figure 6 displays an example of histogram stretching to enhance the contrast of a washed-out image. A piece-wise linear function is constructed, within each segment of it, the pixel intensities are adjusted linearly to a new range.

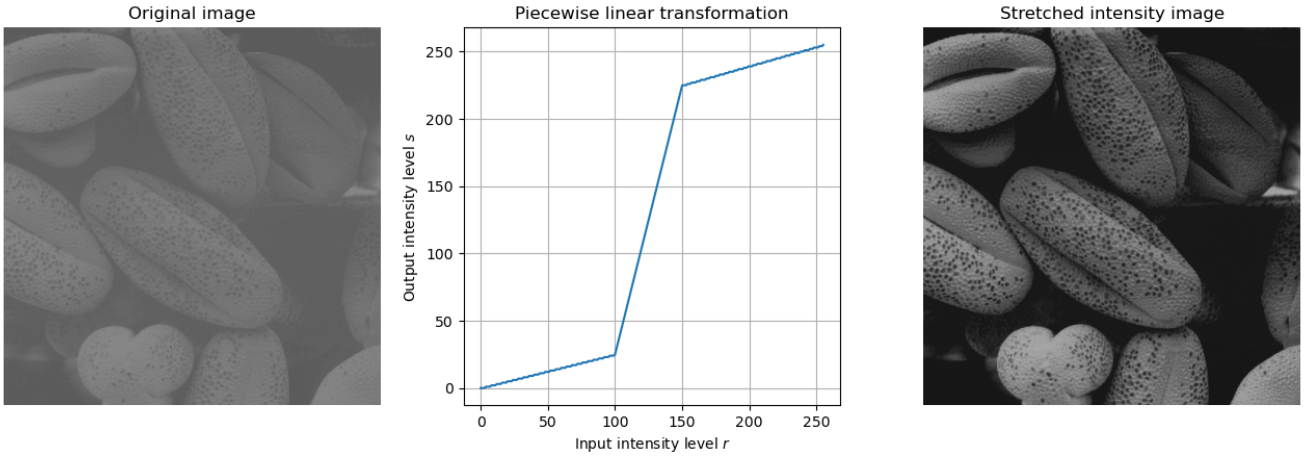


Figure 6: Piece-wise linear transformation.

Such output may be achieved with the following commands:

```
> cd code/exercise_2
> python intensity_piecewise_linear.py
```

- Thresholding. Figure 7 displays the result of a threshold transformation applied to a low-contrast image. By applying thresholding, specific regions of interest may be separated from the background, enhancing their visibility.

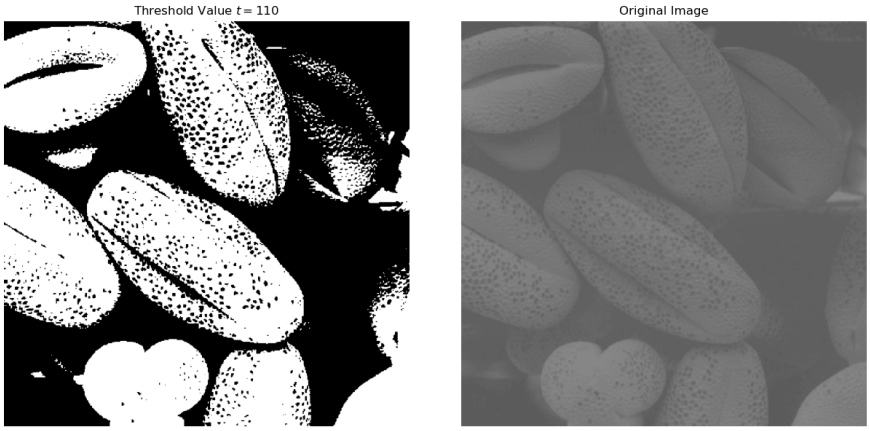


Figure 7: Threshold intensity transformation.

To obtain these results, one can use the following Ubuntu commands:

```
> cd code/exercise_2
> python intensity_threshold.py
```

Histogram Calculation.

A histogram of an image is a graphical representation that shows the frequency or distribution of pixel intensity values within the image. To calculate a histogram for a selected image, one would typically iterate through the image's pixels and count the occurrences of each possible intensity value, ranging from 0 to 255. Histogram calculation can be implemented using the following function:

```
def get_intensity_count(image):
    # Store the count of each intensity value [0-255] of an image
    histogram = np.zeros(256).astype(int)
    for pixel_intensity in image.flatten():
        histogram[pixel_intensity] += 1
    return histogram
```

In order to visualize a histogram, one would need to employ a function, that displays the calculated histogram values.

```
def plot_histogram(image):
    histogram = get_intensity_count(image)
    plt.bar(np.arange(0, 256, 1, dtype=np.uint8), histogram, width=1.0)
    plt.title('Histogram')
    plt.xlabel('Intensity Values')
    plt.ylabel('Frequency')
    plt.grid(axis='y')
```

This function uses histogram data to create a bar chart.

Histogram Normalization.

Histogram normalization is a technique used to adjust pixel intensities in an image so that they span the entire intensity range from 0 to 255. This process involves transforming the original histogram to enhance contrast and visibility of image details. Based on theoretical analysis, histogram normalization transformation exists and is expressed as follows:

$$T(r_k) = (L-1) \cdot \mathrm{CDF}_r(r_k) = (L-1) \cdot \sum_{j=0}^{k} p_j(r_j) \tag{1}$$

Here, $T(r_k)$ represents the transformed pixel intensity value, $CDF_r(r_k)$ is the cumulative distribution function of the original image's intensity values, $p_j(r_j)$ is a discretized probability distribution function of the original image's intensity values, and $(L-1)$ is the highest intensity value (255 for 8-bit images). By applying the transformation to the intensity values of an original image, one would obtain, that a distribution of the intensity values of the transformed image is close to that of a uniform distribution.

In order to implement histogram normalization, probability density, and cumulative distribution functions must be obtained:

```
def calculate_pdf(image):
    histogram = get_intensity_count(image)
    pdf = histogram / len(image.flatten())
    return pdf


def calculate_cdf(pdf):
    cdf = np.zeros(256)
    for i in range(256):
        cdf[i] = np.sum(pdf[: i])
    return cdf
```

After that, a lookup table may be implemented

```
def intensityLUT(transform, cdf=None, ...):
    ...
    # Check the type of transformation requested
    if transform=='equalization':
        # Create a lookup table of possible cdf values for histogram equalization
        lut = (255 * cdf).astype(int)
    ...
    return lut
```

Full implementation of the histogram normalization transform may be accessed within a python script `'histogram_normalization.py'`, located in a sub-directory `code/exercise_2`. The resulting Figure 8 is obtained, when the following line in Ubuntu command window is run

```
> cd code/exercise_2
> python histogram_normalization.py
```

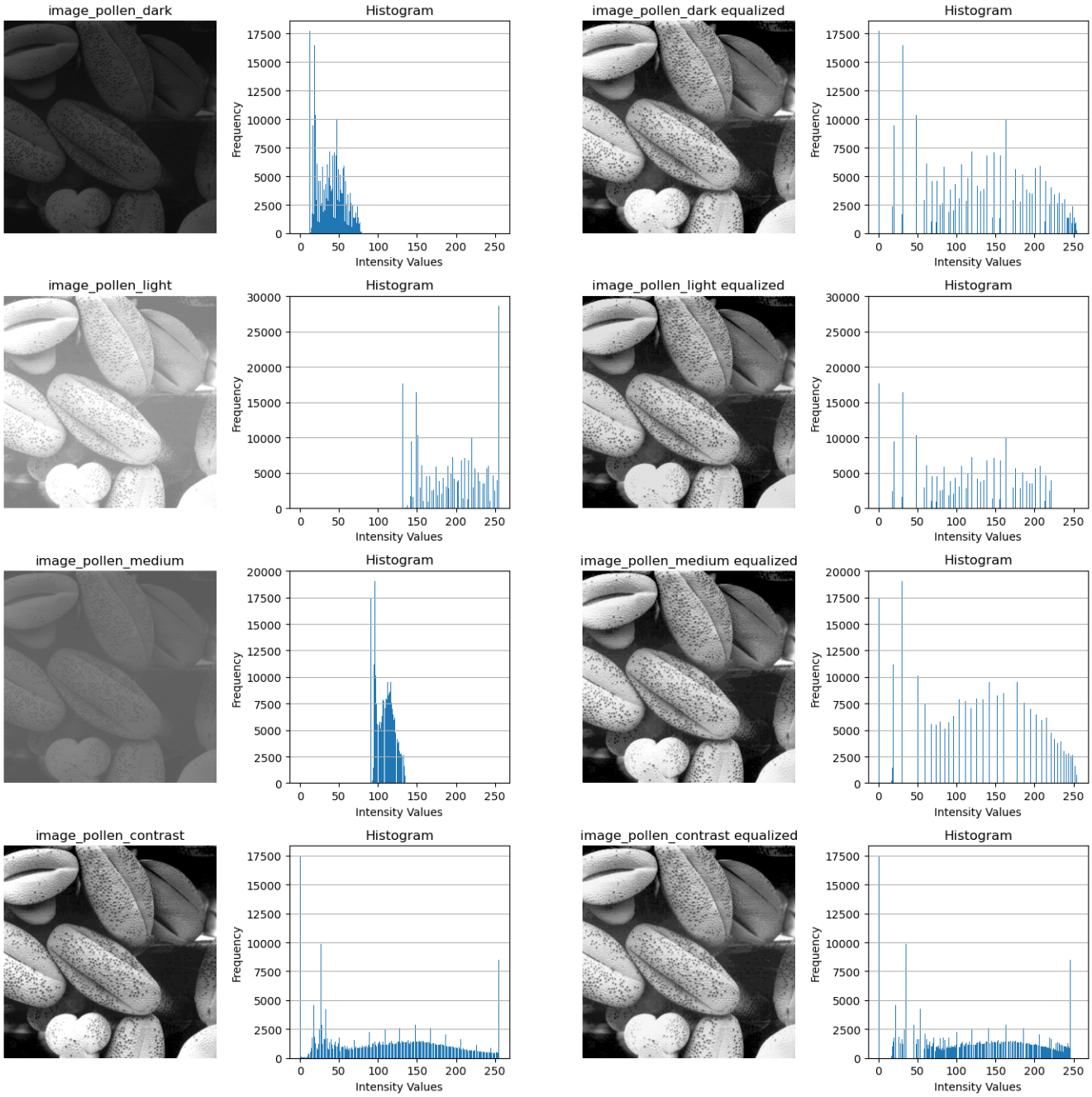


Figure 8: Histogram normalization.

Lookup Table Intensity Transformations.

Evaluating the intensity transformation from each pixel and using a lookup table are two different approaches to applying image intensity transformations. In this assignment, each one of the intensity transformations used had lookup tables implemented. A function that generates a lookup table for each transformation is defined. It can be found in the script `'intensityLUT.py'` that is inside `code/exercise_2/functions`. However, lookup table technique is not the only choice and one must also consider pixel-wise evaluation of the transforms. Here are the key differences

between the two methods:

1. Pixel-wise evaluation.

- The intensity transformation is applied to each pixel in the image individually.
- Method can handle dynamic transformations where the transform function may change for different pixels.

2. Lookup table.

- A precomputed lookup table is used to store the mapping of input intensity values to output intensity values.
- Computationally efficient and faster, especially for complex transformations, as it eliminates the need for repeated calculations.

The choice between these methods is completely dependent on the specific transformation and the image data. Lookup tables work effectively for transformations where the relationship between the input and output intensity values is constant across the entire image. However, there are image/pixel formats that may not be suitable for lookup table implementation, such as floating-point pixel formats. Lookup tables are specifically designed for integer-based pixel formats as there is a relatively small amount of values to be computed and stored. In the case of floats, intensity values are from a significantly larger domain, therefore, storing them would require an impractical amount of memory. In such cases, pixel-wise calculations would be a better choice.

2.3 Multistep Image Processing and Spatial Filtering

Purpose.
Gain practical experience with:

- Linear and non-linear filtering in the spatial domain
- Multi-step image processing

Pixel Format Conversion.

Converting between 8-bit and float formats is a common operation in image processing. That is because many spatial filtering (or other) techniques work with real numbers (float format) rather than integer values. However, for the correct display of images, they must be transformed back to 8-bit format, ensuring that pixel intensity values fall within the [0, 255] range.

To convert an 8-bit image to a float image, the following function is used. This function divides the 8-bit pixel intensity values by 255.0, ensuring that the resulting float values are in the range [0, 1].

```
def toFloat(image):
# Convert pixel intensity values to float type
converted_image = image.astype(float) / 255.0
return converted_image
```

To convert a float image to an 8-bit image, the provided function is employed. It offers two modes of conversion - minmax and truncate. In minmax mode, the image is scaled to the full [0, 255] range, minimum float value maps to 0, and maximum - to 255. In truncate mode, values outside of the range [0, 1] are clipped (values below 0 are set to 0, and values above 1 are set to 1). The function then scales the clipped values to the [0, 255] range.

```
def to8bit(image, mode='minmax'):
    # Min-Max scaling: Scale the image to the full [0, 255] range
    if mode == 'minmax':
        converted_image = (255 * (image - np.min(image)) / (np.max(image) - np.min(image)))
    # Truncate mode: Clip values to the [0, 255] range
    if mode == 'truncate':
        converted_image = (255 * np.clip(image, 0, 1)).astype(np.uint8)
    return converted_image
```

The code is from a Python script named `'format_conversion.py'` located in the sub-directory `code/exercise_3/functions`.

Image Blurring.

Average block filtering is one of the simplest linear filtering techniques. It is used to blur or reduce noise in an image. It operates by averaging the pixel values in a local neighborhood around each pixel, replacing the pixel's value with a computed average. A block of average values, referred to as a filter, needs to be generated beforehand and then applied to an image using a convolution (or correlation in this case) operation. Figure 9 serves as an illustration of the average block filtering technique.

Executing the provided Ubuntu command lines produces the resulting Figure 9

```
> cd code/exercise_3
> python block_average_blur.py
```

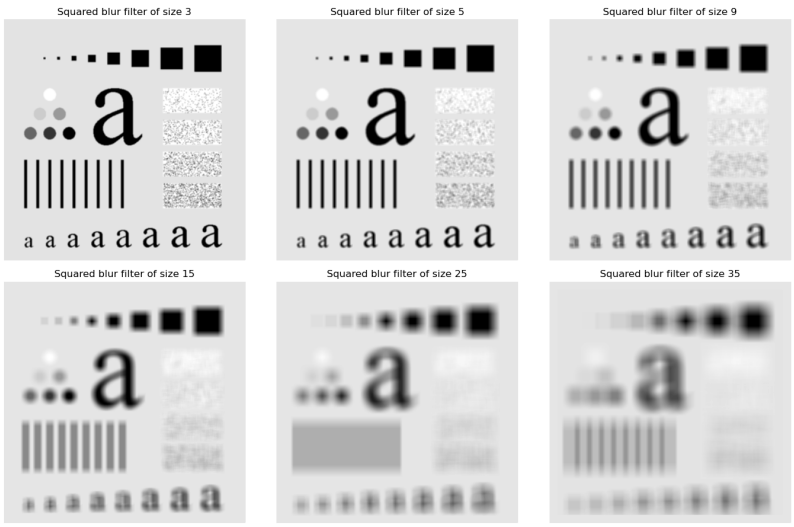


Figure 9: Blurring with average block filter.

Image Sharpening.

Image sharpening is a linear filtering method, that aims to emphasize transitions in intensity, enhancing the clarity and definition of edges and fine details within an image. Two common image sharpening techniques we'll explore are unsharp masking and Laplacian sharpening. Unsharp masking involves creating a blurred version of the original image, subtracting it from the original to retain edge information, and then summing the amplified result with the original to enhance details. This method is effective in increasing the overall sharpness of an image. The effectiveness in increasing image clarity can be seen in Figure 10.

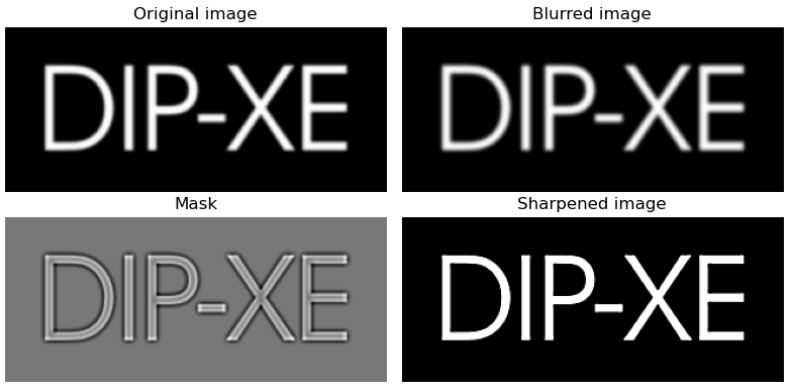


Figure 10: Unsharp masking.

Laplacian sharpening, on the other hand, emphasizes edges by applying a Laplacian filter to the image. The Laplacian filter is constructed by taking the second derivative of an image. Mathematically, it's computed by applying the discrete version Laplace operator to each pixel in the image. This filter highlights regions of rapid intensity change, resulting in sharper edges. The improved overall image quality can be seen in Figure 11.

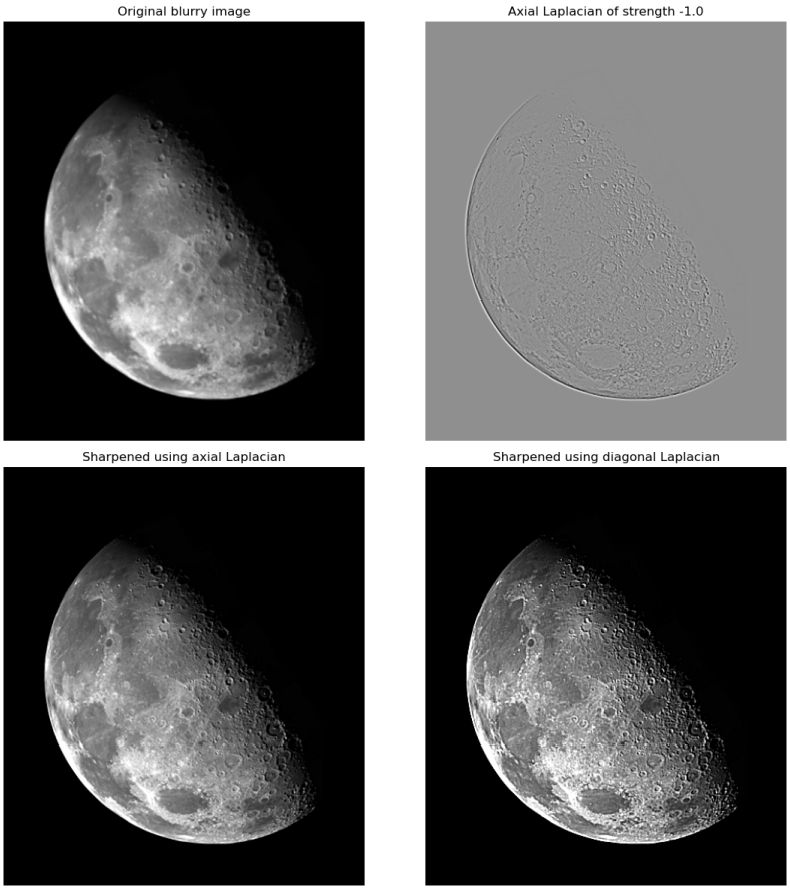


Figure 11: Laplacian sharpening.

Figures 10 and 11 can be generated by executing the following commands in Ubuntu Linux's command line:

```
> cd code/exercise_3
> python unsharp_masking.py
> python laplacian_sharpening.py
```

Sobel Operators.

Sobel operators are designed to highlight edges and transitions in intensity within images. The Sobel operators consist of two 3x3 filters, that resemble discretized gradients - one for detecting changes in horizontal intensity and the other for vertical changes. By convolving these filters with an image, two gradient images are obtained, representing the intensity changes in x and y directions. To measure the overall strength of intensity changes, a new image is obtained by calculating the magnitude of two gradient images. This magnitude image is particularly useful for identifying edges or other features in an image. As an example, when applying Sobel operators to an image of a contact lens, the resulting magnitude image highlights the edges and contours of the lens. By examining the magnitude image, seen in Figure 12, the defects of the lens become more pronounced.

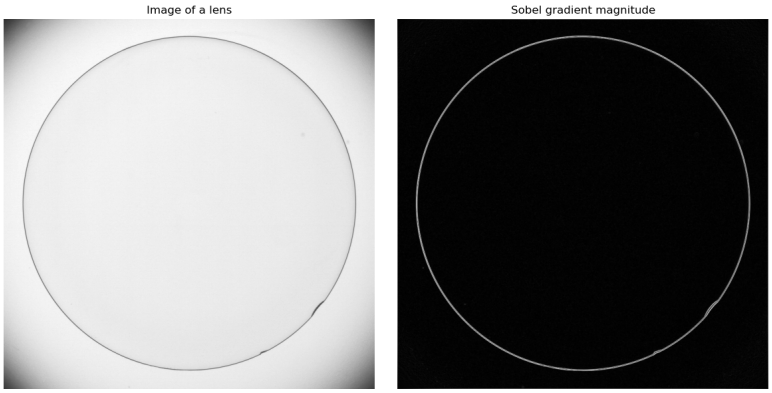


Figure 12: Sobel operators for edge detection.

In order to achieve the results shown in Figure 12, the following commands have to be executed:

```
> cd code/exercise_3
> python sobel_operators.py
```

Multistep Image Processing.

So far we have only focused on individual enhancement approaches. Most of the time, a given task will demand the application of multiple complementary methods in order to accomplish a specific result. In this section, we demonstrate how a combination of several techniques can help us tackle a challenging image enhancement task. Figure 13 displays steps a to h that were taken in order to achieve the desired results.

a: The initial step involves starting with the acquired image. In this case, it's a nuclear whole-body bone scan used for detecting diseases like bone infections and tumors.

b: To enhance fine details in the image, the Laplacian of the original image is computed. The Laplacian is a second-order derivative operator known for its ability to highlight fine detail.

c: In this step, we aim to sharpen the image by adding the Laplacian (computed in step b) to the original image. This process enhances overall image sharpness.

d: The Sobel gradient of the original image is computed, which provides information about the intensity transitions within the image. The Sobel operator is effective at highlighting edges and prominent features.

e: The Sobel image is smoothed using a 5x5 Gaussian filter. Smoothing helps reduce noise and retains important edge information, making it suitable for subsequent processing.

f: A mask is created by multiplying the sharpened image from step c with the smoothed Sobel image from step e. This step combines the strengths of both Laplacian and Sobel processing.

g: To obtain a sharpened image, we add the result from step f to the original image. This process results in improved sharpness in various parts of the image, such as the ribs, spinal cord, pelvis, and skull.

h: The final step involves applying a power-law transformation to expand the dynamic range of the intensity levels. This transformation helps reveal additional image details. It's important to use a value of $\gamma < 1$ to spread the intensity levels effectively. In this particular case, $\gamma = 0.4$ is used.

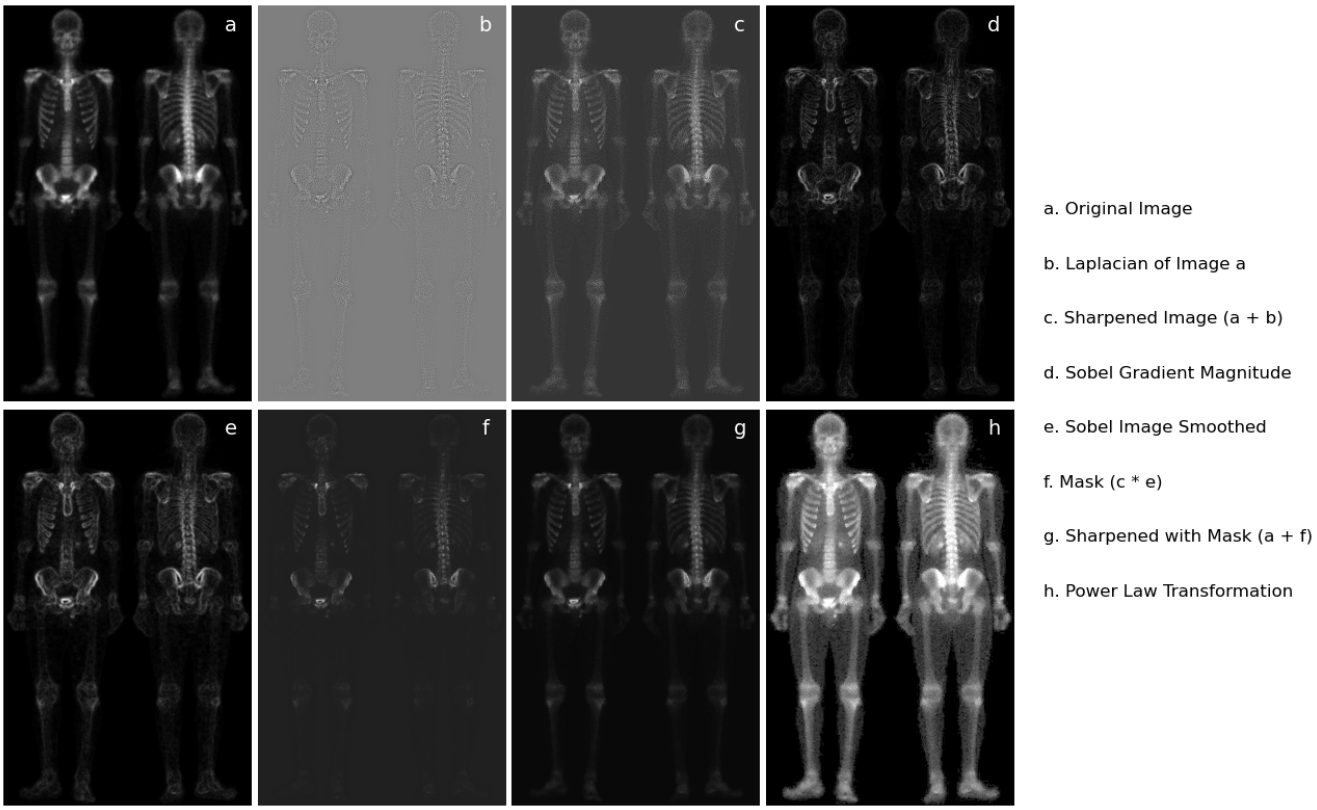


Figure 13: Multistep processing of whole body bone scan image.

By following these steps, we have significantly enhanced the original image, bringing out skeletal details and making it more suitable for disease detection and analysis. The multistep processing is a sophisticated way to employ the intensity transformation and spatial filtering tools, where the objective of the task is to obtain an image with a higher content of visual detail.

The results of the multistep image enhancement can be obtained by executing the following commands on the Ubuntu command line:

```
> cd code/exercise_3
> python multistep_processing.py
```