Exercise3 - Pixelwise operations

In this exercise you will learn to perform pixelwise operations using Python.

Learning Objectives

After completing this exercise, the student should be able to do the following:

1. Convert from unsigned byte to float images using the scikit-image function `img_as_float`
2. Convert from float to unsigned byte images using the scikit-image function `img_as_ubyte`
3. Implement and test a function that can do linear histogram stretching of a grey level image.
4. Implement and test a function that can perform gamma mapping of a grey level image.
5. Implement and test a function that can threshold a grey scale image.
6. Use Otsu's automatic method to compute an optimal threshold that seperates foreground and background
7. Perform RGB thresholding in a color image.
8. Convert a RGB image to HSV using the function `rgsb2hsv` from the `skimage.color` package.
9. Visualise individual H, S, V components of a color image.
10. Implement and test thresholding in HSV space.
11. Implement and test a program that can do perform pixelwise operations on a video stream

Installing Python packages

In this exercise, we will be using both scikit-image and OpenCV. You should have both libraries installed, else instructions can be found in the previous exercises.

We will use the virtual environment from the previous exercise (`course02502`).

Exercise data and material

The data and material needed for this exercise can be found here: (https://github.com/RasmusRPaulsen/DTUImageAnalysis/blob/main/exercises/ex3-PixelwiseOperations/data/)

Explorative data analysis

First we will be working with an X-ray image of the human vertebra, `vertebra.png`. This type of images can for example be used for diagnosis

of osteoporosis. A symptom is the so-called vertebral compression fracture. However, the diagnosis is very difficult to do based on x-rays alone.

Exercise 1: *Start by reading the image and inspect the histogram. Is it a* bimodal* histogram? Do you think it will be possible to segment it so only the bones are visible?*

Exercise 2: *Compute the minimum and maximum values of the image. Is the full scale of the gray-scale spectrum used or can we enhance the appearance of the image?*

Pixel type conversions

Before going further, we need to understand how to convert between between pixel types and what should be considered. A comphrehensive guide can be found here (it is not mandatory reading, we just use some highlights). One important point is that we should avoid using the `astype` function on images.

Conversion from unsigned byte to float image

In *unsigned byte* images, the possible pixel value range is [0, 255]. When converting an *unsigned byte* image to a *float* image, the possible pixel value range will be [0, 1]. When you use Python skimage function `img_as_float` on an *unsigned byte* image, it will automatically divide all pixel values with 255.

Exercise 3: *Add an import statement to your script:*

```
from skimage.util import img_as_float
from skimage.util import img_as_ubyte
```

Read the image `vertebra.png` and compute and show the minumum and maximum values.

Use `img_as_float` to compute a new float version of your input image. Compute the minimum and maximum values of this float image. Can you verify that the float image is equal to the original image, where each pixel value is divided by 255?

Conversion from float image to unsigned byte image

As stated above, an (unsigned) float image can have pixel values in [0, 1]. When using the Python skimage function `img_as_ubyte` on an (unsigned) float image, it will multiply all values with 255 before converting into a byte. Remember that all decimal number will be converted into integers by this, and some information might be lost.

Exercise 4: *Use `img_as_ubyte` on the float image you computed in the previous exercise. Compute the Compute the minimum and maximum values of this image. Are they as expected?*

Histogram stretching

You should implement a function, that automatically stretches the histogram of an image. In other words, the function should create a new image, where the pixel values are changed so the histogram of the output image is *optimal*. Here *optimal* means, that the minimum value is 0 and the maximum value is 255. It should be based on the *linear histogram stretching* equation:

$$g(x,y) = \frac{v_{\text{max,d}} - v_{\text{min,d}}}{v_{\text{max}} - v_{\text{min}}}(f(x,y) - v_{\text{min}}) + v_{\text{min,d}} \ .$$

Here $f(x,y)$ is the input pixel value and $g(x,y)$ is the output pixel value, $v_{\text{max,d}}$ and $v_{\text{min,d}}$ are the desired minimum and maximum values (0 and 255) and v_{max} and v_{min} are the current minumum and maximum values.

Exercise 5: *Implement a Python function called* ***histogram_stretch****. It can, for example, follow this example:*

```
def histogram_stretch(img_in):
    """
    Stretches the histogram of an image
    :param img_in: Input image
    :return: Image, where the histogram is stretched so the min values is 0 and the maximum
    """
    # img_as_float will divide all pixel values with 255.0
    img_float = img_as_float(img_in)
    min_val = img_float.min()
    max_val = img_float.max()
    min_desired = 0.0
    max_desired = 1.0

    # Do something here

    # img_as_ubyte will multiply all pixel values with 255.0 before converting to unsigned
    return img_as_ubyte(img_out)
```

Exercise 6: *Test your* ***histogram_stretch*** *on the* ***vertebra.png*** *image. Show the image before and after the histogram stretching. What changes do you notice in the image? Are the important structures more visible?*

Non-linear pixel value mapping

The goal is to implement and test a function that performs a γ-mapping of pixel values:

$$g(x,y) = f(x,y)^{\gamma} \ .$$

You can use the *Numpy* function `power` to compute the actual mapping function.

Exercise 7: *Implement a function,* `gamma_map(img, gamma)`*, that:*

1. Converts the input image to float
2. Do the gamma mapping on the pixel values
3. Returns the resulting image as an unsigned byte image.

Exercise 8: *Test your* `gamma_map` *function on the vertebra image or another image of your choice. Try different values of* γ*, for example 0.5 and 2.0. Show the resuling image together with the input image. Can you see the differences in the images?*

Image segmentation by thresholding

Now we will try to implement some functions that can seperate an image into *segments*. In this exercise, we aim at seperating the *background* from the *foreground* by setting a threshold in a gray scale image or several thresholds in color images.

Exercise 9: *Implement a function,* `threshold_image` *:*

```
def threshold_image(img_in, thres):
    """
    Apply a threshold in an image and return the resulting image
    :param img_in: Input image
    :param thres: The treshold value in the range [0, 255]
    :return: Resulting image (unsigned byte) where background is 0 and foreground is 255
    """
```

Remember to use `img_as_ubyte` when returning the resulting image.

Exercise 10: *Test your* `threshold_image` *function on the vertebra image with different thresholds. It is probably not possible to find a threshold that seperates the bones from the background, but can you find a threshold that seperates the human from the background?*

Automatic thresholds using Otsu's method

An optimal threshold can be estimated using *Otsu's method*. This method finds the threshold, that minimizes the combined variance of the foreground and background.

Exercise 11: *Read the documentation of Otsu's method and use it to compute and apply a threshold to the vertebra image.*

Remember to import the method:

```
from skimage.filters import threshold_otsu
```

How does the threshold and the result compare to your manually found threshold?

Exercicse 12: *Use your camera to take some pictures of yourself or a friend. Try to take a picture on a dark background. Convert the image to grayscale and try to find a threshold that creates a* ***silhouette*** *image (an image where the head is all white and the background black).*

Alternatively, you can use the supplied photo **dark_background.png** found in the exercise data.

Color thresholding in the RGB color space

In the following, we will make a simple system for road-sign detection. Start by reading the image **DTUSigns2.jpg** found in the exercise data. We want to make a system that do a *segmentation* of the image - meaning that a new binary image is created, where the foreground pixels correspond to the sign we want to detect.

We do that by tresholding the colour-channels individually. This code segments out the blue sign:

```
r_comp = im_org[:, :, 0]
g_comp = im_org[:, :, 1]
b_comp = im_org[:, :, 2]
segm_blue = (r_comp < 10) & (g_comp > 85) & (g_comp < 105) & \
            (b_comp > 180) & (b_comp < 200)
```

Exercise 13: *Create a function* ***detect_dtu_signs*** *that takes as input a color image and returns an image, where the blue sign is identified by foreground pixels.*

Exercise 14: *Extend your* ***detect_dtu_signs*** *function so it can also detect red signs. You can add an argument to the function, that tells which color it should look for. You should use one of the explorative image tools to find out what the typical RGB values are in the red signs.*

Color thresholding in the HSV color space

Sometimes it gives better segmentation results when the tresholding is done in HSI (also known as HSV - hue, saturation, value) space. Start by reading the **DTUSigns2.jpg** image, convert it to HSV and show the hue and value (from here):

```
hsv_img = color.rgb2hsv(im_org)
hue_img = hsv_img[:, :, 0]
value_img = hsv_img[:, :, 2]
fig, (ax0, ax1, ax2) = plt.subplots(ncols=3, figsize=(8, 2))
ax0.imshow(im_org)
ax0.set_title("RGB image")
```

```
ax0.axis('off')
ax1.imshow(hue_img, cmap='hsv')
ax1.set_title("Hue channel")
ax1.axis('off')
ax2.imshow(value_img)
ax2.set_title("Value channel")
ax2.axis('off')

fig.tight_layout()
io.show()
```

Exercise 15: *Now make a sign segmentation function using tresholding in HSV space and locate both the blue and the red sign.*

Real time pixelwise operations on videos

In the exercise material, there is a Python script using OpenCV that:

1. Connects to a camera
2. Acquire images, converts them to gray-scale
3. Do a simple processing on the gray-scale (inversion) or the colour image (inversion of the red channel)
4. Computes the frames per second (fps) and shows it on an image.
5. Shows input and resulting images in windows.
6. Checks if the key `q` has been pressed and stops the program if it is pressed.

It is possible to use a mobile phone as a remote camera by following the instructions in exercise 2b.

Exercise 16: *Run the program from the exercise material and see if it shows the expected results?*

Exercise 17: *Change the gray-scale processing in the exercise material script to be for example thresholding, gamma mapping or something else. Do you get the visual result that you expected?*

Exercise 18: *Real time detection of DTU signs*

Change the rgb-scale processing in the exercise material script so it does a color threshold in either RGB or HSV space. The goal is to make a program that can *see* DTU street signs. The output should be a binary image, where the pixels of the sign is foreground. Later in the course, we will learn how to remove the noise pixels.